UNDERSTANDING BRITTLENESS TO VARIABLE RENAMING IN DECODER-ONLY CODE MODELS

Anonymous authors
Paper under double-blind review

ABSTRACT

Transformer-based code models increasingly support compiler and programming-language tasks, including program analysis, performance prediction, and code generation. Recent work has shown that Transformer-based models for code are sensitive to simple semantics-preserving edits such as variable renaming. For example, a simple variable renaming can induce sizable prediction shifts in these models. The mechanisms that underlie this brittleness to variable renaming are not well understood, though. In this paper, we aim to investigate such mechanisms. In particular, we investigate where failures arise in the model, which components are involved in such failures, and whether the diversity of variable names in the training set affects the robustness of models to variable renaming. To enable our controlled experiments, we introduce TINYTRACER, a framework that generates synthetic Python programs and their execution traces under controlled training and test distributions. We use this framework to train small decoder-only Transformers from scratch to predict execution traces, while varying the diversity of variable names in the training set. We test the robustness of the trained models by taking programs that the models trace correctly and consistently replacing their variable names with other names. In these small scale models trained with low variable-name diversity, the first error in a failed trace most often occurs where a variable name should end: the model generates another identifier character instead of the required delimiter. Increasing variable-name diversity during training reduces these termination errors and improves robustness to variable names in this setting. We identify attention heads that attend to delimiters following earlier occurrences of a variable name, and find that ablating these heads reduces accuracy at name endings, supporting a causal role for these heads in identifying where variable names end. Higher training-name diversity is also associated with stronger attention to these delimiters. Experiments with another larger-scale decoder-only model (158M-parameter) trained on real Python code also show improved robustness with increased variable name diversity. Complementary analyses identify a shared difficulty in generating unfamiliar names, with errors concentrated at the initial tokens of variable names.

1 INTRODUCTION

Transformer-based code models are increasingly used in compilers and programming-language systems, where they support tasks such as optimization heuristics, performance prediction, program analysis, and code generation. In many of these settings, programs that are semantically equivalent but are slightly different syntactically (e.g., two equivalent programs that only differ in variable names) should induce the same behavior in the models. A growing body of work shows that this is not the case, though. Research has shown that Transformer-based models for code exhibit systematic degradation when semantics-preserving edits such as variable renaming are applied (Henkel et al., 2022; Bielik & Vechev, 2020; Orvalho & Kwiatkowska, 2025).

The mechanisms underlying this brittleness remain poorly understood, though. In this paper, we aim to investigate such mechanisms. We investigate where failures occur during prediction, which decisions the model fails to make, which model components contribute to them, and whether the diversity of variable names in the training data affects robustness to renaming.

Answering these questions with large pretrained models is difficult: their training corpora are often opaque, and their scale makes controlled retraining costly. We therefore introduce TINYTRACER, a framework that generates random Python programs and their execution traces with precise control over the training and test distributions. This control allows specifying which variable names appear in each split and varying their diversity while holding other aspects of the data distribution fixed.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

Our study begins with a small decoder-only Transformer trained from scratch to predict these execution traces (we call it the *tracing model*). Its token-by-token generation supports fine-grained error analysis, allowing us to identify the first incorrectly predicted token and examine the model's prediction at that step.

Using TINYTRACER, we investigate whether the limited diversity of variable names in the training set contributes to brittleness to variable renaming. We vary variable-name diversity under controlled conditions and test whether robustness changes systematically as diversity increases. We test robustness by taking programs that each model traces correctly, consistently replacing their variable names with names unseen during training, and evaluating the model on the renamed programs.

Our analyses show that, in the tracing model trained under low-diversity, the first error in a failed trace after renaming most often occurs where a variable name should end: the model generates another identifier character instead of the required delimiter. Increasing variable-name diversity during training reduces these termination errors and improves robustness to unseen variable names.

Our interpretability analyses on this model identify attention heads that attend to delimiters following earlier occurrences of a variable name. Ablating these heads reduces accuracy at name endings, supporting a causal role for these heads in identifying where variable names end. Redirecting their attention to the delimiter following an earlier occurrence of the name also repairs a subset of incorrect name-ending predictions. We also find that having higher variable-name diversity when training this model is associated with stronger attention to these delimiters.

To further validate our findings on a larger code model, we train `tiny_starcoder_py`, a modern 158M-parameter decoder-only Transformer Li et al. (2023) pretrained on GitHub Python code von Werra (2021). We find that increasing variable-name diversity in the training data improves the model's robustness to variable renaming.

We also extend the identifier-generation and attention-head analyses to this larger-scale model (we refer to it as StarCoder). These analyses identify difficulty in generating variable names, with renaming-induced identifier-generation errors concentrated at the initial token of a variable name. Redirecting selected attention heads toward earlier occurrences of the target name repairs some initial-token errors, while directing them toward the other variables of the function shifts predictions toward those variables' initial tokens.

Our contribution is a controlled diagnosis of variable-renaming brittleness, combining variation in variable-name diversity in training with precise failure localization and causal tests of contextual retrieval in two code models. We identify difficulty reproducing unfamiliar variable names as a major observed failure mode under renaming in the two studied models. Failures concentrate at name endings in the tracing models and at initial name tokens in StarCoder. Ablation and attention-redirection experiments show that retrieval from earlier name occurrences contributes causally to these predictions and can repair some renaming-induced errors. We also show that low variable-name diversity in the training set contributes to brittleness to variable renaming in the studied models.

In summary, **our contributions are** as follows: **1)** We introduce TINYTRACER and use it to study brittleness to semantics-preserving variable renaming. **2)** We show that low variable-name diversity in the training set contributes to brittleness to variable renaming in the studied models. **3)** We show that, in the low-diversity models analyzed, models struggle to generate variable names after renaming, with failures concentrated at name endings in the tracing model and first name tokens in starcoder. **4)** Through ablation and attention redirection, we show that heads attending to earlier name occurrences or their delimiters contribute causally to identifier predictions, and that redirecting them repairs some renaming-induced errors.

2 OVERVIEW

In the TINYTRACER framework, the model is trained to perform the task of *code tracing*. In this task, the model takes as input a program annotated with its current execution state (value of each variable and a pointer to the last instruction that has been executed); It predicts as output the program annotated with the new execution state (as shown in Figure 1). Prediction of the new annotated program is done token by token (where generated tokens are fed back as input to the model).

Using TINYTRACER, we generate synthetic Python programs and their reference traces, and train small decoder-only Transformers from scratch to predict these traces. We use the tracing task to investigate how variable renaming affects a model's predictions. Variable renaming is a semantics-

preserving transformation of the code: variables are renamed in a way that preserves the original semantics of the code. A model that correctly traces the original program should therefore also produce the correct trace for its renamed version. However, prior work has shown that code models exhibit degradation when variable renaming is applied (Henkel et al., 2022; Bielik & Vechev, 2020; Orvalho & Kwiatkowska, 2025). We call this sensitivity brittleness to variable renaming. Our goal is to study the mechanisms underlying this brittleness.

We begin our investigation with the hypothesis that limited variable-name diversity during training contributes to brittleness. To test this hypothesis, we vary the number of distinct variable names in the training sets while keeping the other program-generation settings fixed. For each model, we take programs it traces correctly, replace their variable names with names unseen during training, and evaluate the predictions against the corresponding renamed reference traces. Greater variable-name diversity improves performance on these renamed programs, supporting limited diversity as a contributing factor to brittleness in our settings.

We then ask where the model's predictions break after variable renaming. To answer this, we examine the first incorrect prediction in each failed renamed trace. In the tracing model, the error in prediction most often occurs where a variable name should end: the model generates another identifier character instead of the required delimiter. For example, if the reference contains `abcd =`, the model might generate another letter after `abcd`, incorrectly extending the name. This observation motivates a closer examination of how the model decides whether to continue or end an identifier.

We then ask which components are involved in deciding where a variable name should end. An earlier occurrence of the same name provides a useful cue: the character that follows it marks its end. We identify attention heads that attend to this cue. Attention to a useful location alone does not establish that a head affects the prediction. We therefore disable the selected heads by setting their outputs to zero and measure accuracy at positions where variable names should end. We also disable equally sized sets of randomly selected heads as a control. We find that removing the selected heads causes larger accuracy drops at unfamiliar name endings than the random controls, supporting a causal contribution to these predictions.

We next ask how diversity relates to attention to these delimiters. We show that greater training-name diversity is associated with stronger attention to delimiters following earlier occurrences of a variable name. Alongside improved robustness, this pattern is consistent with better use of the current program to recognize where unfamiliar names end.

Finally, we test the diversity intervention at a larger model scale (the StarCoder model). We find that increasing variable name diversity in training improves robustness to variable renaming in this model also. We then repeat our mechanistic interpretability analyses on this model. We also find that the model has difficulty in generating unfamiliar names, but errors after renaming mainly occur at initial name tokens in the low-diversity case, while attention and ablation analyses identify the heads responsible for this behavior. Both our experiments, in the tracing model and in StarCoder, expose difficulties of the trained models to generate unseen names, although the dominant error position differs.

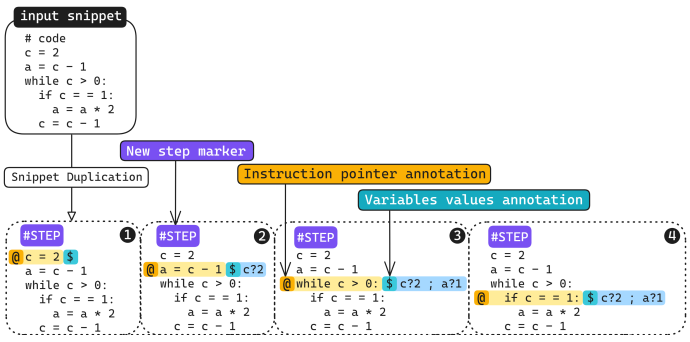

Figure 1: Illustration of the TinyTracing task. At each execution step (we show the first 4 steps), the entire input code snippet is duplicated and annotated with corresponding execution information: The instruction pointer value is represented by setting a special symbol (here `@`) to the left of the current line to be executed, and the variable states are presented as (key, value) pairs to the right of the current execution line. A special symbol (here `#STEP`) indicates the beginning of a new step (see Appendix A.3 for a full example).

3 THE TINYTRACER EXPERIMENTAL FRAMEWORK

We developed TINYTRACER to generate programs and execution traces under a controlled distribution. An explicit trace lets us locate a failure at a particular token or decision, rather than infer the cause from a wrong final answer; therefore it better fits our goal to understand why the model fails. We use this framework, its model (called the tracing model) and dataset for our small-scale experiments (Sections 4.1, 4.2, 4.3, and 4.4). For our validation on a larger-scale model (StarCoder), we use another model and dataset (Section 4.5).

TinyTracing Task We propose a controlled experimental setup in which models are trained to perform the task of *TinyTracing*. For a given input program (Figure 1-Top), the model predicts the full trace of execution of that input program (Figure 1-Bottom). The trace is composed of a sequence of steps where each step is a copy of the original program annotated with the state of variables and the instruction pointer at that step. The whole trace is generated token-by-token in an autoregressive way. By adjusting the complexity of the programming language and code snippets used for tracing, the task can be adapted to fit the computational budget of users, allowing cost-effective experiments.

Model Architecture We use NanoGPT Karpathy (2022): a small-scale, decoder-only Transformer architecture, trained from scratch, inspired by GPT-2. An overview figure of the model alongside more details on both the hyperparameters and tokenization can be found in the supplementary material (resp. Figure 10, Sec. A.9, and Sec. A.7).

Data Generation Tool To complete our experimentation setup, we need to be able to access arbitrary quantities of data with precisely controlled distributional characteristics. To this end, we propose a data generation tool, referred to as **TinyTrace-Generator**. This tool enables the synthesis of random code snippets that are expressed in a subset of Python and that follow a certain data distribution (defined using a set of constraints on the Python language). The tool also allows creating the execution trace of these snippets (tool described in Appendix A.5).

Dataset and Model Training One of the goals of TINYTRACER is to allow fast model training, enabling users to perform a large set of experiments fast. Early in the project, we trained on a rich subset of Python that included while loops. Training a competent tracer for that setting required a $\sim$65M-parameter model and several days of training, which slowed experimentation. Instead of that, we use a simpler subset of Python that includes: variable assignments, arithmetic operations, and conditionals only (without loops). We call this subset **TINY-α**. With this subset, we can train a tiny model to attain (near-)perfect tracing performance within a few hours, enabling rapid experiments.

For the tracing model, we use a dataset (TINY-α) that has 24 million Python snippets in the TINY-α subset of Python, with each snippet having its full trace. The datasets are fully synthesized using the TinyTrace-Generator described in Sec. A.5.1. Training of the tracing model is conducted on 4$\times$A100 (80GB) GPUs for one epoch per language, taking 2h to train.

Evaluation Method We evaluate the model using *Exact Match Accuracy*. For each test sample, the model autoregressively generates the complete execution trace given the code snippet as a prompt. A generated trace is considered correct only if it is a character-for-character match to the ground truth. Unless otherwise specified, the accuracy reflects testing the model on 1024 snippets.

4 EXPERIMENTS

4.1 DOES VARIABLE-NAME DIVERSITY AFFECT ROBUSTNESS TO VARIABLE RENAMING?

Our hypothesis is that low variable-name diversity in the training set is a contributing factor to brittleness to variable renaming. The intuition is straightforward. Many code datasets are dominated by identifiers that commonly appear in natural language, meaning that common variable names tend to recur and that only a thin slice of the possible identifier space is observed during training. This can encourage the model to rely on familiar names rather than learn that variable names mainly serve as placeholders for values. Building on this, we hypothesize that training a model on a dataset with a higher diversity in variable names produces models that are more robust to variable renaming.

Before arriving at this hypothesis, we explored others that did not hold up empirically. For example, we tested whether the problem could be resolved by scaling the amount of training data (without increasing variable-name diversity), and whether changing the positional encoding or tokenizer was helpful (Appendix C). This kind of hypothesis search would have been much harder to carry out directly at the scale of `tiny_starcoder_py` or in a full programming language such as Python: training would be far more expensive, iteration would be much slower, and negative results would

be harder to interpret because they could reflect many confounded factors at once, including task difficulty, language complexity, insufficient optimization, or insufficient data. In contrast, TINY-TRACER is deliberately designed so that a small model can reach high accuracy on a simple but semantically meaningful task with modest data and compute. This makes the framework useful as a hypothesis-search instrument: once the model has learned the task well, remaining failures are less likely to be explained by undertraining or task difficulty, and can more plausibly be attributed to the data-distribution factor under study.

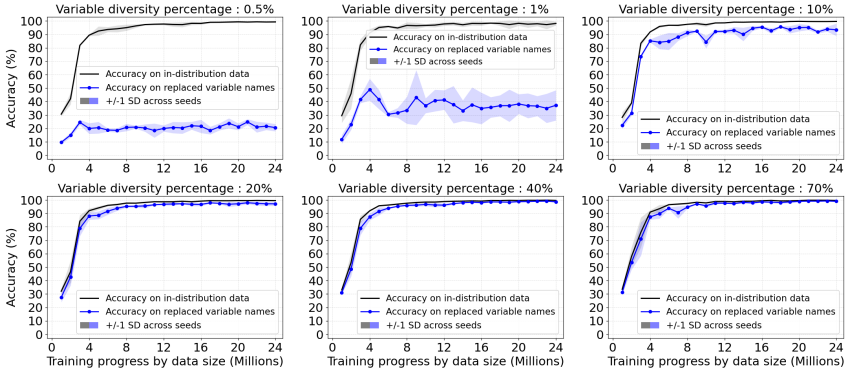

Figure 2: Accuracy during training for six models trained on datasets with different variable-name diversities. Training checkpoints are saved and evaluated every 1 million examples. Averaged over 3 training seeds.

Variable-name diversity. Given a dataset, let D_L be the number of distinct identifiers of length L that appear in that dataset. Let P_L be the number of all possible identifiers of length L. We define variable-name diversity at length L as $(D_L/P_L) \times 100\%$. In other words, this metric measures the fraction of unique identifiers of length L compared to the whole space of possible identifiers of length L. For example, if the variable-name diversity for length 2 is 50%, that means that half of all the possible variable names of length 2 are present in the dataset.

We say that the variable-name diversity of a dataset is D if the variable-name diversity at length L is D for all variable lengths L in the dataset except for length $L = 1$ (single-character variables). For single-character variables, we always include all the possible variable names (all 26 characters).

When we measure the variable-name diversity in realistic datasets, we find that it is often low. For example, in CodeSearchNet (Husain et al., 2019), the variable-name diversity is below 0.1% for identifier lengths of 4 characters or more. This is consistent with the fact that identifiers are typically derived from natural language: common names recur across many snippets, while only a small portion of the possible identifier space is used.

Testing the hypothesis using our framework We follow the training setup explained in Sec. A.8 and train six models on variants of the Tiny-α dataset. The only difference between these variants is in the variable-name diversity we use. We sweep the variable-name diversities over six values: 0.5%, 1%, 10%, 20%, 40%, and 70%. Therefore, we obtain six variants of the dataset, the first with a variable-name diversity of 0.5%, the second with a variable-name diversity of 1%, and so on. For each variant, we train a model. For each model trained, we evaluate on two test sets. The first is an in-distribution test set of 1024 snippets, each following the same distribution as the training set. The second dataset is constructed by only keeping the correctly traced snippets from the first dataset, then replacing all identifiers with names unseen during training. Because the total number of snippets is different, accuracy for both datasets is calculated out of the original 1024, in order to maintain a consistent baseline. This second set isolates robustness to variable renaming. This experiment is repeated 3 times, each time with a different training seed, then we display the average scores (alongside the standard deviation) in figure 2. The results show a clear pattern: as variable-name diversity increases, performance on unseen variable names improves steadily. At 0.5%, the gap between the in-distribution and renamed test sets is large. At 1% and 10%, the gap narrows but remains visible. Starting from 20%, the renamed-set curve nearly matches the in-distribution curve. These results provide controlled evidence that low variable-name diversity in the training distribution is a substantial contributing factor to rename brittleness in this setting.

4.2 LOCALISING THE FAILURE

We next examine where errors occur in the generated traces, focusing on failures to generate unseen variable names. We analyze failures across different diversity levels and random seeds. The errors are strongly concentrated at the end of variable names. In particular, at low diversity, most failures after variable renaming occur when the model finishes generating a variable name and is supposed to generate a delimiter marking the end of the variable (e.g., the assignment operator =). Across the three seeds, this accounts for 91.2% of failures at 0.5% diversity and 93.7% at 1% diversity. These correspond to thousands of failed traces rather than a small number of isolated errors.

Thus, the large number of failures after variable renaming at low diversity is predominantly associated with a specific prediction: whether to terminate a variable name or continue generating its characters. We therefore focus the subsequent analysis on this name-ending decision.

4.3 LOCALISING THE COPYING MECHANISM

In our tracing task, generating a trace involves reproducing variable names that already occur in the input program. The delimiter following an earlier occurrence of the same name provides a contextual cue to where that name ends. We therefore examine attention to these delimiters to identify candidate heads involved in predicting variable-name endings. We first measure whether attention heads attend to the boundary of the previous occurrence of the variable name. As training diversity increases, the strongest heads show increasingly strong copying behavior. We identify candidate attention heads using a *copy score*, which measures how much attention a head assigns to this earlier delimiter. For simplicity, we will refer to the training distribution as ID (in-distribution), while we refer to the codes obtained after renaming to variables unseen during training as OOD (out-of-distribution).

Copy scores are low at 0.5% diversity but increase substantially with greater diversity (Figure 3). This effect is also reflected in the difference between ID and OOD names at low diversity, where the same heads retrieve the boundary much less reliably for unseen names.

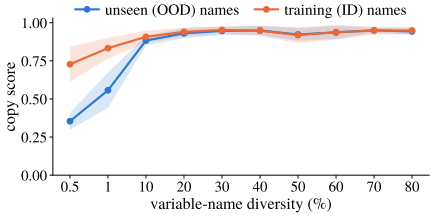

Figure 3: Copy score of the highest-scoring attention head, measured at name-ending positions on renamed (OOD) and original (ID) programs. Points are means over three training seeds; shaded bands show ± 1 standard deviation across seeds.

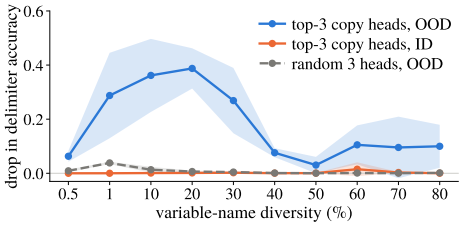

Figure 4: Change in the accuracy of predicting the delimiter that ends a variable name when the three highest-scoring copy heads are set to zero, compared with 50 random sets of three heads. The effect on ID examples is negligible throughout. Points are means over three training seeds; shaded bands show ± 1 standard deviation.

We next test whether these heads are causally involved in the name-ending decision. Ablating the highest-scoring heads reduces accuracy on OOD, while the corresponding effect on ID examples is generally negligible (Figure 4), supporting their role in the failure.

However, the relevant computation is not confined to a small number of heads. We therefore progressively ablate the top 1 through 15 copying heads and measure the resulting OOD accuracy (Figure 5). The effect accumulates as more heads are removed. For example, at 60% diversity, OOD accuracy falls gradually as more heads are ablated. In contrast, randomly selected heads produce little change, with OOD accuracy remaining high across the same range. Thus, the name-ending behavior depends on a distributed set of attention heads that collectively retrieve the boundary information needed to terminate an unfamiliar variable name (unseen during training).

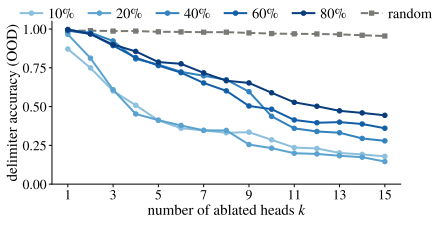

Figure 5: Accuracy of predicting the delimiter that ends a variable name, on renamed programs, after ablating the k highest-scoring copy heads, for five diversity levels. The dashed line shows random sets of the same size k. Curves are means over 3 seeds.

Diversity therefore strengthens the copying of unfamiliar names (Figure 3) and improves accuracy on renamed programs (Figure 2); this link is correlational.

4.4 Redirecting copy heads to earlier delimiters repairs name-ending errors.

Head ablations establish that the identified heads contribute to correct name termination. To test whether redirecting their attention can repair actual renaming failures, we pair each test program with a renamed variant that preserves identifier lengths. We select name-ending positions where the model correctly predicts the delimiter in the original program but fails in the renamed variant. At each failing position, we force the 3 highest-scoring copy heads to attend exclusively to the delimiter following an earlier occurrence of the same renamed identifier, then recompute their outputs using the model's standard computation. This intervention repairs 22% of failures at 0.5% diversity and 37% at 1% diversity, compared with 1% and 2%, respectively, when applied to 3 random heads from the same layers. Thus, redirecting only 3 heads to contextual delimiter information repairs a sizeable fraction of the observed failures, far more than chance.

Redirecting the same heads to the delimiter after a *different* identifier repairs fewer failures (15% and 25%, against 22% and 37%). This suggests that the heads retrieve the boundary of the matching name, not just any delimiter, although part of the repair comes from a generic end-of-name cue.

4.5 Do Our Findings Hold at a Larger Scale? (StarCoder Model)

Having used a small-scale model trained on synthetic data to explore many hypotheses through extensive experimentation, we now test whether the resulting findings in Sections 4.2–4.4 hold in a larger model trained on real code. Starting at small scale served two purposes: it made broad hypothesis testing computationally feasible, since a single training run of the larger model takes around a day on 4×A100 GPUs, and it provided a controlled, simplified setting in which we could isolate potential causes of brittleness to variable renaming.

To test our findings on a larger scale, we use the `tiny_starcoder_py` (Li et al., 2023) model, a modern 158M-parameter decoder-only Transformer based on the GPT-BigCode architecture (Li et al., 2023) and pretrained on 10.5B tokens Python code from the codeparrot-clean dataset (von Werra, 2021) (around 5.3 million Python files sourced from GitHub). We compare two models: (1) a standard baseline trained on the original dataset, and (2) the same model trained on the same dataset except that it has a high diversity of variable names.

We increase identifier diversity in the training dataset by independently renaming each local variable with probability 0.5 to a fresh, corpus-unique string. We leave function names, and built-ins unchanged. These random names provide no semantic cues, encouraging the model to track variables through their context. We retain half of the original names. This mixture preserves informative names typical of real code while encouraging robustness to unfamiliar ones (renaming all variables performed worse in our experiments, Appendix D.5). Aside from the dataset modification, we keep the same configuration for both models; a more detailed description of the training environment is provided in Appendix D.

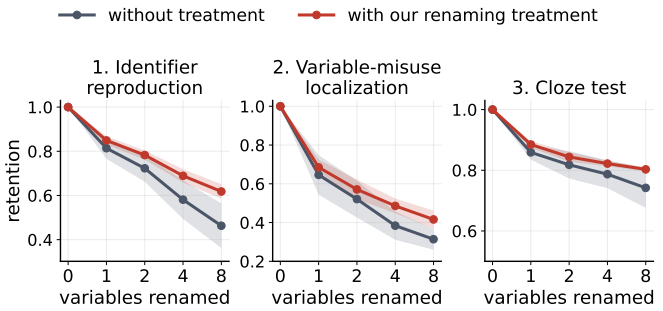

Figure 6: Retention after renaming N variables, comparing the original and increased-diversity training conditions across different tasks. Shaded bands: ± 1 standard deviation over three training seeds.

To test the robustness of the two models to variable renaming, we perform a zero-shot evaluation on 3 tasks. We use a test set composed of the first 2000 Python functions of the CodeSearchNet (Husain et al., 2019) test split (Appendix D.6). The tasks on which we test the two models are: (1) *Identifier reproduction* which tests whether a model is able to complete the code with the correct identifier name given at least one occurrence of that identifier appeared earlier in the function. It answers the question: "Is the model able to write unfamiliar name that it has been shown earlier in the code?"; (2) *Variable-misuse localization*: we construct the test examples by replacing one occurrence of a variable with another variable from the same function, introducing

a bug while preserving valid Python syntax. The model must identify the bug's location. This answers the question: "Can the model still identify an incorrectly used variable after renaming?"; (3) *Cloze test*: this benchmark is the CodeXGLUE's ClozeTest-all benchmark (Lu et al., 2021), where one token in a real function is masked and the model must select the correct token from a fixed set of candidates. We retain only examples whose answer is not a variable name, ensuring that renaming leaves the correct answer unchanged. This answers the question: "Does renaming affect the model's ability to predict other tokens in the code?".

We first record the performance of the model on each of the tasks on the original data (without renaming), then, to evaluate the robustness, we run the same evaluation while renaming N variables in each test sample ($N \in 1, 2, 4, 8$), and for each N value, we compute the fraction of originally correct predictions that remain correct after renaming. We call this fraction "retention". The N variables are drawn at random and replaced by random identifiers of the same length of the original. Appendix D.6 provides the evaluation setup including task scoring. We report retention for both models across all the tasks in Figure 6. Because retention counts only each model's own correct predictions, Appendix D.4 also reports plain accuracy over all test items. Before renaming, the two models are equally accurate (within the variation across seeds); after renaming several variables, the higher-diversity model is more accurate on all three tasks.

Compared with the baseline, the figure shows improved performance with increased diversity across the tasks, with the difference in performance increasing with additional renaming operations; this shows that variable name diversity is a factor that contributes to robustness to variable renaming.

Renaming also removes meaningful names. To check that the gain comes from diversity, we train a control model with the same renaming procedure used in the high-diversity model, except that the new names are drawn from a small set, so diversity stays low. On all three tasks, the control is generally less robust than the high-diversity model, indicating that the improvement on these tasks mostly comes from the high diversity (Appendix D.7).

Localising the failure. Section 4.2 asked where the errors caused by renaming occur. We ask the same question but on the StarCoder model. To do this, we record the first token at which each wrong name goes wrong. Unlike the tracing model, this model produces many errors with the original names, so we count only the names that are correct in the original programs and wrong after renaming. At low diversity, 90% of these names first go wrong at their first token (Appendix E.2). As in the tracing model, the failure is not spread over the name: it is concentrated at a single decision. The tracing model fails to end an unfamiliar variable name; here, it fails to start one. In both cases, it fails at generating an unfamiliar variable name (the name introduced after renaming). We therefore focus our study in the next experiments on the first token of a variable name.

Copy score. Section 4.3 identified attention heads that retrieve a name from its earlier occurrences, using a copy score: the attention a head sends to the delimiter after the earlier occurrence of the same name. We repeat this measurement at the first token: a head's copy score is now the attention it sends from the position that predicts a name's first token to the first tokens of earlier occurrences of the same name. As in Figure 3, higher diversity strengthens the copying of renamed names (OOD): their copy score rises from low diversity to high diversity (Figure 7A).

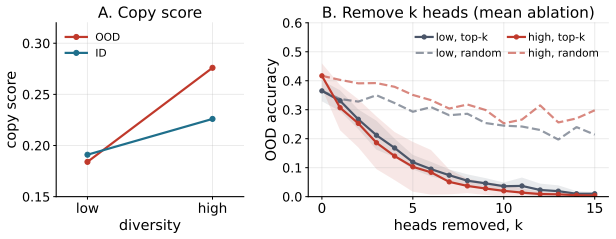

Figure 7: Reproduction of the analyses of Section 4.3 on real Python code (158M models). (A): copy score of the highest-scoring head on original (ID) and renamed (OOD) names (one seed). (B): first-token accuracy on renamed names after removing the k highest-scoring copy heads (solid) or k random heads (dashed); a removed head's output is replaced by its mean output (mean ablation). Low and high stand for the amount of variable name diversity in training. Means over three seeds.

Removing the copying heads. Section 4.3 then tested whether these heads are needed: removing them broke renamed traces, while random heads of the same number had little effect. We repeat both tests at the first token. In this larger model, setting a head's output to zero damages the model even when the heads are selected randomly. Zero ablation can remove average contributions that downstream computations rely on (Wang et al., 2022). We therefore use mean ablation, a standard alternative (Wang et al.,

2022; Li & Janson, 2024): each removed head's output is replaced with its average output over the test functions. Removing the copy heads drives first-token accuracy on renamed names to almost zero after fifteen heads, whereas fifteen random heads leave most of it (Figure 7B), as in Figure 5. We conclude, as in the smaller-scale experiments, that the heads attending to earlier occurrences are needed to reproduce unfamiliar names, and that this ability is spread over many heads.

Redirecting copy heads to earlier identifiers repairs name-generation errors. Section 4.4 showed that redirecting copy-head attention to earlier delimiters repairs name-ending failures in the small tracing model. We apply an analogous intervention to the larger model at positions where renaming causes an error on the first token of an identifier. At each failing position, we restrict the 12 highest-scoring copy heads to attending to the first-token positions of earlier occurrences of the target identifier. This repairs 76% of failures at low diversity and 78% at high diversity, against 3% and 5% for 12 random heads from the same layers (means over three seeds). Redirecting them instead to the other variables of the function repairs fewer than 2% of failures and shifts the prediction toward those variables' first tokens (63% and 70%, against 35% and 38% without intervention; Appendix E.2). Redirecting the copy heads to the earlier occurrences of the target variable therefore repairs most of the first-token errors caused by renaming, and this repair requires the copy heads: random heads repair almost none.

These results support, for the two models we studied, a difficulty in generating unfamiliar names under low variable name diversity. They also support a robustness benefit from greater variable-name diversity in training. The dominant error position differs between the two models.

5 Related Work

Robustness to variable renaming. Liu et al. (2023) showed that the performance of CodeBERT (Feng et al., 2020) and GraphCodeBERT (Guo et al., 2020) decreases with simple perturbations such as variable renaming. They proposed a method based on contrastive learning to mitigate the issue. Other prior work also documents sensitivity to renaming in code models (Rabin et al., 2021; Henkel et al., 2022; Zhang et al., 2020; Yang et al., 2022). Our goal is complementary: we aim to understand the internal computations and mechanisms associated with robustness to variable renaming. We identify where predictions fail during unfamiliar-name generation and test the attention heads involved through ablation and attention redirection.

Using AI interpretability techniques in code models. In code models, mechanistic interpretability techniques help explain how internal components support behaviors such as completing variable names in Python docstrings (Conmy et al., 2023) and retrieving values through chains of variable assignments (Wu et al., 2025). Lazić et al. (2026) study symbolic diversity and unseen-symbol generation using architectural and embedding interventions. The closest prior work to ours asks how a training distribution changes the strategy a transformer learns. For a single-layer transformer on a synthetic copying task, Kawata et al. (2026) prove that sufficient diversity in the spacing between repeated trigger tokens favors contextual copying through an induction head over a shortcut based on token positions. The shortcut fails when the positional pattern changes at test time. Our study varies the diversity of variable names rather than the spacing between repeated tokens. We combine this controlled variation with precise failure localization and attention interventions to repair some incorrect name-generation predictions.

While prior work characterizes renaming sensitivity, analyzes copying mechanisms, or improves unseen-symbol generation, we focus on understanding brittleness to variable renaming and how variable-name diversity relates to the ability of code models to reproduce unfamiliar identifiers. Our contribution combines controlled training experiments with causal tests of the internal computations contributing to renaming failures.

6 Conclusion

We show that difficulty generating unfamiliar variable names is a central source of brittleness to variable renaming in the models studied. Using TinyTracer for controlled experiments and mechanistic interpretability techniques, we find that errors concentrate at name endings in the small-scale low-diversity tracing models. A larger StarCoder-based model trained on real Python code exhibits a similar difficulty at initial name tokens. Ablation and attention-redirection experiments show that heads attending to earlier name occurrences contribute causally to correct generation. Increasing variable-name diversity improves robustness in both settings and is associated with stronger attention to earlier name occurrences and their delimiters.

AI USE STATEMENT

We made light use of large language models (LLMs) to (i) polish grammar and phrasing in parts of the manuscript, (ii) help identify potentially relevant related work during the literature review, and (iii) to perform some coding tasks. In all cases, we always thoroughly check the AI output and never use its output directly without checking.

REPRODUCIBILITY STATEMENT

We strive to make all results fully reproducible. The problem setup and method are specified in the main text, and all implementation choices, training protocols, and evaluation procedures are cross-referenced there and detailed in the supplementary material (hyperparameters, optimization settings, architecture comparison combinations, and exact model configurations). We will release (with the camera-ready paper) the full source code, datasets, and evaluation scripts, including all parameters and hyperparameters used to produce every table and figure, as well as configuration files for each experiment, random seeds, and step-by-step run commands. The repository also includes environment specifications (e.g., requirements), preprocessing utilities for data, and scripts to reproduce metrics and plots.

REFERENCES

Pavol Bielik and Martin Vechev. Adversarial robustness for code, 2020. URL https://arxiv.org/abs/2002.04694.

Arthur Conmy, Augustine Mavor-Parker, Aengus Lynch, Stefan Heimersheim, and Adrià Garriga-Alonso. Towards automated circuit discovery for mechanistic interpretability. *Advances in Neural Information Processing Systems*, 36:16318–16352, 2023.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. In *Findings of the association for computational linguistics: EMNLP 2020*, pp. 1536–1547, 2020.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. GraphCodeBERT: Pre-training Code Representations with Data Flow, 2020. URL https://arxiv.org/abs/2009.08366v4.

Jordan Henkel, Goutham Ramakrishnan, Zi Wang, Aws Albarghouthi, Somesh Jha, and Thomas Reps. Semantic Robustness of Models of Source Code. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 526–537. IEEE, 2022. doi: 10.1109/saner53432.2022.00070. URL http://dx.doi.org/10.1109/saner53432.2022.00070.

Hugging Face. Codeparrot dataset (cleaned). https://huggingface.co/datasets/codeparrot/codeparrot-clean, 2022. Deduplicated and filtered Python source from GitHub; 5,361,373 files.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*, 2019.

Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. Learning and evaluating contextual embedding of source code. In *International conference on machine learning*, pp. 5110–5121. PMLR, 2020.

Andrej Karpathy. karpathy/nanoGPT, December 2022. URL https://github.com/karpathy/nanoGPT.

Ryotaro Kawata, Yujin Song, Alberto Bietti, Naoki Nishikawa, Taiji Suzuki, Samuel Vaiter, and Denny Wu. From shortcut to induction head: How data diversity shapes algorithm selection in transformers. *Advances in Neural Information Processing Systems*, 38:69460–69497, 2026.

Taku Kudo and John Richardson. SentencePiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. In Eduardo Blanco and Wei Lu (eds.), *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pp. 66–71, Brussels, Belgium, November 2018. Association for Computational Linguistics. doi: 10.18653/v1/D18-2012. URL https://aclanthology.org/D18-2012/.

Nevena Lazić, Liam Fowl, András György, and Csaba Szepesvári. To see the unseen: on the generalization ability of transformers in symbolic reasoning. *arXiv preprint arXiv:2604.21632*, 2026.

Maximilian Li and Lucas Janson. Optimal ablation for interpretability. *Advances in Neural Information Processing Systems*, 37:109233–109282, 2024.

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 Randy 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, Nourhan Fahmy, Urvashi Bhattacharyya, W. Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jana Ebert, Tri Dao, Mayank Mishra, Alexander Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean M. Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. Starcoder: may the source be with you! *ArXiv*, abs/2305.06161, 2023. URL https://api.semanticscholar.org/CorpusID:258588247.

Shangqing Liu, Bozhi Wu, Xiaofei Xie, Guozhu Meng, and Yang Liu. Contrabert: Enhancing code pre-trained models via contrastive learning. In *Proceedings of the 45th International Conference on Software Engineering*, ICSE '23, pp. 2476–2487. IEEE Press, 2023. ISBN 9781665457019. doi: 10.1109/ICSE48619.2023.00207. URL https://doi.org/10.1109/ICSE48619.2023.00207.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. Codexglue: A machine learning benchmark dataset for code understanding and generation. *CoRR*, abs/2102.04664, 2021.

Pedro Orvalho and Marta Kwiatkowska. Are large language models robust in understanding code against semantics-preserving mutations?, 2025. URL https://arxiv.org/abs/2505.10443.

Ofir Press, Noah A. Smith, and Mike Lewis. Train short, test long: Attention with linear biases enables input length extrapolation, 2021.

Md Rafiqul Islam Rabin, Nghi D.Q. Bui, Ke Wang, Yijun Yu, Lingxiao Jiang, and Mohammad Amin Alipour. On the generalizability of Neural Program Models with respect to semantic-preserving program transformations. *Information and Software Technology*, 135:106552, 2021. ISSN 0950-5849. doi: 10.1016/j.infsof.2021.106552. URL http://dx.doi.org/10.1016/j.infsof.2021.106552.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI*, 2019. URL https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf. Accessed: 2024-11-15.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer, January 2020. ISSN 1532-4435.

Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. In Katrin Erk and Noah A. Smith (eds.), *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1715–1725, Berlin, Germany, August 2016. Association for Computational Linguistics. doi: 10.18653/v1/P16-1162. URL https://aclanthology.org/P16-1162/.

Jianlin Su, Yu Lu, Shengfeng Pan, Ahmed Murtadha, Bo Wen, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding, 2021.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017.

Leandro von Werra. Training codeparrot from scratch. Hugging Face Blog, 2021. URL https://huggingface.co/blog/codeparrot. Accessed: 2026-09-21.

Changhan Wang, Kyunghyun Cho, and Jiatao Gu. Neural machine translation with byte-level subwords. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pp. 9154–9160, 2020.

Kevin Wang, Alexandre Variengien, Arthur Conmy, Buck Shlegeris, and Jacob Steinhardt. Interpretability in the Wild: a Circuit for Indirect Object Identification in GPT-2 small, 2022. URL https://arxiv.org/abs/2211.00593v1.

Yiwei Wu, Atticus Geiger, and Raphaël Millière. How do transformers learn variable binding in symbolic programs? *arXiv preprint arXiv:2505.20896*, 2025.

Zhou Yang, Jieke Shi, Junda He, and David Lo. Natural attack for pre-trained models of code. In *Proceedings of the 44th International Conference on Software Engineering*, pp. 1482–1493. ACM, 2022. doi: 10.1145/3510003.3510146. URL http://dx.doi.org/10.1145/3510003.3510146.

Huangzhao Zhang, Zhuo Li, Ge Li, Lei Ma, Yang Liu, and Zhi Jin. Generating Adversarial Examples for Holding Robustness of Source Code Processing Models. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(01):1169–1176, 2020. ISSN 2374-3468 ; 2159-5399. doi: 10.1609/aaai.v34i01.5469. URL http://dx.doi.org/10.1609/aaai.v34i01.5469.

A THE TINYTRACER FRAMEWORK

A.1 FRAMEWORK OVERVIEW

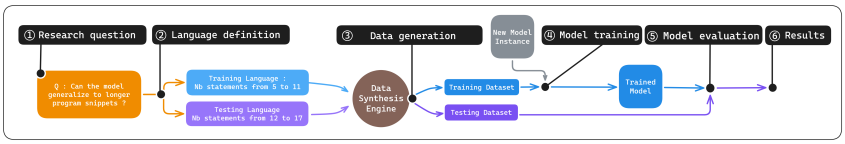

Figure 8: TinyTracer Framework summarized in 6 steps: Research question, Language definition, Data generation, Model training, Model evaluation, and Results

In this section, we describe the general capabilities of TINYTRACER and illustrate supported configurations. Individual experiments use their own training and test distributions, depending on their intended goal.

Our framework, presented in Figure 8, lets the user pose a concrete research question, generate the exact data needed, train a small model, and evaluate it to obtain an answer to the question. For example, to test whether a model trained only on programs with at most 10 statements can generalize to longer programs, we define two program families: one where all programs have ≤ 10 statements and another where all programs have > 10 statements. We synthesize a training set from the first family and a test set from the second. A Transformer-based language model is then trained to perform the task of *TinyTracing* (given a code and the current execution state, predict the next execution state). The model is then evaluated on the longer programs using exact-match over full traces (a prediction is correct only if every step matches). This workflow is reused throughout the paper: each experiment chooses the appropriate data distributions, synthesizes training and test datasets, trains a small tracer to reach high in-distribution accuracy, and evaluates it under the intended shift. To support experiments such as the previous one, the framework offers four building blocks:

1. **Task (TinyTracing; Sec. A.3).** The model is given a code snippet and the *current execution state* (which instruction has just been executed and the current variable values) and must predict the *next* state, step by step. Because each prediction is a concrete state change, performance reflects program behavior (operational semantics) rather than surface form.
2. **A programming language (Sec. A.4).** The framework defines a programming language called TINY-α. It is compact enough so that a small model can reach (near-)perfect accuracy in the TinyTracing task within a few hours of training, yet expressive enough to enable interesting experiments.
3. **Data synthesis engine (Sec. A.5).** A rule-based tool that generates programs and their stepwise traces under user-chosen distributions (e.g., maximum statements, maximum nesting, symbol coverage). It also creates clean train/test splits by removing duplicates.
4. **Lightweight model (Sec. A.6).** A small decoder-only Transformer and an easy-to-run setup that make broad ablations and repeated runs easy.

A.2 DESIGN GOALS

Our design for the TINYTRACER framework targets the following goals:

Goal 1: A cost-effective tiny-LM testbed. Design a programming language on which a small model can be trained to (near-)perfect accuracy on our task in a few hours, enabling rapid iteration. The language is minimal enough for fast training yet rich enough to support non-trivial control flow and state updates. We propose two simple languages that meet these constraints.

Goal 2: Fine-grained control of data distributions. We want to enable users to generate datasets of programs with precise, fine-grained control over the underlying data distribution: users specify the space of possible programs and per-construct probabilities, and the data-synthesis engine samples programs that respect these specifications (Sec. A.5).

Goal 3: Learning code semantics. We want a task on which we train the model, such that learning the task entails learning the program's semantics. We choose the task of code-tracing and encode it such that it captures the operational semantics of code (Sec. A.3).

We do not aim to rank proprietary LLMs or claim universality beyond the present tiny-LM, code-tracing setup.

A.3 TINYTRACING TASK

In the TINYTRACER framework, the model is trained to perform the task of *TinyTracing*. In this task, the model takes as input a program annotated with its current execution state (value of each variable and a pointer to the last instruction that has been executed); It predicts as output the program, annotated with the new execution state. Prediction of the new annotated program is done token by token (where generated tokens are fed back as input to the model so it predicts the next token). A depiction of the TinyTracing task is in Figure 9.

In the TinyTracing task, a program p is paired with a *stepwise execution trace* $T = \langle s_0, \ldots, s_k \rangle$. Each step s_i duplicates the full code and annotates the code with the following pieces of information: (i) the instruction pointer (marked with a special symbol `@` at the line being executed), and (ii) the current variable state (values of the program variables rendered as key–value pairs). A delimiter (e.g., `#STEP`) marks step boundaries. This format makes operational semantics observable and supports exact-match evaluation over the full trace. The model takes the tokens of one step as input, and predicts the tokens of the next step one by one. Once a token from the next step is predicted, it is appended to the input tokens (as is usually done in auto-regressive models). Figure 9 illustrates the task with the pointer and environment annotations called out explicitly.

By adjusting the complexity of the programming language and code snippets used for tracing, the task can be adapted to fit the computational budget of users, allowing cost-effective experiments.

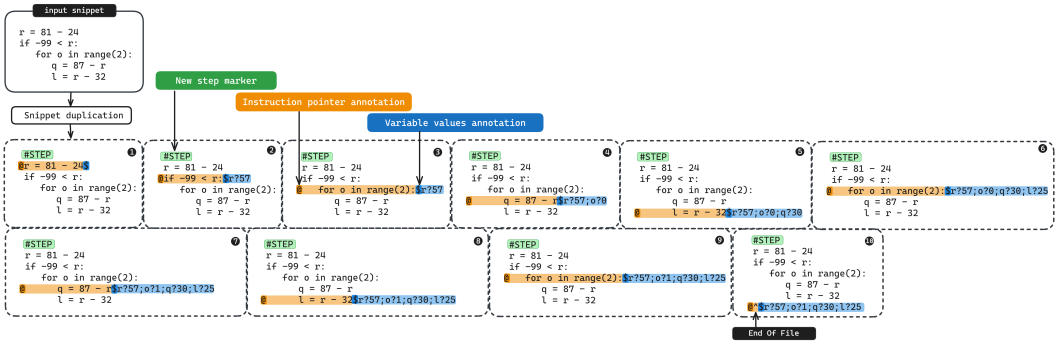

Figure 9: Illustration of the TinyTracing task. At each execution step, the entire input code snippet is duplicated and annotated with corresponding execution information: the instruction pointer value is represented by setting a special symbol (here `@`) to the left of the current line to be executed, and the variable states are presented as (key, value) pairs to the right of the current execution line. A special symbol (here `#STEP`) indicates the beginning of a new step

A.3.1 WHY THE TASK OF TINYTRACING?

We chose the *TinyTracing* task for many reasons. First, the task is *lightweight*: a small model can learn it to (near-)perfection in hours, which makes iteration and ablations feasible. This is in contrast to other tasks, such as predicting the output directly, which might be interesting but require much larger computational power for training and inference. Second, the stepwise format makes *debugging* straightforward. Failures localize to a specific line and variable, so we can analyze where and why the model went wrong. Third, because it makes program semantics observable, checkable, and practical to study. At each step the model must predict the next execution state (instruction pointer and variable values), which directly mirrors *small-step operational semantics*: mastering the task indicates the model has learned how program state evolves, not just surface patterns. Fourth, evaluation is *deterministic*: ground truth comes from the Python interpreter, and our evaluation rule ensures each prediction is a function of the visible state, enabling clear conclusions without noise from nondeterminism. Finally, data generation for this task is easy and automatic, allowing the generation of large amounts of training data.

Early during this project, we considered the task of predicting the final output of code directly instead of the TinyTracing task that we use now. The main challenge that we faced was computational. We developed a model to predict code output, but it was much larger than our current model (above 100M parameters), and required much more data, and training took 10 days compared to our current

model, which trains in a few hours only, and, even with that, the model did not reach high levels of accuracy. Reaching high levels of accuracy requires an even larger model. We then quickly realized that for our framework to be practical, and for us to be able to perform all of our experiments, we needed a much simpler task, that can be learned with a smaller model and trained in a few hours only to reach high levels of accuracy. The TinyTracing task satisfied that.

A.4 THE TINY-α AND TINY-β PROGRAMMING LANGUAGES

A.4.1 LANGUAGE DESIGN

We propose a small language whose programs are cheap to learn and evaluate, so we can run definitive experiments quickly. A user can define other languages depending on their needs. The design of our proposed language follows the following constraints: 1) *Three-address statements.* All statements are in three-address form (one operation and one assignment). This simplifies the task of learning without sacrificing the operational semantics we wish to test. 2) *Cost-effective numerics.* We bound integers to a small, fixed range. Increasing the range mainly inflates data requirements, encouraging memorization rather than adding meaningful semantic complexity, so we keep it narrow to maintain fast training. 3) *Avoiding difficult operators.* We exclude the floating point data type, the multiplication, division, and remainder operations. These tend to require a substantially larger model and more data to be learned effectively; omitting them keeps the task focused on control flow and state updates. 4) *Python-subset design.* We define our languages as *proper subsets of Python*: they reuse Python's concrete syntax and inherit Python's operational semantics for the admitted constructs. This choice lets us execute programs with the standard interpreter and collect stepwise traces using Python's tracing APIs, while keeping the syntax familiar and the tooling (e.g., debuggers, runtimes) immediately usable.

A.4.2 WHY NOT DEFINE A MORE COMPLEX LANGUAGE?

In early iterations of this project we used a richer language that includes multiplication, division, and remainder as well as while loops. Training a competent tracer for that setting required a $\sim$65M-parameter model and several days of training, which slowed experimentation. One of the main goals of our framework is to allow fast model training, enabling users to perform a large set of experiments fast. With the two languages used in this work, a tiny model attains (near-)perfect tracing within a few hours, enabling rapid experiments.

A.4.3 LANGUAGE DEFINITIONS

In this section, we define the two languages TINY-α and TINY-β, with concrete examples shown later.

TINY-α Given the limited capacity of our small Transformer models (and limited computational budget), we focus on a minimalist subset of the Python language, ensuring the TinyTracing task remains learnable. This subset is built around the following programming constructs: variable assignments, arithmetic operations, and conditionals. We enforce the following constraints to define this Python subset: **1) Syntactic Constraints:** The grammar and vocabulary of the generated code are intentionally limited. The grammar includes variable assignments, `if` conditional, and arithmetic expressions. Variable identifiers are constructed by concatenating up to five characters sampled from a set of 26 lowercase English letters, the generation is designed for a uniform length distribution so that for every variable length $L \in \{1, \ldots, 5\}$ occurs with a 20% probability, additionally, the dataset captures 1% of all possible combinations for variable names of a given length $L > 1$, while all single character names are included in the dataset, we refer to this coverage percentage as *variable-name diversity* (formally defined in Sec. 4.1). Snippets exclusively contain one arithmetic operation type, which is limited to either addition or subtraction ($+, -$), while conditional comparisons are limited to less-than and greater-than operators ($<, >$). The only supported data type is integers, and integer values are limited to be within the range [-999, 999]; **2) Structural Constraints:** The overall structure of the code snippets is also bounded. The total number of statements in any generated snippet is constrained to be between 5 and 10. Furthermore, the maximum nesting depth for control flow blocks (i.e., `if` statements) is limited to two. Additionally, the maximum number of control flow blocks allowed per indentation level is 2; **3) Runtime Constraints:** Any generated snippet that results in a variable holding an integer value outside the [-999, 999] range at any point during its execution is discarded. This filter ensures that the model only needs to learn representations for a bounded and manageable range of integer values.

TINY-β This is an expanded version of the previous language, incorporating all prior constraints, with the following modifications: **1) Syntactic Constraints:** The grammar now includes `for` loops also, where the loop iterator is sampled from the same fixed set of variable identifiers. Once an identifier is assigned as a loop index, it is not reused elsewhere in the snippet to guarantee termination. **2) Structural Constraints:** The previously defined nesting depth and the maximum number of control flow blocks per indentation level now apply jointly to both `if` statements and `for` loops.

A.5 DATA SYNTHESIS ENGINE

To complete our experimentation setup, we require the ability to access arbitrary quantities of data with precisely controlled distributional characteristics. For this purpose, we introduce **TinyTrace-Generator**, a data generation tool that enables the synthesis of random code snippets that are written in a subset of Python (or in Python) and that conform to a certain data distribution (defined using a set of constraints on the Python language). The tool also allows the creation of the execution trace of these snippets according to the format of the TinyTracing task. TinyTrace-Generator is structured as a three-stage pipeline, each stage is described in greater detail in the following paragraphs.

A.5.1 CODE SNIPPET GENERATION (SKELETON CONSTRUCTION, CODE INSTANTIATION)

The first stage of our data synthesis engine generates code snippets using an imperative, rules-based generator. It runs two coupled processes: 1) skeleton construction, which builds an abstract snippet with high-level keywords (each keyword represents a language construct), and 2) code instantiation, which turns those keywords into concrete Python code. The skeleton construction process, generates an abstract structure or "skeleton" of the code using a high-level intermediate language composed of specific `user-defined keywords` (e.g. `[ASSIGNMENT]`, `[IF_STATEMENT]`, `[FOR_LOOP]`). The code instantiation process consumes these keywords as they are generated, translating each into a corresponding fragment of concrete Python code. Both processes are controlled by a set of `user-defined generation rules`: `high-level` generation rules govern skeleton construction, while `low-level` generation rules control code instantiation. It is precisely this parameterization via user-defined generation rules that enables the generator to produce code snippets with the desired distributional characteristics for a given experiment. Further details appear in Appendix B of the supplementary material.

A.5.2 GENERATION OF EXECUTION TRACES

The second step of the pipeline involves generating the execution trace of code snippets created in the first stage. It uses the debugging capabilities provided by the Python standard library to generate tracing. This setup not only enables straightforward access to key runtime information, such as the program counter and the variables' state, but also facilitates the implementation of runtime filtering barriers. These barriers serve to enforce constraints on runtime properties that cannot be determined during snippet generation. Depending on the specific requirements of the target experiment, such filters may include the detection of undesirable variable values during execution (e.g., excessively large integers) or the occurrence of runtime exceptions.

A.5.3 DATA DEDUPLICATION

In addition to the previous steps, we also perform data deduplication to make sure that each generated code snippet is unique. This is important so that the testing data is different from the data used for training.

To avoid duplicates, we enforce uniqueness during data generation: once a code snippet is produced, it is hashed to produce a unique ID; if that ID has already appeared, the snippet is filtered out and not added to the dataset. For hashing, we use SHA-256 over the raw snippets. Deduplication is enforced on the training, validation, and test sets. To prevent leakage, we enforce *global* deduplication across train/validation/test. Concretely, we maintain a single global hash set while building the corpus: any example whose hash is already present is discarded *before* splitting. This procedure ensures both within-split uniqueness and cross-split disjointness.

A.6 MODEL ARCHITECTURE

For our experiments, we employ a custom, decoder-only Transformer architecture, trained from scratch. We use NanoGPT: a small-scale, open-source implementation inspired by GPT-2, devel-

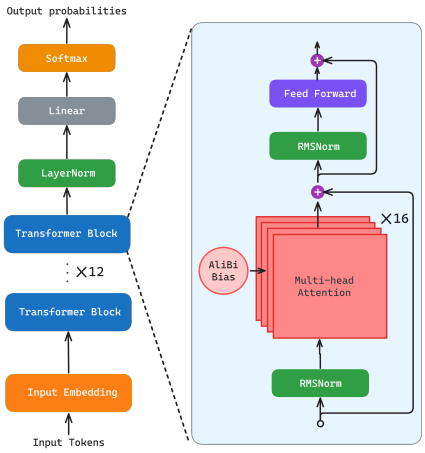

Figure 10: Model architecture

oped by Karpathy (2022). The model is designed to be compact enough for rapid experimentation while being sufficiently powerful to learn the TinyTracing task. Figure 10 shows the architecture of the model that we use in our framework. More details on both the hyperparameters and tokenization can be found in the supplementary material (resp. Sec. A.9, and Sec. A.7).

A.7 Tokenization

We employ a character-based tokenization strategy, as the vocabulary of TinyTrace programs is largely known and fixed. This ensures that each syntactic element is treated atomically and prevents dynamic vocabulary growth. For example, the line `if a < 20:` would be tokenized as `"if", "a", "<", "2", "0", ":"`. Two separate tokenizers are used for our two main languages .

Tiny-α tokenizer. Handles all alphabetic characters (`a–z`), numeric literals (`0–9`), arithmetic operators (=, +, −), the colon (`:`), control keywords (`if`), and structural markers such as newlines, indentation, and tracing delimiters (e.g., `#STEP`). Tokenization proceeds via deterministic regular-expression matching, ensuring consistent encoding. The resulting vocabulary contains 52 atomic symbols.

Tiny-β tokenizer. Extends the Tiny-α vocabulary to support additional control constructs used in loops, such as `for`, `in`, `range`, and parentheses `( )`. This results in a slightly larger vocabulary of 57 atomic symbols. Tokenization remains deterministic and character-based, preserving all syntactic units as indivisible tokens.

All tokenized programs are serialized as sequences of integer identifiers and stored in binary format for efficient streaming during model training. This design ensures reproducibility and maintains the syntactic granularity needed for accurate sequence modeling.

A.8 Training Details

Instead of training the model with multiple epochs, we rather use a sufficient amount of data (determined empirically) and train for a single epoch. Training for multiple epochs in this case does not improve performance. This is useful as it simplifies our experimental setup (more details in Sec. A.9.1).

Evaluation is performed using greedy decoding under an exact-match metric that includes all structural markers. Unless otherwise specified, the reported results correspond to the models trained for a single full pass over each dataset.

Optimization employs the AdamW optimizer with β_1=0.9, β_2=0.95, and weight decay 10^{-1}. A cosine learning rate schedule is applied, peaking at 10^{-3} with a 10% warmup ratio and decaying to

10^{-4} by the final step. Training uses a batch size of 512 sequences, `torch.compile` acceleration, and `bfloat16` precision. Gradients are clipped at 1.0 to prevent instability.

A.9 Model Hyperparameters

The model follows a compact GPT-style architecture, as shown in Figure 10, it contains 12 Transformer blocks, each containing 16 attention heads and an embedding dimension of 384, totaling approximately 21 million trainable parameters. The context window is fixed at 512 tokens. Each block applies pre-normalization using RMSNorm before the self-attention and feed-forward sublayers. The feed-forward network uses a $4\times$ expansion factor with a `SiLU` activation. Attention with Linear Biases (ALiBi) is employed for encoding relative positional information. Dropout is disabled across all layers to preserve deterministic training behavior. Throughout the paper, these are the model's hyperparameters unless specified otherwise.

Table 1 shows the hyperparameters we used when training our model.

Parameter	**Value**
Model Type	Decoder-only Transformer
Number of Layers	12
Number of Heads	16
Embedding Dimension	384
Context Window	512 tokens
Total Parameters	$\sim$21 Million
Normalization	RMSNorm
FFN Activation	SiLU
Positional Encoding	Attention with Linear Biases (ALiBi)

Table 1: Key hyperparameters of the probed Transformer model.

A.9.1 Why do we train for a single epoch only?

One of our goals in this project is to determine how much data is sufficient for the model to reach near-perfect accuracy. In earlier experiments, we approached this by generating a dataset of size N, training for several epochs, evaluating accuracy, then increasing N and repeating the process. However, this protocol adds complexity without any benefit.

Training for a single epoch, on a larger dataset, provides a simpler and cleaner alternative. We can generate a single large dataset, train once over it, and evaluate the model at regular intervals during training. In this setup, each evaluation point corresponds directly to the model having seen a specific number of distinct training examples exactly once. This lets us identify the smallest amount of data needed for the model to reach near-perfect accuracy, with a single training run, simplifying our training process.

Note that our model reaches near-perfect accuracy with a single epoch (since it sees enough data), and therefore training for more epochs is not useful in this case.

B Python Snippets Generation

This section describes the full TinyTracing code generator, including constructs such as `for`-loops and basic data structures.

B.1 Preliminary Concepts: Skeleton Construction, Code Instantiation, and the Context Stack

The snippet generator is structured around two distinct yet complementary processes: skeleton construction and code instantiation. These processes do not operate sequentially; rather, they function in tandem to produce randomly generated Python code snippets that adhere to user-specified distributional properties.

To provide context, we begin with a high-level overview of the respective roles of these two processes in snippet generation. The skeleton construction process, as the name implies, generates an abstract structure or "skeleton" of the code using a high-level intermediate language composed of specific **user-defined keywords**. In parallel, the code instantiation process consumes these keywords as they are produced, translating each into a corresponding fragment of concrete Python code. Both processes are governed by a set of **user-defined generation rules**: **high-level** generation rules control skeleton construction, while **low-level** generation rules govern code instantiation. It is precisely this parameterization via user-defined generation rules that enables the generator to produce code snippets with the desired distributional characteristics for an experiment.

To further elaborate on the internal organization of the generation system underlying the snippet generator, we now introduce a simple illustrative example. As previously noted, the set of keywords used in skeleton construction—referred to as the keyword vocabulary—is defined by the user. In the example considered here, the keyword vocabulary consists of three distinct keywords, which we describe in detail in Table 2.

As shown in Table 2, the three keywords used in our illustrative example are well defined. However, to further develop the description of the generation system of the snippet generator, we need to distinguish a special type of keywords from the others: the **context creation keywords**. These are keywords that lead to the creation of a new **execution context** in a code snippet, which is represented by **indentation** in the case of Python (as an illustrative comparison, it is often the case in other languages, such as C or Java, that new execution contexts are represented with curly braces {}). Based on this definition, two such keywords exist in our example from Table 2: The **[IF_STATEMENT]** and the **[FOR_LOOP]**.

Skeleton Keyword	Description example	Instantiation Example
[ASSIGNMENT]	Corresponds to the initialization of a random variable name with a random constant. The variable name can be one of "a", "b", or "c". The constant can be between 0 and 9.	a = 2
[IF_STATEMENT]	Corresponds to the conditional header of an "if" block. The condition is of the form $n < m$ where n and m are random numbers between 0 and 9.	if $1 < 3$:
[FOR_LOOP]	Corresponds to the looping header of a "for" block. The expression is formatted as "for VAR in range(CTE):" where "VAR" is the index variable of the loop, can be "a", "b" or "c", and "CTE" being the iteration count constant, selected uniformly from this set {1,3}	for b in range (2):

Table 2: Illustrative example of a user-defined keyword vocabulary. Each keyword is given a description along with an instantiation example.

The motivation for distinguishing context creation keywords lies in the fact that many valuable experimental studies can be conducted by manipulating distributional features specifically related to execution contexts creation. These features may include, for instance, the maximum number of nested execution contexts, the permitted nesting combinations, or the maximum number of execution contexts allowed at a given depth. To enable controlled manipulation of such features—while maintaining the flexibility and generality of the data generator—we introduced a specialized data structure that plays a central role in the snippet generator: the **context stack**.

As its name implies, the context stack is a stack-based data structure that plays a pivotal role in the snippet synthesis process. It not only facilitates the controlled creation of execution contexts but also serves as a coordination mechanism between the skeleton construction and code instantiation processes. Broadly speaking, a new level is pushed onto the stack whenever a new execution context is introduced during generation, and correspondingly, levels are popped from the top of the stack as these contexts are closed. Each level of the context stack is itself a composite data structure,

consisting of two distinct sub-components which are associated with the execution context that the level corresponds to:

- **A General Information Dictionary (GID).** This component is an extensible dictionary of user-defined **(key, value)** pairs designed to store relevant metadata about the associated execution context. Such metadata may include, for example, the type of execution context (e.g., loop, conditional), or metrics such as the number of code lines contained within the context. The GID allows for flexible customization and tracking of context-specific properties during snippet generation. These properties can then be used to further control the generation.
- **A Context Keyword Queue (CKQ).** This component is a simple queue data structure responsible for storing the sequence of keywords that will constitute the sub-skeleton of the corresponding execution context. Typically, the CKQ is populated by the skeleton construction process and later consumed by the code instantiation process.

Figure 11 illustrates the concepts outlined above on the structure and function of the context stack.

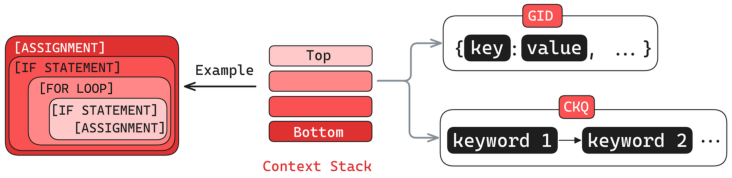

Figure 11: Diagram illustrating a context stack with four nested levels (center), including a detailed view of the substructures within a single stack level (right), and an example of a corresponding skeleton that could give rise to such a stack configuration (left).

B.2 The TinyTrace-Generator Generator Interface

Having established the foundational concepts of skeleton construction, code instantiation, and the context stack, we can now present a comprehensive description of the underlying generation system of the snippet generator.

The skeleton construction and code instantiation processes are each implemented as separate algorithms: the **Skeleton Construction Algorithm (SCA)** and the **Code Instantiation Algorithm (CIA)**, respectively. Within these algorithms, users are to define the generation rules that encode the data constraints required for their specific experiments: high-level generation rules are expressed within the skeleton construction algorithm, while low-level generation rules are defined in the code instantiation algorithm.

However, to ensure compatibility with the overall generation system, including integration with the context stack, these generation rules must conform to a predefined algorithmic template. This template, which we refer to as the **TinyTrace-Generator Interface (TGI)**, is illustrated in Figure 12.

As shown in Figure 12, the TinyTrace-Generator Interface consists of three core algorithms that collectively define the generation system of the snippet generator. These three algorithms include:

1. The skeleton construction and code instantiation algorithms that we described previously.
2. A central **Main Algorithm (MA)** which is in charge of coordinating them.

By appropriately modifying the sections of the TGI designated for user-defined generation rules within the SCA and the CIA—highlighted with a **star (*)** symbol in Figure 12—users can implement a wide range of data distributions tailored to the requirements of their experiments. To demonstrate how this flexibility is achieved, the following paragraphs will describe the structure and functionality of each algorithm comprised in the interface, starting from the topmost component in Figure 12, which represents the main algorithm, before moving to the skeleton construction and code instantiation algorithms.

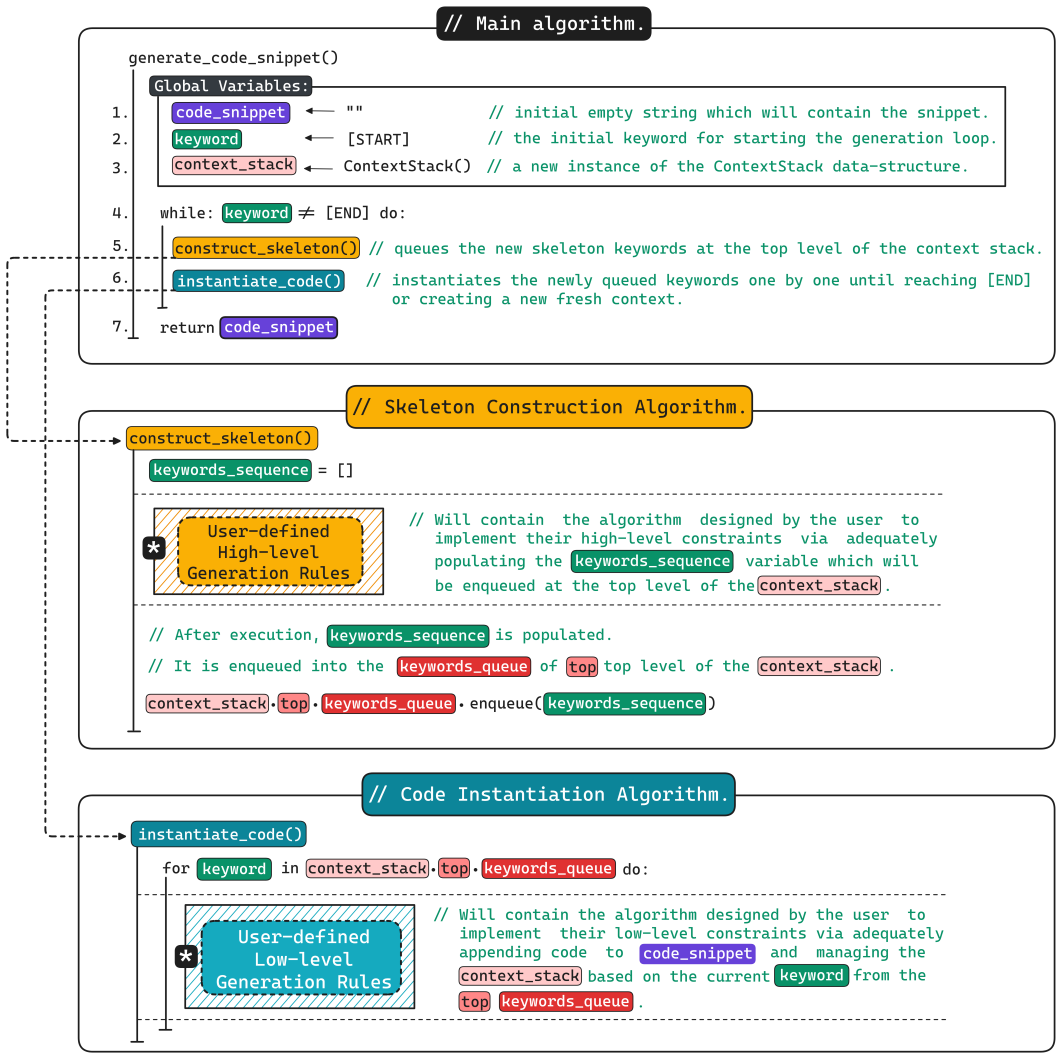

Figure 12: The TinyTrace-Generator Interface (TGI) of the snippet generator.

Algorithm 1. **The Main Algorithm.** This algorithm is implemented by the **generate_code_snippet()** function. It begins by initializing three global variables that are central to the functioning of the generation system: **code_snippet**, **keyword**, and **context_stack**. Understanding the role of each of these variables provides insight into the overall logic of the main algorithm and offers a preliminary view of the two other auxiliary algorithms, which it coordinates:

- **code_snippet**: A string variable that accumulates the final generated code. It is initialized as an empty string, which is to be progressively extended with new code fragments as they are generated by instantiation.
- **keyword**: This variable holds the current skeleton keyword to be instantiated into a Python code fragment. It is initialized with the special keyword **[START]** to enter the main generation loop (line 4) and is to be eventually set to **[END]** to signal the termination of snippet generation.
- **context_stack**: This variable holds the context stack, which we described earlier and depicted in Figure 11. It is initialized with a single level representing the outermost execution context, which is also called indentation level 0.

Following these initializations, the main generation loop (line 4) is entered. The loop's structure is deliberately simple: at each iteration, it invokes the **construct_skeleton()** procedure (line 5),

followed by the **instantiate_code()** procedure (line 6). These two procedures correspond to the **Skeleton Construction** and **Code Instantiation** algorithms, respectively, and will be described in detail in the following dedicated paragraphs.

Broadly speaking, **construct_skeleton()** is responsible for generating new skeleton keywords—according to user-defined rules—and queuing them in the topmost level of the context stack (i.e., the **current context**). These keywords are then sequentially dequeued and processed by **instantiate_code()**, which translates each into a corresponding Python code fragment. This translation is also governed by user-defined rules and may involve additional actions necessary to maintain consistency with the overall generation system. Once the generation loop is exited, the final **code_snippet** is returned (line 7).

Algorithm 2. **The Skeleton Construction Algorithm.** This algorithm is implemented by the **construct_skeleton()** procedure. The algorithm begins by initializing a local variable named **keywords_sequence**. This variable, which begins as an empty list, is to be filled with the sequence of new keywords that should be enqueued in the current context (i.e., the top of the stack). Right after this initialization is the user-defined area for the high-level generation rules, which is the portion of the interface that the user must edit in order to implement the desired high-level distributional constraints. Basically, in order to integrate consistently with the rest of the generation system modeled by the TGI, this user-defined region must populate the **keywords_sequence** variable with the appropriate keywords to be enqueued in the current context, at the end of the **construct_skeleton()** procedure, as shown by the interface.

Algorithm 3. **The Code Instantiation Algorithm.** This algorithm is implemented by the **instantiate_code()** procedure. The algorithm begins by entering a local main loop. This main loop will dequeue the keywords of the current context, one by one, into the global **keyword** variable. Inside the loop is the user-defined area for the low-level generation rules, which is the portion of the interface that the user must edit in order to implement the desired low-level distributional constraints. Basically, in order to integrate consistently with the rest of the generation system modeled by the TGI, this user-defined region must be implemented so that for each keyword, the corresponding code fragment is appended to the global **code_snippet** variable, and, **in case of a context creation keyword**, a new context must be pushed onto the context stack.

— Figure 13 provides an example of a TGI-consistent user implementation for the high-level and low-level generation rules, expressed in pseudo-code. These user-defined generation rules allow for generating Python snippets with the following distributional constraints:

1. **Example of High-Level Constraints:**
    - The snippets are always structured as an if block followed by an initialization statement.
    - The interior of the if block can either be structured as a for block followed by an initialization, with a probability of 30%, or two consecutive initializations, with a probability of 70%.
    - The interior of the for loop contains at least a single initialization.
2. **Example of Low-Level Constraints:**
    - These are represented by the keyword translations described in Table 2.

As illustrated in the example of Figure 13, the user-defined generation rules take the form of an imperative description: that is, they are expressed through procedural algorithms which specify the desired structure of the code snippets in a manner that integrates coherently with the rest of the generation system imposed by the TinyTrace-Generator Interface.

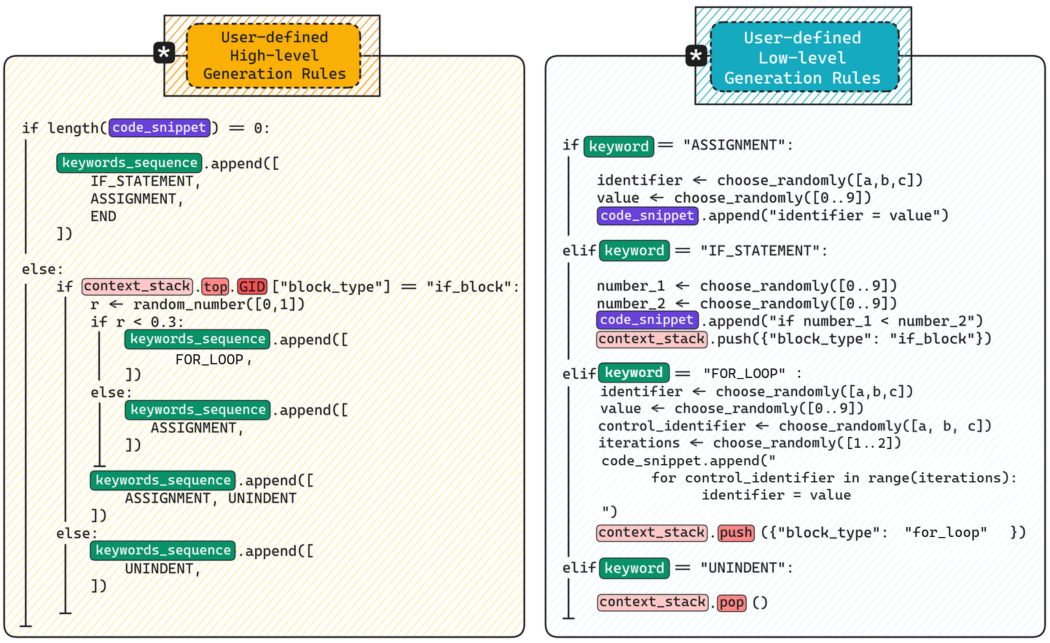

Figure 13: Example of a TGI-consistent implementation of the user-defined high-level and low-level generation rules.

C Does Training on More Data, or with Alternative Tokenizers, or Positional Encodings Help in Robustness to Variable Renaming?

C.1 Hypothesis testing

We have shown that variable diversity is one factor that improves model robustness when exposed to unseen variable names. We have tested other candidates to determine whether they are deciding factors as well. The rest of this section explores two additional hypotheses. We show that scaling the size of the training data does not lead to more robustness to variable renaming; we also show that changing the positional encoding and tokenizer does not solve the robustness issue (bound to the set of components we tested).

C.2 Does Scaling Training Data Improve robustness to unseen variable names?

In this section, we examine the effect of increasing the amount of training data on generalization to unseen variable names. In other words, would the model better generalize to unseen variable names if we train it with more data of the same distribution (in-distribution)? To assess this, we generate a new test set and evaluate our trained small-scale model on this test set. The goal of this evaluation is to evaluate whether our trained model generalizes to unseen variables. We then train another model, but with much more ID data (300M data points) and analyze how increasing the amount of ID data impacts the accuracy of the model on the testset.

C.2.1 Test set of Unseen Variable Names

As described in Sec. 3, the main dataset (TINY-α) is generated according to a language definition that determines its distribution. Consequently, distribution shifts can be introduced by modifying at least one of the rules that govern TINY-α. Using this principle, we define a new test set, obtained by altering a single rule: Use variable names that did not appear in the training dataset.

C.2.2 Evaluation of the Base Model

We evaluate the base model, trained on one epoch of the TINY-α dataset (Sec. 3), on the test set of unseen variable names, in addition to an in-distribution (ID) test set for reference. The model achieved an ID accuracy of 99.71% but failed to generalize to the test set of *unseen variable names*, achieving an accuracy of 4.59% only.

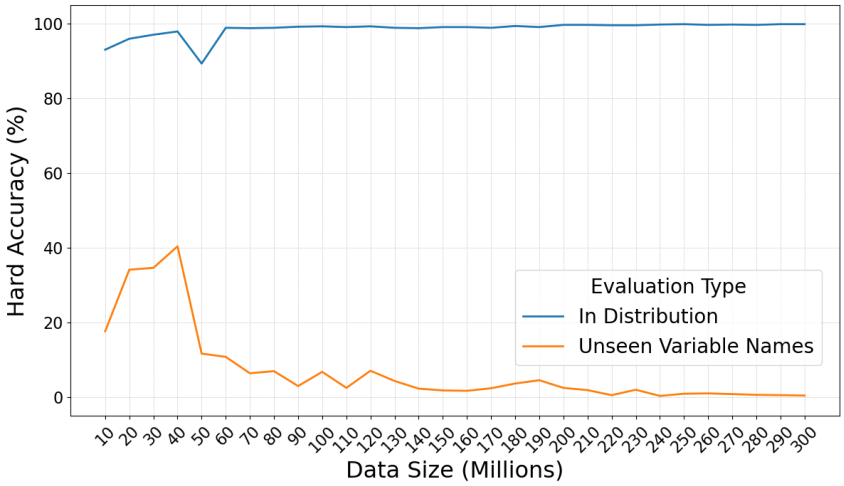

Figure 14: Accuracy evolution of a single model trained on 300 million data examples, with checkpoints saved every 10 million examples

C.2.3 Scaling Out Data

Having established the initial performance of the model trained on TINY-α, we conduct the following experiment. Using the same model architecture, training data distribution, and training environment, the model is trained on a new similar dataset (same distribution of TINY-α) but has 300 million snippets, with a checkpoint saved each time the training reaches a 10 million example milestone. This procedure yields 30 checkpoints, each of which is evaluated on the test set of unseen variable names to observe the evolution of accuracy as more ID data is added.

An interesting phenomenon occurs for the *unseen variable names* variant, where accuracy initially increases till it peaks at 40% (when exposed to 40 million snippets) but subsequently declines with further data, eventually falling to near 0%. This reverse effect can be explained by the overfitting of the model on variable names. The model initially started forming an understanding of variables as value placeholders; however, continued exposure to the same variable names over 300 million snippets pushes the model to memorize variable names, leading the model to link the meaning of a variable to the set of names it saw during training, hence failing to generalize to unseen variable names.

C.3 Studying the Effect of Architecture Variants on Generalization to Unseen Variable Names

In this section, we study the effect of varying the positional encodings and tokenizers on unseen variable names.

We note that the above experiment was done in an early phase of the project using an earlier version of the data-generation setup where the data distribution of the training set was slightly different. While the results in this section cannot be compared with the results in the other sections of the paper (given that the data distributions are slightly different), the comparison in this section remains valid in indicating that different tokenizers and position encodings do not solve the problem of robustness to unseen variable names.

Positional Encoding We compare four positional encoding schemes: (i) *learned Absolute Positional Embeddings (APE) Vaswani et al. (2017)*; (ii) *Rotary Positional Embedding (RoPE) Su et al. (2021)*; (iii) *Relative Positional Bias (RPB) Raffel et al. (2020)*; and (iv)*Attention with Linear Biases*

Variant		ID	UV
Positional Encoding	ALiBi	99.71%	4.59%
	APE	99.32%	2.05%
	RoPE	99.61%	3.32%
	RPB	99.71%	2.44%
Tokenizer	CL	99.71%	4.59%
	BBPE	99.71%	1.27%
	SP	99.80%	5.57%

Table 3: Architecture comparison showing different accuracy results for different positional encoding/tokenizer combinations; models were trained on TINY-α dataset, evaluated on the "in-distribution" dataset (ID) and "Unseen Variable-names" dataset (UV).

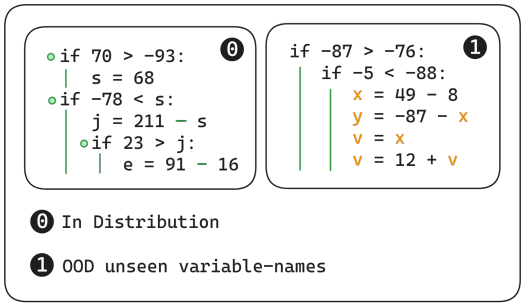

Figure 15: One ID example and one unseen variable name example (TINY-α)

(ALiBi) Press et al. (2021). To maintain model comparability with the ALiBi baseline, APE, RoPE and RPB models were implemented at the same scale by adjusting only the embedding dimension while keeping layers, heads, and context window fixed. Among all of these positional encodings, ALiBi delivered the best accuracy on average, but overall, changing the positional encoding does not show remarkable changes to the unseen variable names scores. The detailed results are presented Table 3.

Tokenizers By fixing ALiBi in our architecture, we compare three tokenization schemes: (i) a *character-level (CL)* tokenizer (our base configuration); (ii) a *SentencePiece Unigram (SP)* tokenizer Kudo & Richardson (2018); and (iii) a *Byte-Level Byte-Pair Encoding (BBPE)* tokenizer Sennrich et al. (2016); Radford et al. (2019); Wang et al. (2020), both with a fixed vocabulary size of 512. The SentencePiece Unigram tokenizer yielded the best accuracy across both datasets, with the Character level tokenizer following up, but, similarly to positional encodings, switching tokenizers does not seem to yield substantial improvements in unseen variable names, but rather keeps similar results, if not worse. Detailed results are provided in Table 3.

D MORE DETAILS ON THE STARCODER MODEL EXPERIMENT

D.1 MODEL ARCHITECTURE

We use the architecture and tokenizer of `tiny_starcoder_py` (Li et al., 2023). The model is a GPT-BigCode decoder. We train at a context length of 1024 tokens, which reduces the parameter count to 158.6 million (compared to the original 164.1 million parameters of the `tiny_starcoder_py` model, that has a context length of 8192), the other dimensions of the architecture are unchanged.

D.2 TRAINING

Each model is trained for 40000 optimizer steps, with a global batch of 262144 tokens per step, giving 10.49 billion tokens in total.

D.3 DATA

We used CodeParrot-clean (Hugging Face, 2022), which contains 5.36 million python files sourced from Github.

D.4 ABSOLUTE ACCURACY OF THE STARCODER MODELS

Figure 6 reports retention: the fraction of a model's originally correct predictions that remain correct after renaming. Since each model is evaluated on its own originally correct predictions, retention scores may be computed over different sets of test items. Table 4 therefore complements this comparison with absolute accuracy for the same models. For each task, it reports accuracy on the original functions and after renaming N variables, using all test items rather than only those originally predicted correctly.

Three observations emerge. First, on the original code, the higher-diversity model is slightly less accurate on identifier reproduction and variable-misuse localization. The difference is smaller than the variation across training seeds. On the cloze test, the two models are equally accurate. Second, after renaming one variable, the models are tied on identifier reproduction and variable-misuse localization. Third, when two or more variables are renamed, the higher-diversity model is more accurate on all three tasks, with the gap increasing with N. Thus, the robustness gain in Figure 6 persists when both models are evaluated on the same test items and cannot be explained solely by scoring each model on its own originally correct predictions.

			accuracy after renaming N variables				retained at N=8
task	model	original	N=1	N=2	N=4	N=8	(3 seeds)
Identifier reproduction	baseline	.505	.468	.433	.364	.298	20164 / 42828
	diversity	.477	.461	.447	.413	.382	25102 / 40456
Variable-misuse loc.	baseline	.581	.466	.389	.295	.241	851 / 2664
	diversity	.550	.469	.409	.356	.311	1059 / 2521
Cloze test	baseline	.318	.310	.308	.301	.293	1144 / 1540
	diversity	.322	.318	.312	.310	.317	1250 / 1555

Table 4: Absolute accuracy of the baseline and higher-diversity models (158M parameters), averaged over three training seeds. "Original" is the accuracy on unmodified functions. The next four columns give the accuracy after renaming N variables, over all test items. The last column shows how many answers were correct before renaming (second number) and how many of those were still correct after renaming eight variables (first number), added over the three seeds (e.g., 20164 / 42828 means that 42828 of the baseline's identifier-reproduction answers were correct before renaming and 20164 of them were still correct after). Standard deviations across seeds are at most 0.09. Test functions and renamings are identical for all models and seeds.

D.5 RENAMING EVERY VARIABLE

In a pilot experiment with a smaller decoder model that uses the same tiny_starcoder_py architecture and data but has only 18.9M parameters (three seeds per setting), renaming every variable instead of half lowered accuracy on the original code: next-token accuracy dropped from 0.554 to 0.518 (0.557 without renaming), and variable-misuse accuracy from 0.791 to 0.708 (0.822 without renaming). We believe that this is due to the fact that the model might benefit from cues given by certain variable names, and training the model with full renaming teaches the model to avoid relying on such cues.

D.6 EVALUATION DETAILS FOR THE STARCODER MODELS

Test functions. We read the Python test split of CodeSearchNet in its original order and keep the first 2000 functions that pass fixed filters. A kept function parses, is a single top-level function of

120 to 2000 characters, has at least three local variables that can be renamed safely, has no such variable longer than 20 characters, contains at least one integer constant, and uses no `global` or `nonlocal` declaration. We remove comments and docstrings. The checks on renaming described below drop two functions, leaving 1998. The filters depend only on the source code, so every model is evaluated on the same functions.

Renaming. A variable is eligible for renaming if it is a parameter or a local variable of the function. Imported names, global names, function and class names, attributes, keyword-argument names, built-ins, `self` and `cls` are never renamed. To rename N variables, we draw N eligible variables at random, with a seed fixed per function and per N. Each receives a new name made of random lowercase letters, of the same length as the original name, that does not already occur in the function. If a function has fewer than N eligible variables, all of them are renamed, so N is an upper bound on the number of renamed variables. We verify every renaming by applying the inverse mapping: if this does not reproduce the original function exactly, the function is dropped. The renamed functions are identical for every model and seed.

Context length. The models are trained with a context of 1024 tokens. All three tasks read the first 320 tokens of each function. How each task handles longer functions is stated below.

Identifier reproduction. The model reads the function left to right, always with the correct preceding text as input (teacher forcing). We score every occurrence of an eligible variable whose name has already appeared earlier in the function. An occurrence is correct when the model's most likely next token matches every token of the name. Occurrences of renamed and non-renamed variables are both scored. Occurrences whose token boundaries do not match the name's boundaries, and occurrences after the first 320 tokens, are not scored.

Variable-misuse localization. The task follows CuBERT (Kanade et al., 2020). From each function we build one item: we pick at random one use of an eligible variable that was assigned earlier and has at least one other variable in scope, and replace it with another variable in scope. The model sees only this buggy function and must find the modified position. Because a left-to-right model cannot point at a position, we let it compare possible fixes. At every candidate position (every use of an eligible variable that was assigned earlier and has at least two variables in scope), we substitute each other variable in scope. We then compute the log-likelihood of the whole edited function, divided by its number of tokens. The edit with the highest score is the model's answer, and localization is correct when this edit is at the modified position. Ties count as wrong. Renaming is applied consistently to the bug, the correct answer and every candidate. An item is dropped if this correspondence cannot be verified exactly for every N, if it has more than 150 candidate edits, or if any renamed version does not fit within 320 tokens with 24 tokens to spare, so that no candidate edit falls outside the model's window. At most 1600 items are kept.

Cloze test. We sample 3000 items evenly from CodeXGLUE ClozeTest-all (Python). The benchmark does not release its answers, so we recover each answer by matching the item to its source function in CodeSearchNet. We keep only items whose answer is not a variable, so renaming never changes the correct answer. A left-to-right model sees only the code before the masked position. The candidates are all distinct answers in the item set that are a single token, and the model's answer is the candidate with the highest probability at the masked position. Items whose answer spans several tokens, or whose mask lies after the first 320 tokens, count as wrong for every model and every N, which lowers absolute accuracy equally for both models. Because the benchmark is a token list rather than parsed code, renaming here treats as a variable any identifier that is not a keyword, a built-in, an attribute, a function or class name, a called name, an imported name, or a keyword-argument name.

D.7 Isolating name diversity: a small-pool control

The control model. The high-diversity model renames each local variable of every function with probability 0.5 to a fresh name that occurs nowhere else in the corpus. The control model renames exactly the same variables, in the same documents, to names of exactly the same lengths. Each new name is instead taken from a fixed pool of 1,024 random names (256 for each length from 5 to 8

letters), so the same names recur across the whole corpus. Within a document, pool names never coincide with each other or with any existing identifier, so every renaming preserves the program's meaning. In 0.05% of cases, a document needed more pool names of one length than the pool holds, and the fresh name was kept. We verified on 2,160 documents that, with the pool names swapped back, the control's training text is byte-for-byte identical to the high-diversity model's. The two corpora differ in size by 0.04% of tokens. Architecture, trainer, seeds and number of steps are identical. The two renamed models therefore differ only in how many distinct names the renamed variables receive.

Evaluation. All three models are re-evaluated together with the procedure of Appendix D.6. This run keeps slightly fewer test items than Figure 6 and Table 4, so its values differ slightly from those; all comparisons in this appendix are on identical items.

Results. Figure 16 compares the three models. On identifier reproduction and variable-misuse localization, the control is no more robust than the baseline, whereas the high-diversity model is: after renaming eight variables, the high-diversity model's retention exceeds the control's by 0.225 ± 0.020 and 0.106 ± 0.030 (difference between models trained with the same seed, mean $\pm$ standard deviation over three seeds). On these tasks, the benefit comes from name diversity. On the cloze test, the control is also less robust, but the difference is small (0.049 ± 0.033).

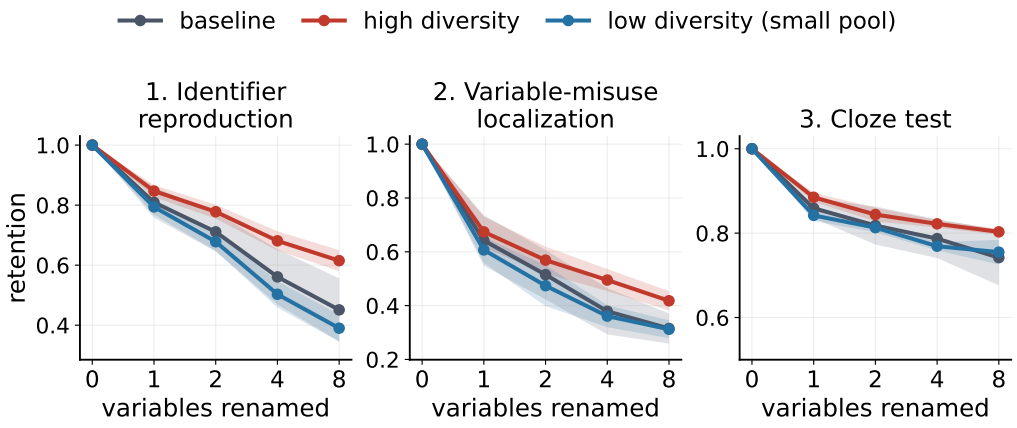

Figure 16: Retention after renaming N variables for the baseline, the high-diversity model (fresh names) and the low-diversity control (the same renaming with a small pool of names). Shaded bands: ± 1 standard deviation over three training seeds.

E Details of the head analyses and interventions

E.1 Tracing model (Sections 4.3 and 4.4)

Query positions. A query is a name-ending position in a generated trace: the position of the last character of a variable name, where the correct next token is the identifier delimiter. We include only queries whose variable occurs earlier in the program, ensuring that an earlier delimiter is available to retrieve. All positions are teacher-forced: the model reads the correct trace up to the query.

Copy score. For head h and query q, the copy score is the sum of the attention weights that h assigns at q to the delimiter positions immediately following earlier occurrences of the same variable. A head's score is the mean over all queries. Heads are ranked on renamed programs from the selection half of the test programs and evaluated on the other half. Figure 3 reports the score of the highest-ranked head.

Forced attention. Each probe is a name-ending query in a renamed program. We construct three sets of 1,024 probes, each drawn from the failures of one of the three 0.5%-diversity models. Every

model is evaluated on all 1,024 probes of each set; the failures in Table 5 are the probes the unmodified model gets wrong, summed over the three sets (at most 3,072). Every model is evaluated on all three sets. At the probe position only, we replace each selected head's attention row with equal weights on the target positions. The head's output is recomputed as $\sum_j a_j v_j$ using the model's own value vectors. All other heads, all other positions and all earlier layers remain unchanged, while the layers above are recomputed. As a self-check, recomputing each head's output with its original attention weights reproduces the model to within 3×10^{-5} in the logits, with no changes in prediction.

A *repair* is a probe where the unmodified model is incorrect and the intervened model's most likely next token is a delimiter. The *repair rate* is the number of repairs divided by the number of probes the unmodified model gets wrong. For each model, we pool the counts over the three probe sets, then average the resulting repair rates over the three training seeds. Repair is a local, single-token measure: it does not imply that the rest of the trace is generated correctly.

Conditions. (i) *Matching name*: the three highest-ranked copy heads attend to the delimiters after earlier occurrences of the same name. (ii) *Random heads*: three heads drawn from the same layers as the selected heads, with the same targets. We pool results over five draws. (iii) *Different name*: the selected heads attend to the delimiters after earlier occurrences of all other variables in the program. This condition is restricted to probes where such occurrences exist (98–99% of probes). We compare it with condition (i) on exactly the same probes.

diversity	seed	selected heads (k=3)	failures	matching name	different name†	random heads
0.5%	1	L9H12, L9H13, L9H15	2878	446 (.155)	277 / 439 of 2834	98 / 14390 (.007)
0.5%	2	L9H12, L9H13, L9H14/15	2760	679 (.246)	489 / 668 of 2724	142 / 13800 (.010)
0.5%	3	L9H13, L9H14, L7H14	2627	672 (.256)	413 / 666 of 2589	183 / 13135 (.014)
1%	1	L9H10, L9H15, L9H12/13	2034	545 (.268)	368 / 540 of 2008	93 / 10170 (.009)
1%	2	L8H12, L9H14, L9H13/15	2040	832 (.408)	574 / 824 of 2026	161 / 10200 (.016)
1%	3	L8H12, L9H13, L9H14	1308	568 (.434)	360 / 562 of 1299	281 / 6540 (.043)

Table 5: Forced attention in the tracing model: repairs over failures, pooled over the three probe sets. Layers are numbered from 0 (12 layers, 16 heads each). The selected set can differ by one head across the three probe sets (shown as H14/15). †Repairs with a different name's delimiters / repairs with the matching name, on the same probes (those where another variable occurs earlier). Random heads: five draws of three heads from the selected heads' layers, pooled.

E.2 StarCoder models (Section 4.5)

Items. We use the evaluation functions of Appendix D.6. An item is an occurrence of a local variable that has appeared earlier in the same function. Each item is evaluated with teacher forcing twice: once in the original function, and once after every local variable in the function is renamed to a random name of the same length. We retain an item only if the name aligns with token boundaries in both versions and the item falls within the 320-token window. Using a fixed seed, we randomly split the functions into two halves, A and B. Heads are ranked on half A, and the head analyses (copy score, mean ablation and forced attention) are measured on half B (9,548 items per seed); the first-error analysis involves no head selection and uses all functions (18,723 items per seed).

Query and decision. The query is the position immediately before the name's first token. The decision is the model's most likely next token at this position, which we compare with the name's first token.

Where the errors fall ("Localising the failure"). We look at the variable names that the model writes correctly in the original function but incorrectly after renaming. For each of these names, we find the first token the model gets wrong. It can be the first token of the name, a token in the middle, or the end of the name (the model writes every token correctly but does not stop). The percentage reported in Section 4.5 is the share of these names whose first error is at the first token, computed over all names and averaged over the three training seeds.

At low diversity, 15,267 names (summed over seeds) are correct before renaming and wrong after; 89.6% of them first go wrong at the first token, 9.7% at a middle token and 0.7% at the end. Because a short name has fewer places where an error can occur, we also split by the length of the renamed name: the first-token share is 97.3%, 96.2%, 91.7% and 84.1% for names of one, two, three, and four or more tokens. At high diversity (10,261 names), it is 96.9% overall and between 95.9% and 97.9% at every length.

Copy score (Figure 7-A). For head h and query q, the copy score is the sum of the attention weights at q on the first-token positions of all earlier occurrences of the same variable. A head's score is the mean over queries. Heads are ranked by this score on the renamed functions of half A. Figure 7-A reports the largest half-B score among the five highest-ranked heads, measured separately on original (ID) and renamed (OOD) functions.

Mean ablation (Figure 7-B). We replace each removed head's output, at every position, with its mean output over all tokens of the half-A functions. This mean is computed separately for the original and renamed text. We remove the k highest-ranked copy heads, $k = 1, \ldots, 15$. The random control uses ten random sets of k heads drawn from the remaining heads. We measure first-token accuracy on half B.

Forced attention. A *failure* is a half-B item whose first token the model predicts correctly with the original names but incorrectly after renaming. At the query only, we replace each selected head's attention row with equal weights on the target positions. The head's output is recomputed using the renamed function's own value vectors; in this multi-query architecture, all heads of a layer share one value vector per position. Nothing else changes. A *repair* is a failure for which the intervened model's most likely token is the correct first token. The repair rate is the number of repairs divided by the number of failures. The self-check reproduces the model to within 4×10^{-5}, with no changes in prediction.

Conditions, and equal head counts. (i) The k highest-ranked copy heads attend to the first tokens of earlier occurrences of the same variable. (ii) Random heads: for each selected head, we draw one other head from the same layer, once per model. The control therefore uses the same number of heads, k, and, in all but one case, the same layer composition. The exception occurs in one model (higher diversity, seed 1, $k = 12$): layer 18 had no unselected heads left, so the two remaining random heads were drawn from layer 0. (iii) Different variable: the selected heads attend to the first tokens of earlier occurrences of all other variables in the function, restricted to failures where such occurrences exist.

We ran $k = 3$ and $k = 12$, always comparing k copy heads with k random heads. The main text reports $k = 12$, because copying is distributed across many heads at this scale (Figure 7-B). With $k = 3$, the copy heads repair 70% and 76% of failures (low and high diversity), compared with 19% for three random heads. Because each three-head random set is a single draw, its repair rate varies widely across seeds (from 2% to 35% at low diversity and from 2% to 49% at high diversity). With twelve heads, the random control's repair rate remains at or below 13% in every seed.

"Writes that identifier." In condition (iii), we count the model as writing the other variable if its most likely token matches the *first token* of one of that variable's earlier occurrences. We do not require the complete name to be predicted correctly. We did not exclude items where another variable's first token matches the correct first token; in such cases, one prediction can count both as a repair and as writing the other variable. These double counts are a subset of the condition-(iii) repairs, which are at most 1.2% per model at $k = 12$. They therefore increase the writing rate by at most 1.2 points, while the writing rate exceeds the no-intervention rate by 16 to 49 points in every model.

The redirected heads attend to all other variables of the function at once, not to one designated alternative. The result therefore shows that the prediction is steered toward other variables' first tokens; it does not show that the model generates one specific complete name.

Results per seed. There are 2,427 to 2,527 failures per model at low diversity and 1,818 to 1,887 at high diversity. With twelve heads, redirecting the copy heads to the target repairs 75.6% of

failures at low diversity (68.1–81.0% across seeds) and 77.5% at high diversity (69.3–81.7%); twelve random heads repair 3.3% (0.1–7.8%) and 5.0% (0.0–13.3%). Redirecting the copy heads to the other variables repairs 0.7% and 1.1% of failures. It shifts the prediction toward those variables' first tokens in 63.2% (51.3–80.1%) and 69.6% (57.1–84.0%) of the failures where another variable occurs earlier (1,790 to 2,498 per model), against 34.5% (33.2–35.3%) and 38.2% (35.1–40.7%) without intervention.

Selected heads. In every model, the copy heads lie in the last five of the 20 layers, although their exact identities differ across training seeds (Table 6).

diversity	seed	the twelve highest-ranked copy heads, in order (the first three form the $k = 3$ set)
low	1	L18H8 L15H2 L16H1 L17H9 L16H9 L17H6 L16H7 L18H11 L17H8 L16H8 L18H1 L15H8
low	2	L17H10 L17H3 L18H5 L17H9 L18H11 L18H2 L17H8 L16H5 L18H9 L16H10 L16H4 L15H4
low	3	L17H11 L18H10 L18H3 L19H11 L18H9 L17H9 L18H4 L17H0 L19H5 L19H3 L17H10 L17H6
high	1	L18H4 L18H11 L18H0 L18H8 L18H1 L17H8 L18H9 L19H8 L17H2 L17H6 L18H6 L17H11
high	2	L18H9 L17H3 L17H5 L18H11 L16H10 L17H8 L18H4 L17H0 L17H9 L16H0 L18H6 L17H7
high	3	L17H10 L16H2 L17H11 L17H5 L18H10 L16H4 L18H3 L16H6 L18H9 L16H11 L16H5 L17H0

Table 6: Copy heads of the 158M models, ranked on half A of the renamed functions. Layers numbered from 0 (20 layers, 12 heads each).