DO LMS KNOW WHAT VARIABLES DO? INTERPRETING CODE REPRESENTATIONS VIA VARIABLE ROLES

Anonymous authors
Paper under double-blind review

ABSTRACT

Variable roles describe recurring behaviors such as accumulation, traversal, and retaining the best candidate encountered. We use Sajaniemi's independently defined taxonomy to study which role distinctions can be recovered from large language models, how sparse autoencoders (SAEs) retain those distinctions, and whether associated directions support control of generation. We introduce static annotation rules for all eleven roles and a corpus of 126,926 natural-code files across five languages. Across five model–SAE systems, residual probes outperform individual SAE features and tested lexical baselines, with modest losses after neutral target renaming. Combining SAE features closes much of the single-feature gap, but probes on SAE reconstructions are less sensitive to role-changing edits reviewed by LLMs. High paired ranking accuracy coexists with frequent false positives at fixed thresholds. During generation, interventions often change role annotations at the cost of correctness; changes accompanied by passing tests show no clear aggregate advantage over random controls. The benchmark separates role-related recoverability, SAE fidelity, and correctness-preserving control through matched prediction, source edits, and generation interventions.

1 INTRODUCTION

Large language models generate and modify source code, from code completion to agents that plan, edit, and test software (Jiang et al., 2024; Dong et al., 2025). Yet programming success alone does not establish which program properties a model represents: models may exploit identifiers and familiar code patterns, and their performance can change under controlled input transformations (Hooda et al., 2024). We investigate this question through the behavior of individual variables. A variable may advance through a predictable sequence, accumulate values, or retain the best candidate seen so far. These behaviors depend on relationships across updates: recognizing an addition alone does not distinguish a regular progression from accumulation of input-dependent data. The same behavior can also use different names and language constructs, motivating three questions: *which role distinctions are recoverable, how faithfully do sparse autoencoders retain them, and can associated directions change generation while preserving correctness?*

We investigate these questions using Sajaniemi's *variable roles*, developed in the psychology of programming to characterize recurring uses of variables (Sajaniemi, 2002; Sajaniemi & Navarro Prieto, 2005). The taxonomy defines behavioral criteria independently of the models under study. Prior studies recover program structure and intermediate execution states from model representations (Hernández López et al., 2022; Jin & Rinard, 2024) and find shared representations across programming languages (Kargaran et al., 2025; Yin et al., 2026). We extend these analyses to variable behavior through matched prediction, controlled source edits, and generation interventions.

We combine program analysis with sparse autoencoder (SAE) features and probes, contributing:

- **Large-scale multilingual dataset and benchmark.** We introduce a dataset of 126,926 natural-code files across five programming languages, to our knowledge the first to automatically annotate all eleven Sajaniemi variable roles at this scale. We construct a benchmark with matched and mined negatives, LLM-reviewed role-changing edits, and role-preserving target renaming to measure recoverability under controlled comparisons.

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.

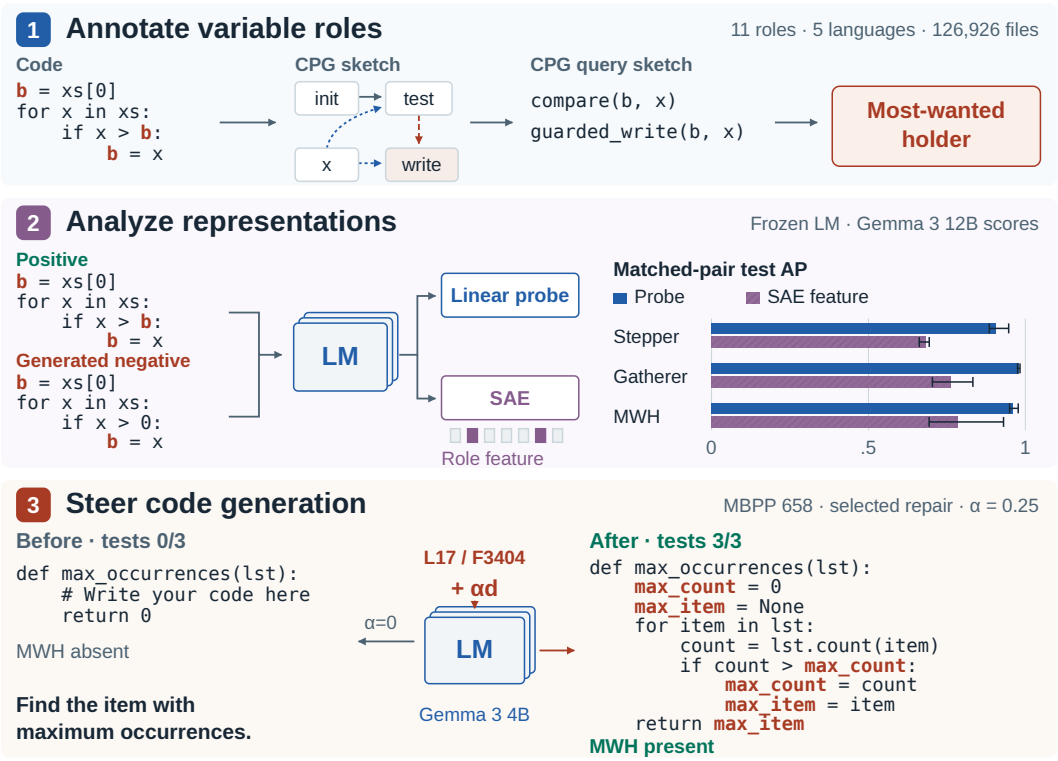


Figure 1: Variable roles connect code annotation, representation analysis, and steering. **(1)** Code Property Graph (CPG) queries annotate all eleven Sajaniemi variable roles at scale. **(2)** Probes and SAE features recover role information; bars show matched-pair test average precision (AP) for Gemma 3 12B (four-layer mean and range). Code and graph sketches illustrate the method. **(3)** In Gemma 3 4B, steering a most-wanted-holder (MWH) feature changes a stub into a working search.

We will publicly release the benchmark, annotations, evaluation splits, and extraction pipeline, with instructions for reconstructing the corpus.

- **Recoverability and SAE fidelity.** Across five model–SAE systems, we compare residual probes, reconstruction probes, individual features, and probes combining 64 features at observation points after the selected operation. Controlled edits test sensitivity beyond matched prediction; language, implementation, and readout comparisons qualify recoverability.
- **Limits of intervention-based control.** We measure role induction and suppression alongside functional correctness and random-direction controls. Annotation changes are substantially more frequent than changes accompanied by passing tests; Figure 1 illustrates a selected successful repair.

2 Background: Variable Roles in Program Comprehension

Program comprehension and programming knowledge. Research on human program comprehension distinguishes representations of operations and control flow from a *situation model* of program goals, with task-dependent prominence (Pennington, 1987). *Programming plans* describe familiar patterns of operations that accomplish particular goals (Soloway & Ehrlich, 1984). Sajaniemi's roles bring this perspective to individual variables through *variable plans*, or stereotyped uses of variables. The taxonomy originated in textbook programs; subsequent card sorting and interviews with professional programmers recovered groupings consistent with the roles, alongside variation in behavioral interpretations (Sajaniemi & Navarro Prieto, 2005).

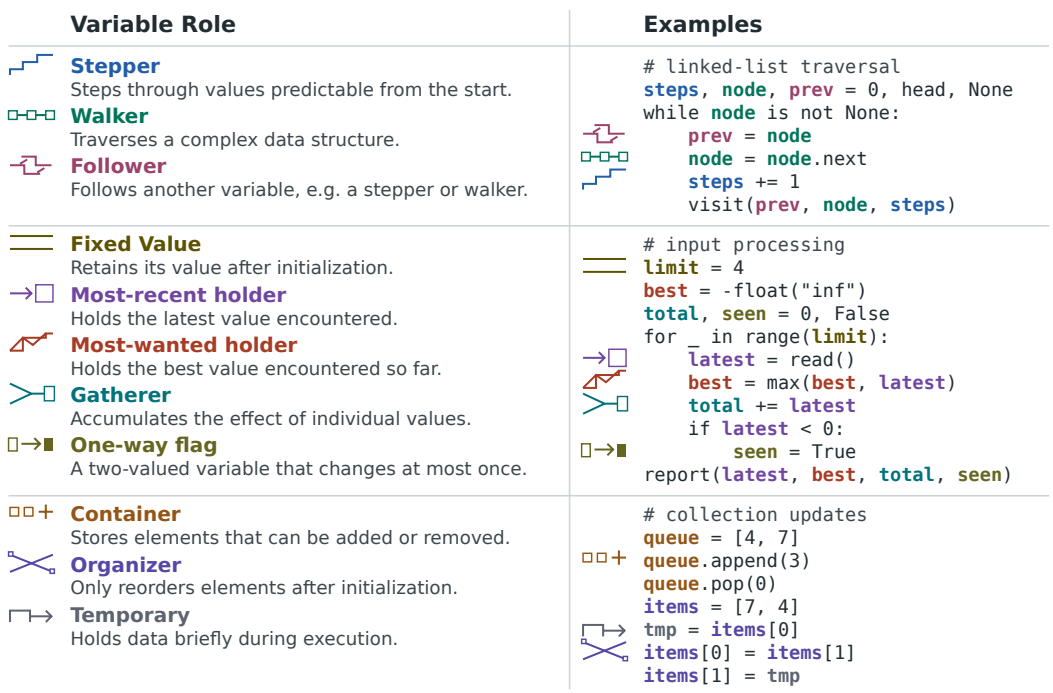


Figure 2: Overview of conceptual variable roles according to Sajaniemi (2002) and Sajaniemi & Navarro Prieto (2005). Matching colors and symbols link definitions to variable occurrences in Python code. Examples are deliberately simple, chosen for illustration. Appendix A shows more complex examples from real-world code.

Variable roles as behavioral concepts. A role characterizes a variable through its successive values and how they are obtained (Sajaniemi, 2002; Sajaniemi & Navarro Prieto, 2005): a *stepper* follows a predictable progression, a *gatherer* combines contributions, and a *most-wanted holder* retains the best candidate encountered. Figure 2 summarizes all eleven Sajaniemi variable roles; Appendix L illustrates alternative implementations. Roles describe variable behavior, so role preservation differs from program equivalence: minimum and maximum computations can share a role, whereas equivalent programs can use variables with different roles. Although defined independently of the models, these roles can correlate with identifiers or syntax; testing their representation therefore requires controls for such surface cues.

Variable roles in neural models of code. Hussain et al. (2023) detect steppers and walkers and inject their roles into Code2Seq representations. Demirtaş et al. (2025) use LLMs to detect programming plans in student programs. These studies use programming abstractions as input features or prediction targets; we compare role recoverability under controlled source edits with intervention outcomes during generation. This comparison requires annotations defined independently of model activations and linked to individual variables in natural code.

3 Annotating Variable Roles at Scale

To evaluate all eleven roles in multilingual natural code, we develop an automatic annotation procedure with explicit behavioral contracts.

Automatic role extraction. Scaling annotation requires operational definitions of these cognitive abstractions. Prior systems cover subsets of the taxonomy, use execution traces, or adopt alternative definitions (Table 1). We operationalize all eleven Sajaniemi variable roles as explicit *role contracts* over *Joern* Code Property Graphs (CPGs), which represent syntax, control flow, and data dependencies (Yamaguchi et al., 2014). Queries recover initializations, updates, and uses, whose relationships are checked against the contracts. For example, the *most-wanted holder* contract links a candidate

Table 1: Automatic variable-role analysis and reported corpus sizes. All but Demyanova et al. use Sajaniemi's taxonomy. NR: not reported; vars.: role-labeled variables; NN: neural classifier. †: role-definition sample. Appendix B details annotation units, breakdowns, and comparability.

Work	Signal	Languages	Roles	Corpus	Labeled data
Gerdt & Sajaniemi (2004)	Static analysis	Pascal	3	NR	NR
Bishop & Johnson (2005)	Static analysis	Java	Checking	17 programs	NR
Gerdt & Sajaniemi (2006)	Static + learning	Pascal	9	87 programs	305 vars.
Demyanova et al. (2013)	Static analysis	C	14 custom	5.2 KLOC†	NR
Lui et al. (2020)	Traces + NN	C	4 classes	110 programs	8,907 cases
Hussain et al. (2023)	Syntax rules	Java	2	14,022,318 methods	1,438,035 vars.
Ours	**Static CPG analysis**	**C, C++, JS, Python, Ruby**	**11**	**126,926 files**	**1,737,758 vars.**

comparison to conditional replacement of the current value, with consistency checks on initialization and other writes. These static checks require neither execution nor a buildable project, enabling annotation of isolated source files. Appendix C gives the contracts and pseudocode.

Constructing a multilingual natural-code dataset. We apply these contracts to CodeParrot GitHub-Code, combining an expanded annotation run with an earlier role-enriched subset, including Ruby. Deduplicating sources and matching declarations yields 126,926 files across five languages and 3,525,530 variables, of which 1,737,758 have at least one extracted role. Variables lacking a target annotation serve as candidate negatives, including any role instances missed by the extractor. For evaluation, we form five approximately equal folds stratified by role and language, keeping exact-source and detected clone groups intact. Appendix D reports role and language distributions.

Annotation validity and evaluation scope. Source inspection and implementation checks informed contract refinement, but coverage remains limited to supported static patterns. Counterfactual evaluation additionally uses case-level LLM source review of original behavior and proposed edits (Appendix H). This assesses individual contrasts; corpus-wide annotation precision and recall remain unmeasured. We therefore distinguish automatic-label prediction from sensitivity to reviewed edits. Full-corpus baselines appear in Appendix E; representation comparisons use a smaller shared cohort.

4 Recoverability of Variable-Role Distinctions

We first measure recoverability of role annotations, then test sensitivity to role-changing edits and stability under role-preserving renaming.

Matched prediction and diagnostic readout. To reduce surface differences, we pair each annotated positive with a variable lacking that annotation, matching language and declaration kind and favoring similar surface characteristics. The test set contains 4,238 balanced pairs across eleven roles in C, C++, JavaScript, and Python. Because assignment targets often precede their determining expressions, we read the state of a target reference inserted after the selected statement or loop/conditional header. Appendix H records selection, exclusions, and fitting; Appendix K illustrates the contrasts.

Four predictors separate feature restriction from reconstruction. We evaluate Gemma 2 2B, Gemma 3 4B/12B, Llama 3.1 8B, and Qwen3.5 9B at four layers each. SAEs encode residual activations as sparse feature vectors and reconstruct them through a linear decoder (Huben et al., 2024; Gao et al., 2024). Ridge probes on residual activations, SAE reconstructions, and 64 SAE features separate reconstruction fidelity from restricted feature readout; a single-feature predictor tests individual coordinates. Training fixes standardization, feature ranking, and polarity; validation selects regularization, the single feature, and decision thresholds. The *name* and *code* surface baselines are

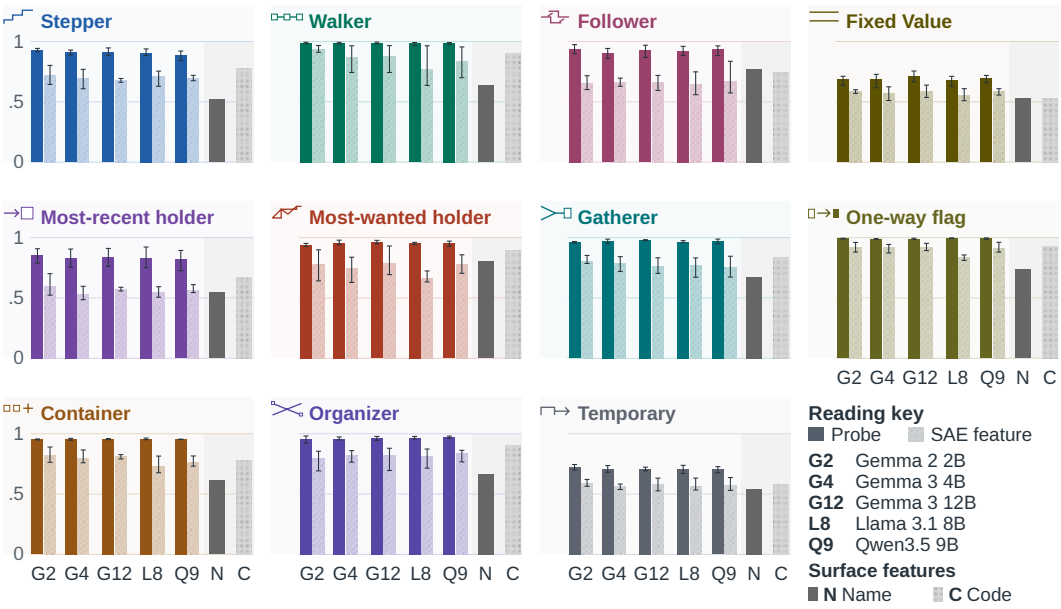


Figure 3: **Model representations support role recovery beyond the tested surface baselines.** Residual probes attain higher mean test average precision (AP) than both surface baselines and individual SAE features for every role and model. Evaluation uses matched C, C++, JavaScript and Python pairs from one source-group split. Model bars average AP over four prescribed layers; whiskers show the layer range. Gray baselines use character n-grams of names (N) or code prefixes through the causal readout (C). Role colors and symbols follow Figure 2; solid and hatched bars denote Probe and validation-selected SAE feature.

Predictor	Matched AUROC	Counterfactual paired accuracy
Residual probe	0.906–0.913	0.813–0.877
SAE reconstruction probe	0.857–0.881	0.739–0.784
Top-64 SAE probe	0.853–0.878	0.722–0.753
Single SAE feature	0.722–0.782	0.622–0.713

Table 2: **Multiple-feature readouts close much of the single-feature gap; reconstruction probes have lower counterfactual sensitivity.** Values span five model means, each weighting eleven roles and four prescribed layers equally, with matched-negative training. Ranges show variation across systems; paired uncertainty for reconstruction-minus-residual differences is reported in the text. Counterfactual accuracy compares each original only with its corresponding source-reviewed edit.

TF–IDF logistic classifiers over character n-grams of the target name and the code prefix through the diagnostic token, respectively. Fits are frozen before evaluation; Appendix F lists SAE releases.

Matched recoverability exceeds the tested lexical baselines. Residual probes exceed both surface baselines and individual SAE features in every role–model comparison (Figure 3). Averaging roles, models, and layers equally, average precision (AP) is 0.90 for residual probes and 0.72 for single features; constant-score AP is 0.50. Code features are competitive with individual SAE features: for most-wanted holder, their AP is 0.90, versus 0.75 for single features and 0.95 for residual probes. Reconstruction and top-64 probes close much of the single-feature gap (Table 2), showing the cost of restricting the readout to one coordinate.

Counterfactual ranking and classification give different answers. To test whether matched recoverability tracks defining behavior, we evaluate edits that remove the target role. On 1,698 LLM-reviewed pairs from 948 source groups, residual probes rank originals above edited negatives with paired accuracy 0.813–0.877; ties receive half credit. The code-prefix baseline reaches 0.750, while unchanged names give the name baseline 0.500. Reconstruction probes fall 4.0–12.3

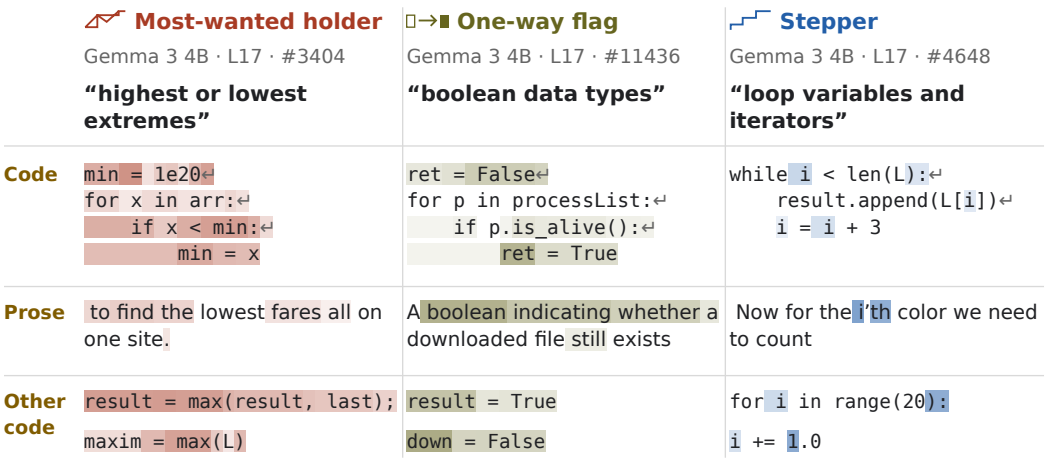


Figure 4: **Role-associated latents connect code updates to broader extrema, Boolean, and indexing concepts.** Role colors and symbols follow Figure 2; descriptions and prose come from Neuronpedia, and code from our training corpus. Highlight intensity indicates latent activation strength on a linear scale within each latent; return marks denote newlines. Code activations use standalone snippets; prose windows retain their original Neuronpedia contexts.

percentage points below residual probes, with all five paired 95% source-group bootstrap intervals excluding zero: reconstruction retains less counterfactual sensitivity. At the validation thresholds, however, residual probes classify 68–71% of edited negatives as positive, yielding balanced accuracy of 0.610–0.637. The scores therefore respond to role-changing edits more reliably than their binary decisions reject the edited examples.

Neutral target renaming preserves most ranking performance. As a complementary role-preserving control, renaming both members of 100 test pairs per role reduces residual AUROC (area under the ROC curve) by 0.6–1.7 percentage points, to 0.878–0.887. On these same pairs, code-surface AUROC changes from 0.740 to 0.723 and name-surface AUROC from 0.602 to 0.492. Renaming preserves other identifiers, comments, and implementation cues, isolating sensitivity to the target spelling. Appendix G reports uncertainty and decision changes; Appendix I evaluates prompted recognition.

Readout and implementation condition the signal. The diagnostic readout also matters because it follows the selected operation. At the second prescribed layer, residual counterfactual paired accuracy is 0.495–0.594 at the native identifier, 0.790–0.915 at statement end, and 0.856–0.920 at the inserted identifier. Separately fitted readouts thus expose much of this sensitivity at later unmodified positions. Even at the diagnostic readout, recoverability varies: with multilingual training, stepper AUROC ranges from 0.682 in JavaScript to 0.972 in Python; implementation contrasts can also be confounded by language (Appendix G).

Figure 4 illustrates the broader extrema, Boolean, and indexing associations of selected features. These contexts aid interpretation but do not establish role specificity (Appendix N). Neither contextual associations nor counterfactual sensitivity establish control; we next test interventions during generation.

5 Role Interventions and Correctness-Preserving Control

To test control beyond recoverability, we apply activation addition and direction removal (Turner et al., 2023; Arditi et al., 2024; Ferrando et al., 2024; Tahimic & Cheng, 2025) on Mostly Basic Programming Problems (MBPP) (Austin et al., 2021). The SAE-feature experiments cover Gemma 2 2B and Gemma 3 4B/12B; a separate comparison normalizes probe and SAE directions under a correctness constraint. For each role r, binary extractor labels on reference (R_{ref}) and unsteered (R_0) solutions define cohorts. These labels describe implementations: a correct alternative

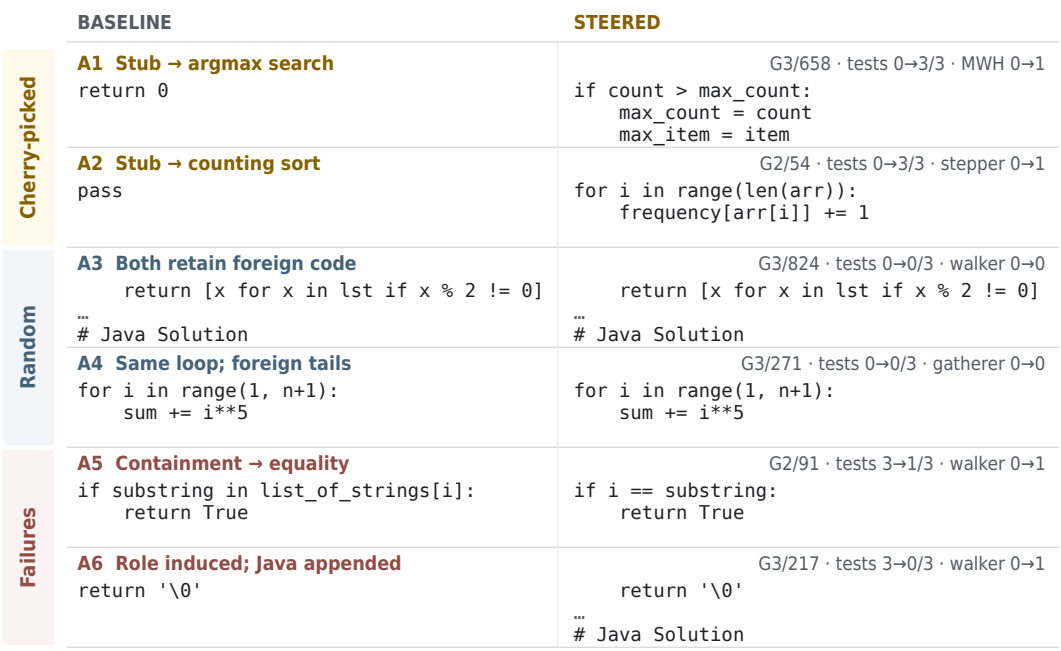


Figure 5: **Positive steering can repair stubs but can also break correct programs.** Each pair shows held-out MBPP excerpts; test counts are passing supplied assertions out of three, and role bits denote extractor output. G2/G3 denote Gemma 2 2B/Gemma 3 4B; MWH denotes most-wanted holder. The selected repairs and failures are illustrative, while the random cases come from a uniform draw without replacement. Outcomes evaluate complete outputs, including any omitted foreign-language tails.

may use a different role. Target and random directions share eligible tasks; Pass@1 measures the fraction passing the supplied tests.

Reference-conditioned induction. For feature f, we add αd_f, where $d_f = W_{\text{dec}}[f, :]$, to the residual stream at every generated token; R_α denotes the resulting role annotation. The operational score *sufficiency* measures induction of an annotation present in the reference but absent from the baseline:

$$\text{Sufficiency}_r = P(R_{\alpha^\star} = 1 \mid R_{\text{ref}} = 1, R_0 = 0).$$

Eligible examples are split equally into calibration and test sets; $\alpha^\star$ maximizes calibration sufficiency over a predefined strength grid and is fixed for testing. Reference-negative induction, denoted *spurious induction* in Figure 6, uses a separate cohort:

$$\text{SpuriousInduction}_r = P(R_{\alpha^\star} = 1 \mid R_{\text{ref}} = 0, R_0 = 0).$$

Because reference-negative induction may yield a valid alternative, both cohorts require separate correctness evaluation.

Reference-conditioned suppression. To test suppression of roles present in both reference and baseline, we normalize the decoder direction to $\hat{d}_f$ and orthogonalize matrices writing to the residual stream (Arditi et al., 2024; Ferrando et al., 2024; Tahimic & Cheng, 2025):

$$W'_{\text{out}} = W_{\text{out}} - (W_{\text{out}}\hat{d}_f^\top)\hat{d}_f.$$

The operational score *necessity* is the frequency of role absence after removal, denoted by $R_{\text{ort}} = 0$:

$$\text{Necessity}_r = P(R_{\text{ort}} = 0 \mid R_{\text{ref}} = 1, R_0 = 1).$$

With no fitted strength, removal uses all eligible examples. Unlike generation-time addition, global removal also changes prompt processing; effect sizes reflect different perturbation scopes.

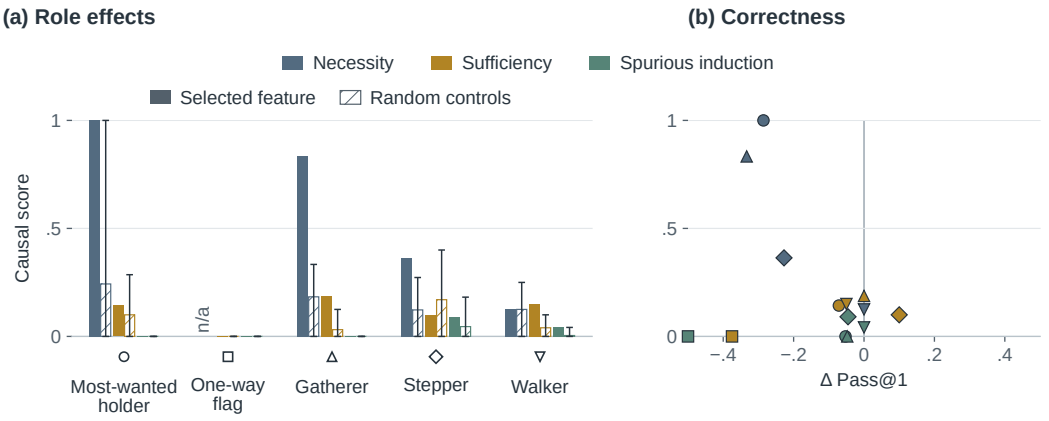


Figure 6: **Causal effects of representative role-associated SAE features in Gemma 2 2B.** (a) Scores for one selected feature per role and same-layer random SAE controls (hatched means; range whiskers); n/a denotes undefined necessity. (b) Causal scores versus changes in MBPP Pass@1; shapes match role labels in (a). Removal suggests causal involvement with limited specificity and substantial correctness costs; positive steering provides weak evidence of sufficiency.

		Gemma 2 2B		Gemma 3 4B	
Role	Representative syntax	Pos. steer.	Orth.	Pos. steer.	Orth.
Gatherer	Augmented assignment	+1.9 (107)	−58.8 (17)	+2.3 (44)	−50.0 (4)
Most-wanted holder	`if` statement	−4.3 (92)	−35.0 (20)	+17.6 (17)	−75.0 (4)
One-way flag	Boolean literal	−5.9 (17)	–	−11.1 (18)	–
Stepper	`range` call	+20.3 (59)	−20.0 (40)	+9.3 (54)	−63.6 (11)
Walker	Subscript expression	−2.5 (40)	+0.0 (8)	−5.1 (39)	+0.0 (10)

Table 3: **Role-adjacent syntax changes after intervention.** Cells give intervention-minus-baseline changes (percentage points) in program frequencies of Tree-sitter patterns, with counts of pairs with two clean parses in parentheses. Patterns are structural correlates, not role definitions or evidence of full-role sufficiency. Cohorts and baselines differ by intervention; dashes mark unavailable removal generations. Counts repeat tasks across features; Appendix P reports parse survival.

Removal suggests causal involvement with limited specificity. Some Gemma 2 2B features have high necessity (Figure 6). Gatherer suppression exceeds its random-control range, whereas most-wanted holder's score of 1.00 ties its control maximum. Their Pass@1 losses (-0.33 and -0.29) suggest suppression may partly reflect broader generation disruption, limiting evidence of role-specific control.

This correctness cost persists across all selected SAE features: removal suppresses annotations in 41/91, 72/109, and 131/187 target–task outcomes for Gemma 2 2B, Gemma 3 4B, and Gemma 3 12B. Only 2/91, 4/109, and 15/187 outcomes also pass the supplied tests; requiring both baseline and intervention to pass leaves 0, 3, and 13. Rates of annotation changes with passing tests exceed matched random controls by 1.3–3.7 percentage points for removal and 0.9–1.9 for addition, but all six task-grouped bootstrap 95% intervals include zero (Appendix P). These joint outcomes show no clear advantage over random controls.

Weak sufficiency, partial structural effects. Positive steering yields nonzero sufficiency scores of only 0.10–0.19, even where they exceed random-control ranges (Figure 6); low reference-negative induction likewise reflects weak effects. Yet Table 3 shows increased stepper-associated `range` calls under positive steering in both models, with smaller increases in gatherer-associated augmented assignments. These partial structural effects do not establish full-role sufficiency; other shifts are mixed. Only pairs with two clean parses contribute; Gemma 3 4B removal retains just 4/30 gatherer and 11/48 stepper pairs, from two and five tasks, limiting generalization.

Correctness-constrained calibration usually selects zero. To test whether calibration can avoid these correctness costs, we constrain correctness in a normalized probe/SAE comparison; ten of eleven completed model–role cells select no intervention (Appendix M). In the remaining cell, each target direction produces one induction accompanied by passing tests among five eligible tasks, matching the random residual direction. Figure 5 illustrates repairs and failures, with selected successes and unfiltered random cases documented in Appendix O.

6 Related work

These distinctions connect work on behavior, representations, and interventions (Appendix J).

Behavioral robustness and semantic priors. Semantics-preserving transformations test reliance on surface cues. ReCode finds reduced generation robustness after changes to identifiers, syntax, formatting, and docstrings (Wang et al., 2023); related mutations destabilize program-output predictions (Orvalho & Kwiatkowska, 2025). Masked-code models retain predictive ability after operand and branch reordering and identifier renaming, suggesting semantic learning (Ahmed et al., 2023).

Representations of programming concepts. Probes recover syntax and structure (Karmakar & Robbes, 2021; Wan et al., 2022), including complete abstract syntax trees (Hernández López et al., 2022), and intermediate execution states in synthetic-language models (Jin & Rinard, 2024). Whereas probes target predefined properties, CoCoA discovers token-representation clusters and interprets their lexical, syntactic, and semantic content through static analysis and LLM annotation (Sharma et al., 2025). Variable roles let us compare residual and SAE recoverability under source edits, separating association from sensitivity to defining behavior.

Shared representations across programming and natural languages. Cross-language studies test dependence on language-specific syntax. The same attention heads convey variable-type information across three programming languages (North et al., 2025), while language-specific and language-agnostic neurons coexist within a model (Yin et al., 2026). Intermediate-layer decoding and neuron ablations also reveal overlap with English (Kargaran et al., 2025). Our multilingual fits complement transfer studies by measuring variation in role recovery within each evaluated language.

Causal and mechanistic analyses. Coding failures can coexist with useful computations: steering restores disrupted type predictions (Lucchetti & Guha, 2025), while synthetic parentheses analyses trace errors to interference between reliable and unreliable components (Rai et al., 2025). Yet SAE features predictive of code correctness need not support effective steering (Tahimic & Cheng, 2025). We extend this distinction to role annotations, comparing changes, joint passing outcomes, and matched random controls.

7 Conclusion, Limitations, and Future Work

Residual probes recover operational role annotations beyond tested lexical baselines. Combining SAE features narrows the single-feature gap, but reconstruction probes retain less sensitivity to reviewed role-changing edits. Removal suggests causal involvement with limited specificity and correctness costs, while positive steering yields weak full-role induction despite structural shifts.

Limitations. Corpus-wide annotation precision, recall, and inter-annotator reliability are unmeasured. Decoding covers four layers per model and one source-group split, with uncertainty conditional on fitted predictors. The corpus includes development data without repository-level holdout, and target renaming retains other lexical cues. Interventions use small overlapping cohorts and finite prompt-visible tests, with unresolved historical scale provenance (Appendix P).

Future work. Our corpus combines natural-code complexity with independently defined roles and controlled source edits, offering a semi-controlled setting for studying conceptual abstraction beyond toy tasks. Future work can test whether role representations transfer to unseen languages and implementations, combine into higher-level programming plans, and support interventions that change variable behavior while preserving correctness.

8 USE OF AI ASSISTANCE

We used large language models and coding agents to assist literature searches, manuscript drafting and revision, visualization code, and experimental protocols. The authors reviewed all references, procedures, figures, and manuscript content before inclusion. LLM reviewers also assessed counterfactual source edits, recording original-role validity, edited behavior, and rejection reasons; Appendix H distinguishes this procedure from human annotation validation.

AI assistance was more extensive in implementing the variable-role extractor, whose multilingual static-analysis rules combine syntax, control flow, and data flow. The authors specified behavioral requirements and rejection conditions for each role (Appendix C); coding agents implemented these contracts using Joern CPG facts and the existing extraction infrastructure. We inspected implementations and annotations manually on source-code examples and automatically through checks on annotations and intermediate variable facts. Failures and false positives or negatives informed contract refinements, followed by implementation updates and re-evaluation until both stabilized. The authors controlled the choice of taxonomy, operational definitions, final annotation rules, experimental design, interpretation, and manuscript.

REFERENCES

Youssef Abdelsalam, Norman Peitek, Anna-Maria Maurer, Marvin Wyrich, and Sven Apel. A Mechanistic Lens on Semantic Conflicts: Using Activation Patching to Understand LLM Behavior, July 2026. URL http://arxiv.org/abs/2607.05587. arXiv:2607.05587 [cs.SE].

Toufique Ahmed, Dian Yu, Chengxuan Huang, Cathy Wang, Prem Devanbu, and Kenji Sagae. Towards understanding what code language models learned. *arXiv preprint arXiv:2306.11943*, 2023. URL https://arxiv.org/abs/2306.11943.

Abhinav Anand, Shweta Verma, Krishna Narasimhan, and Mira Mezini. A critical study of what code-LLMs (do not) learn. In *Findings of the Association for Computational Linguistics: ACL 2024*, pp. 15869–15889, Bangkok, Thailand, 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.findings-acl.939. URL https://aclanthology.org/2024.findings-acl.939/.

Andy Arditi, Oscar Obeso, Aaquib Syed, Daniel Paleka, Nina Panickssery, Wes Gurnee, and Neel Nanda. Refusal in language models is mediated by a single direction. In *Advances in Neural Information Processing Systems*, volume 37, pp. 136037–136083, 2024.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc V. Le, and Charles Sutton. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Craig Bishop and Colin G. Johnson. Assessing roles of variables by program analysis. In *Proceedings of the Fifth Annual Finnish/Baltic Sea Conference on Computer Science Education*. University of Joensuu, 2005. URL https://kar.kent.ac.uk/14232/.

Simin Chen, Zexin Li, Wei Yang, and Cong Liu. DeciX: Explain deep learning based code generation applications. *Proceedings of the ACM on Software Engineering*, 1(FSE), 2024. doi: 10.1145/3660814.

Siyue Chen, Yifu Guo, Yuquan Lu, Zishan Xu, Jiaye Lin, Jianbo Lin, Siyu Zhang, Cheng Yang, Junxin Li, Yujia Li, Yu Huo, and Ruixuan Wang. From Brewing to Resolution: Tracing the Internal Lifecycle of Code Reasoning in LLMs, August 2026. URL http://arxiv.org/abs/2606.17648. arXiv:2606.17648 [cs.AI].

Arthur Conmy, Augustine Mavor-Parker, Aengus Lynch, Stefan Heimersheim, and Adrià Garriga-Alonso. Towards Automated Circuit Discovery for Mechanistic Interpretability. In *Advances in Neural Information Processing Systems*, volume 36, pp. 16318–16352. Curran Associates, Inc., 2023. doi: 10.52202/075280-0719. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/34e1dbe95d34d7ebaf99b9bcaeb5b2be-Abstract-Conference.html.

Mehmet Arif Demirtaş, Claire Zheng, Max Fowler, and Kathryn Cunningham. Generating planning feedback for open-ended programming exercises with LLMs. In *Artificial Intelligence in Education*, 2025. URL https://arxiv.org/abs/2504.08958.

Yulia Demyanova, Helmut Veith, and Florian Zuleger. On the concept of variable roles and its use in software analysis. In *2013 Formal Methods in Computer-Aided Design*, pp. 226–230. IEEE, 2013. doi: 10.1109/FMCAD.2013.6679414.

Boyi Deng, Xu Wang, Yaoning Wang, Yu Wan, Yubo Ma, Baosong Yang, Haoran Wei, Jialong Tang, Huan Lin, Ruize Gao, Tianhao Li, Qian Cao, Xuancheng Ren, Xiaodong Deng, An Yang, Fei Huang, Dayiheng Liu, and Jingren Zhou. Qwen-scope: Turning sparse features into development tools for large language models. *arXiv preprint arXiv:2605.11887*, 2026. URL https://arxiv.org/abs/2605.11887.

Yihong Dong, Xue Jiang, Jiaru Qian, Tian Wang, Kechi Zhang, Zhi Jin, and Ge Li. A survey on code generation with LLM-based agents. *arXiv preprint arXiv:2508.00083*, 2025. URL https://arxiv.org/abs/2508.00083.

Javier Ferrando, Oscar Obeso, Senthooran Rajamanoharan, and Neel Nanda. Do I know this entity? knowledge awareness and hallucinations in language models. *arXiv preprint arXiv:2411.14257*, 2024. URL https://arxiv.org/abs/2411.14257.

Leo Gao, Tom Dupré la Tour, Henk Tillman, Gabriel Goh, Rajan Troll, Alec Radford, Ilya Sutskever, Jan Leike, and Jeffrey Wu. Scaling and evaluating sparse autoencoders. *arXiv preprint arXiv:2406.04093*, 2024. URL https://arxiv.org/abs/2406.04093.

Shuzheng Gao, Cuiyun Gao, Chaozheng Wang, Jun Sun, David Lo, and Yue Yu. Two Sides of the Same Coin: Exploiting the Impact of Identifiers in Neural Code Comprehension. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pp. 1933–1945, Melbourne, Australia, May 2023. IEEE. doi: 10.1109/ICSE48619.2023.00164. URL https://ieeexplore.ieee.org/document/10172869/.

Petri Gerdt and Jorma Sajaniemi. An approach to automatic detection of variable roles in program animation. In *Proceedings of the Third Program Visualization Workshop*, pp. 86–93. University of Warwick, 2004.

Petri Gerdt and Jorma Sajaniemi. A web-based service for the automatic detection of roles of variables. In *Proceedings of the 11th Annual Conference on Innovation and Technology in Computer Science Education*, pp. 178–182. ACM, 2006. doi: 10.1145/1140123.1140172.

Muhammad Umair Haider, Umar Farooq, A. B. Siddique, and Mark Marron. Looking into Black Box Code Language Models. In *2026 International Conference on Semantic Computing (ICSC)*, pp. 32–39. IEEE, 2026. doi: 10.1109/ICSC67292.2026.00011. URL https://ieeexplore.ieee.org/document/11486438/.

Ahmad Haji Mohammadkhani, Chakkrit Tantithamthavorn, and Hadi Hemmati. Explaining Transformer-based Code Models: What Do They Learn? When They Do Not Work? In *2023 IEEE 23rd International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pp. 96–106, Bogotá, Colombia, October 2023. IEEE. doi: 10.1109/SCAM59687.2023.00020. URL https://ieeexplore.ieee.org/document/10356671/.

Yicheng He, Zheng Zhao, Kaiyu Zhou, Bryan Dai, Jie Fu, and Yonghui Yang. Codecircuit: Toward inferring llm-generated code correctness via attribution graphs. *arXiv preprint arXiv:2602.07080*, 2026. URL https://arxiv.org/abs/2602.07080.

Zhengfu He, Wentao Shu, Xuyang Ge, Lingjie Chen, Junxuan Wang, Yunhua Zhou, Frances Liu, Qipeng Guo, Xuanjing Huang, Zuxuan Wu, Yu-Gang Jiang, and Xipeng Qiu. Llama scope: Extracting millions of features from llama-3.1-8b with sparse autoencoders. *arXiv preprint arXiv:2410.20526*, 2024. URL https://arxiv.org/abs/2410.20526.

Stefan Heimersheim and Jett Janiak. A circuit for Python docstrings in a 4-layer attention-only transformer, 2023. URL https://www.alignmentforum.org/posts/u6KXXmKFbXfWzoAXn/

a-circuit-for-python-docstrings-in-a-4-layer-attention-only. AI Alignment Forum technical note.

José Antonio Hernández López, Martin Weyssow, Jesús Sánchez Cuadrado, and Houari Sahraoui. Ast-probe: Recovering abstract syntax trees from hidden representations of pre-trained language models. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022. doi: 10.1145/3551349.3556900.

Ashish Hooda, Mihai Christodorescu, Miltiadis Allamanis, Aaron Wilson, Kassem Fawaz, and Somesh Jha. Do Large Code Models Understand Programming Concepts? Counterfactual Analysis for Code Predicates. In *Proceedings of the 41st International Conference on Machine Learning*, pp. 18738–18748. PMLR, 2024. URL https://proceedings.mlr.press/v235/hooda24a.html.

Li Huang, Zhongxin Liu, Yifan Wu, Tao Yin, Dong Li, Jichao Bi, Nankun Mu, Hongyu Zhang, and Meng Yan. DeepGuard: Secure Code Generation via Multi-Layer Semantic Aggregation. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 19793–19813, San Diego, California, United States, July 2026. Association for Computational Linguistics. doi: 10.18653/v1/2026.acl-long.907. URL https://aclanthology.org/2026.acl-long.907/.

Robert Huben, Hoagy Cunningham, Logan Riggs Smith, Aidan Ewart, and Lee Sharkey. Sparse autoencoders find highly interpretable features in language models. In *International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=F76bwRSLeK.

Aftab Hussain, Md Rafiqul Islam Rabin, Bowen Xu, David Lo, and Mohammad Amin Alipour. A study of variable-role-based feature enrichment in neural models of code. In *2023 IEEE/ACM International Workshop on Interpretability and Robustness in Neural Software Engineering (InteNSE)*, pp. 14–20. IEEE, 2023. doi: 10.1109/InteNSE59150.2023.00007.

Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. A survey on large language models for code generation. *arXiv preprint arXiv:2406.00515*, 2024. URL https://arxiv.org/abs/2406.00515.

Charles Jin and Martin Rinard. Emergent representations of program semantics in language models trained on programs. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, 2024. URL https://arxiv.org/abs/2305.11169.

Amir Hossein Kargaran, Yihong Liu, François Yvon, and Hinrich Schütze. How programming concepts and neurons are shared in code language models. In *Findings of the Association for Computational Linguistics: ACL 2025*, 2025. URL https://aclanthology.org/2025.findings-acl.1379/.

A. Karmakar and R. Robbes. What do pre-trained code models know about code? In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering*, 2021. URL https://arxiv.org/abs/2108.11308.

Anjan Karmakar and Romain Robbes. INSPECT: Intrinsic and Systematic Probing Evaluation for Code Transformers. *IEEE Transactions on Software Engineering*, 50(2):220–238, February 2024. doi: 10.1109/TSE.2023.3341624. URL https://ieeexplore.ieee.org/document/10354028/.

Zhiming Li, Yanzhou Li, Tianlin Li, Mengnan Du, Bozhi Wu, Yushi Cao, Junzhe Jiang, and Yang Liu. Unveiling Project-Specific Bias in Neural Code Models. In *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING 2024)*, pp. 17205–17216, Torino, Italia, May 2024. ELRA and ICCL. URL https://aclanthology.org/2024.lrec-main.1494/.

Yunhao Liang, Chengguang Gan, Ruixuan Ying, Hanjun Wei, Zhe Cui, and Shiwen Ni. Do Code Language Models Follow Tests? Paired Interventions on Program Behavior, September 2026. URL http://arxiv.org/abs/2607.26244. arXiv:2607.26244 [cs.SE].

Tom Lieberum, Senthooran Rajamanoharan, Arthur Conmy, Lewis Smith, Nicolas Sonnerat, Vikrant Varma, Janos Kramar, Anca Dragan, Rohin Shah, and Neel Nanda. Gemma scope: Open sparse autoencoders everywhere all at once on gemma 2. In *Proceedings of the 7th BlackboxNLP Workshop: Analyzing and Interpreting Neural Networks for NLP*, pp. 278–300. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.blackboxnlp-1.19. URL https://aclanthology.org/2024.blackboxnlp-1.19/.

Francesca Lucchetti and Arjun Guha. Understanding How CodeLLMs (Mis)Predict Types with Activation Steering. In *Proceedings of the 8th BlackboxNLP Workshop: Analyzing and Interpreting Neural Networks for NLP*, pp. 358–397. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.blackboxnlp-1.22. URL https://aclanthology.org/2025.blackboxnlp-1.22/.

Andrew Kwok Fai Lui, Tsz Tik Lui, Tommy Cheuk Hin Leung, and Ho Yin Chan. Recognition of roles of variables based on deep learning technologies. In *Technology in Education. Innovations for Online Teaching and Learning*, volume 1302 of *Communications in Computer and Information Science*, pp. 362–374. Springer, 2020. doi: 10.1007/978-981-33-4594-2_30.

Wei Ma, Shangqing Liu, Mengjie Zhao, Xiaofei Xie, Wenhan Wang, Qiang Hu, Jie Zhang, and Yang Liu. Unveiling Code Pre-Trained Models: Investigating Syntax and Semantics Capacities. *ACM Transactions on Software Engineering and Methodology*, 33(7):1–29, September 2024. doi: 10.1145/3664606. URL https://dl.acm.org/doi/10.1145/3664606.

Callum McDougall, Arthur Conmy, Janos Kramar, Tom Lieberum, Senthooran Rajamanoharan, and Neel Nanda. Gemma scope 2: Technical paper. Technical report, Google DeepMind, 2025.

Sharath Naganna, Tanvir Ahmed Sijan, and Uddipta Kalita. Are Arithmetic Heuristic Neurons Form-Invariant? A Mechanistic Analysis of Symbols, Text, and Code in LLMs, July 2026. URL http://arxiv.org/abs/2607.16693. arXiv:2607.16693 [cs.CL].

Marc North, Amir Atapour-Abarghouei, and Nelly Bencomo. Beyond Syntax: How Do LLMs Understand Code? In *2025 IEEE/ACM 47th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, pp. 86–90. IEEE, 2025. doi: 10.1109/ICSE-NIER66352.2025.00023. URL https://ieeexplore.ieee.org/document/11023969/.

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.

Matteo Paltenghi, Rahul Pandita, Austin Z. Henley, and Albert Ziegler. Follow-up attention: An empirical study of developer and neural model code exploration. *IEEE Transactions on Software Engineering*, 2024. URL https://arxiv.org/abs/2210.05506.

Nancy Pennington. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, 19(3):295–341, 1987. doi: 10.1016/0010-0285(87)90007-7.

Daking Rai, Samuel Miller, Kevin Moran, and Ziyu Yao. Failure by Interference: Language Models Make Balanced Parentheses Errors When Faulty Mechanisms Overshadow Sound Ones. In *Advances in Neural Information Processing Systems 38*, pp. 146964–146993. Neural Information Processing Systems Foundation, Inc. (NeurIPS), 2025. doi: 10.52202/085713-4418.

Jorma Sajaniemi. Visualizing roles of variables to novice programmers. In *Proceedings of the 14th Annual Workshop of the Psychology of Programming Interest Group*, pp. 111–127, 2002. URL https://ppig.org/files/2002-PPIG-14th-sajaniemi.pdf.

Jorma Sajaniemi and Raquel Navarro Prieto. Roles of variables in experts' programming knowledge. In *Proceedings of the 17th Annual Workshop of the Psychology of Programming Interest Group*, pp. 145–159, 2005. URL https://ppig.org/files/2005-PPIG-17th-sajaniemi.pdf.

Arushi Sharma, Zefu Hu, Christopher J. Quinn, and Ali Jannesari. Redundancy and Concept Analysis for Code-trained Language Models. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering Workshops*, pp. 24–34. ACM, 2024. doi: 10.1145/3691621.3694931. URL https://dl.acm.org/doi/10.1145/3691621.3694931.

Arushi Sharma, Vedant Pungliya, Christopher J. Quinn, and Ali Jannesari. Analyzing Latent Concepts in Code Language Models, 2025. URL https://arxiv.org/abs/2510.00476.

Rishab Sharma, Fuxiang Chen, Fatemeh Fard, and David Lo. An exploratory study on code attention in BERT. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, pp. 437–448, Virtual Event, May 2022. ACM. doi: 10.1145/3524610.3527921. URL https://dl.acm.org/doi/10.1145/3524610.3527921.

Elliot Soloway and Kate Ehrlich. Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, SE-10(5):595–609, 1984. doi: 10.1109/TSE.1984.5010283.

Kriz Tahimic and Charibeth Cheng. Mechanistic interpretability of code correctness in llms via sparse autoencoders. *arXiv preprint arXiv:2510.02917*, 2025. URL https://arxiv.org/abs/2510.02917.

Adly Templeton, Tom Conerly, Jonathan Marcus, Jack Lindsey, Trenton Bricken, Brian Chen, Adam Pearce, Craig Citro, Emmanuel Ameisen, Andy Jones, Hoagy Cunningham, Nicholas L Turner, Callum McDougall, Monte MacDiarmid, C. Daniel Freeman, Theodore R. Sumers, Edward Rees, Joshua Batson, Adam Jermyn, Shan Carter, Chris Olah, and Tom Henighan. Scaling monosemanticity: Extracting interpretable features from claude 3 sonnet. *Transformer Circuits Thread*, 2024. URL https://transformer-circuits.pub/2024/scaling-monosemanticity/index.html.

S. Troshin and N. Chirkova. Probing pretrained models of source codes. In *Proceedings of the Fifth BlackboxNLP Workshop on Analyzing and Interpreting Neural Networks for NLP*, 2022. URL https://aclanthology.org/2022.blackboxnlp-1.31/.

Alexander Matt Turner, Lisa Thiergart, David Udell, Gavin Leech, Ulisse Mini, and Monte MacDiarmid. Activation addition: Steering language models without optimization. *arXiv preprint arXiv:2308.10248*, 2023.

Yao Wan, Wei Zhao, Hongyu Zhang, Yulei Sui, Guandong Xu, and Hai Jin. What do they capture? a structural analysis of pre-trained language models for source code. In *Proceedings of the 44th International Conference on Software Engineering*, 2022. doi: 10.1145/3510003.3510050.

Shiqi Wang, Zheng Li, Haifeng Qian, Chenghao Yang, Zijian Wang, Mingyue Shang, Varun Kumar, Samson Tan, Baishakhi Ray, Parminder Bhatia, Ramesh Nallapati, Murali Krishna Ramanathan, Dan Roth, and Bing Xiang. ReCode: Robustness Evaluation of Code Generation Models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 13818–13843. Association for Computational Linguistics, 2023. doi: 10.18653/v1/2023.acl-long.773. URL https://aclanthology.org/2023.acl-long.773.

Zhilong Wang, Lan Zhang, Chen Cao, Nanqing Luo, Xinzhi Luo, and Peng Liu. How Does Naming Affect Language Models on Code Analysis Tasks? *Journal of Software Engineering and Applications*, 17(11):803–816, 2024. doi: 10.4236/jsea.2024.1711044. URL https://www.scirp.org/journal/paperinformation?paperid=137296.

Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy*, pp. 590–604, 2014. URL https://mlsec.org/docs/2014-ieeesp.pdf.

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, Pittsburgh Pennsylvania, May 2022. ACM. doi: 10.1145/3510003.3510146. URL https://dl.acm.org/doi/10.1145/3510003.3510146.

Zhe Yin, Xiaodong Gu, and Beijun Shen. Neuron-guided interpretation of code LLMs: Where, why, and how? *Proceedings of the ACM on Software Engineering*, 3(FSE), 2026. doi: 10.1145/3797083. URL https://arxiv.org/abs/2512.19980.

APPENDIX CONTENTS

A Variable roles in real-world code **17**

B Corpus sizes in prior variable-role studies **20**

C Variable-role extraction details **21**

D Dataset statistics **24**

E Metadata and lexical baselines **25**

F SAE Details **26**

G Additional Analyses of Role Representations **27**

H Evaluation populations, review, and selection **30**

I Behavioral role recognition and prompt sensitivity **32**

I.1 Role differences and free generation 33

J Extended related work **34**

K Selecting and generating negative examples **36**

L Different realizations of variable roles **38**

M Matched probe and SAE steering **40**

N Activation contexts of role-associated SAE features **41**

O Qualitative outcomes of SAE steering **43**

P Consolidated quantitative analysis of causal interventions **44**

P.1 Comparison units, calibration, and structural survival 45

A VARIABLE ROLES IN REAL-WORLD CODE

The simple examples in Figure 2 isolate each role's behavior. The six excerpts below cover all eleven Sajaniemi variable roles in natural code from our annotated corpus. Common leading indentation is removed; long lines wrap for display.

Streaming an application release. Nextcloud App Store

```python
def _stream_to_file(self, file: Any, max_size: int,
                    req: requests.Response) -> None:
    # start streaming download
    finished = False
    try:
        size = 0
        for chunk in req.iter_content(1024):
            file.write(chunk)
            size += len(chunk)
            if size > max_size:
                msg = 'Downloaded archive is bigger than the ' \
                      'allowed %i bytes' % max_size
                raise MaximumDownloadSizeExceededException(msg)
        finished = True
    finally:
        # in case any errors occurred, get rid of the file
        file.close()
        if not finished:
            os.remove(file.name)
```

The *fixed value* `max_size` bounds the download. The *gatherer* `size` adds input-dependent chunk lengths, and the *one-way flag* `finished` changes to true only after streaming completes. The *temporary* `msg` holds an exception message. The `finally` block uses the flag to clean up after an error.

Applying byte-level delta operations. Subvertpy

```python
def txdelta_apply_ops(src_ops, ops, new_data, sview):
    """Apply txdelta operations to a source view.

    :param src_ops: Source operations, ignored.
    :param ops: List of operations (action, offset, length).
    :param new_data: Buffer to fetch fragments with new data from
    :param sview: Source data
    :return: Result data
    """
    tview = bytearray()
    for (action, offset, length) in ops:
        if action == TXDELTA_SOURCE:
            # Copy from source area.
            tview.extend(sview[offset:offset+length])
        elif action == TXDELTA_TARGET:
            for i in range(length):
                tview.append(tview[offset+i])
        elif action == TXDELTA_NEW:
            tview.extend(new_data[offset:offset+length])
        else:
            raise Exception("Invalid delta instruction code")
    return tview
```

The *walker* `action` takes the operation code from each tuple in `ops`. The *stepper* `i` advances through the range of each target-copy operation. The *container* `tview` grows through bulk extension and single-byte appends. These roles coexist in a dispatch loop for different copy operations.

Reading blocks and emitting delta windows. Subvertpy

```
def send_stream(stream, handler, block_size=DELTA_WINDOW_SIZE):
    """Send txdelta windows that create stream to handler

    :param stream: file-like object to read the file from
    :param handler: txdelta window handler function
    :return: MD5 hash over the stream
    """
    hash = md5()
    text = stream.read(block_size)
    if not isinstance(text, bytes):
        raise TypeError("The stream should read out bytes")
    while text:
        hash.update(text)
        window = (0, 0, len(text), 0, [(TXDELTA_NEW, 0, len(text))], text)
        handler(window)
        text = stream.read(block_size)
    handler(None)
    return hash.digest()
```

The *most-recent holder* `text` is replaced by each block read from the stream. Each block updates the digest and enters a delta window before the next read. A type check and a final callback surround this acquisition–use cycle.

Monitoring peak memory usage. ipython_memory_usage

```
def during_execution_memory_sampler():
    import time
    import memory_profiler
    global keep_watching, peak_memory_usage
    peak_memory_usage = -1
    keep_watching = True

    n = 0
    WAIT_BETWEEN_SAMPLES_SECS = 0.001
    MAX_ITERATIONS = 60.0 / WAIT_BETWEEN_SAMPLES_SECS
    while True:
        mem_usage = memory_profiler.memory_usage()[0]
        peak_memory_usage = max(mem_usage, peak_memory_usage)
        time.sleep(WAIT_BETWEEN_SAMPLES_SECS)
        if not keep_watching or n > MAX_ITERATIONS:
            # exit if we've been told our command has finished or if it has
↪ run
            # for more than a sane amount of time (e.g. maybe something
↪ crashed
            # and we don't want this to carry on running)
            if n > MAX_ITERATIONS:
                print("{} SOMETHING WEIRD HAPPENED AND THIS RAN FOR TOO LONG,
↪  THIS THREAD IS KILLING ITSELF".format(__file__))
            break
        n += 1
```

The *most-wanted holder* `peak_memory_usage` retains the largest observed usage during each sampling run. Its monitoring loop also handles a stop signal, a sampling delay, and a timeout.

Searching parent directories for configuration. Projeny

```
def findConfigPath(filePath):
    lastParentDir = None
    parentDir = os.path.dirname(filePath)

    while parentDir and parentDir != lastParentDir:
        configPath = os.path.join(parentDir, Prj.ConfigFileName)

        if os.path.isfile(configPath):
            return configPath

        lastParentDir = parentDir
        parentDir = os.path.dirname(parentDir)

    assertThat(False, "Could not find Prj config path starting at path {0}",
    ↪ filePath)
```

The *follower* `lastParentDir` preserves `parentDir`'s old value before the next parent-directory step. Their comparison detects when traversal stops progressing; a file-existence check allows early return.

Computing a median with fallback handling. LIBXSMM

```
def median(list_of_numbers, fallback = None, average = True):
    size = len(list_of_numbers)
    if (0 < size):
        # TODO: use nth element
        list_of_numbers.sort()
        size2 = int(size / 2)
        if (average and 0 == (size - size2 * 2)):
            medval = int(0.5 * (list_of_numbers[size2-1] + list_of_numbers[
    ↪ size2]) + 0.5)
        else:
            medval = list_of_numbers[size2]
        if (None != fallback):
            result = min(medval, fallback)
        else:
            result = medval
    elif (None != fallback):
        result = fallback
    else:
        sys.tracebacklimit = 0
        raise ValueError("median: empty list!")
    return result
```

The *organizer* `list_of_numbers` sorts the elements; subsequent computation only reads the reordered list. The function also handles even-sized input, a fallback value, and empty input.

B CORPUS SIZES IN PRIOR VARIABLE-ROLE STUDIES

Table 4 expands Table 1, distinguishing source-code pools from labeled examples in each study's reported units.

Table 4: Corpus sizes and annotation scope for Table 1. Totals combine reported splits where available. NR: not reported, not zero. Units and annotation quality are not directly comparable.

Study / coverage	Corpus and labeled examples	Scope and qualifications
Gerdt & Sajaniemi (2004) Pascal; 3 roles	Prototype program and variable counts: NR.	Fixed Value, Stepper, and Most-Recent Holder, selected using an earlier textbook survey.
Bishop & Johnson (2005) Java; role checking	17 canonical teaching programs; variable count: NR.	Checks supplied role assignments in procedural Java.
Gerdt & Sajaniemi (2006) Pascal; 9 roles	Learning: 43 programs, 175 variables. Evaluation: 44 programs, 130 variables. Total: 87 programs, 305 variables.	Filtered, human-annotated textbook sample; arrays omitted and role-changing variables split by role.
Demyanova et al. (2013) C; 14 custom roles	5.2 KLOC of cBench for role design. Downstream file and variable totals: NR.	Custom roles evaluated on SV-COMP file-category prediction, not role-label accuracy.
Lui et al. (2020) C; 4 classes	110 educational programs; 8,907 labeled trace cases.	Manually annotated programs; cases are not stated to be distinct declarations. One-Way Flag and Constant are merged.
Hussain et al. (2023) Java; 2 roles	14,022,318 methods; 1,438,035 role-augmented variables (624,701 steppers; 813,334 walkers).	Automatic syntax rules; pool includes methods without either role. Evaluated on method-name prediction.
Ours Five languages; 11 roles	126,926 files; 1,737,758 distinct role-annotated variables (37,006 steppers; 41,686 walkers). Full counts: Appendix D.	Automatic CPG labels; identical sources and matching declarations merged. Unlabeled variables can still have roles.

Comparability. Programs, methods, and files are different source units, as are trace cases, declarations, and variable–role memberships. Counts do not imply matched deduplication, independent examples, or comparable holdouts. Role inventories and recognition rules also differ: Gerdt & Sajaniemi (2006) include Transformation but have no Temporary evaluation examples. Automatic annotations and downstream task performance do not establish role-label accuracy.

Coverage. Our eleven-role, five-language corpus has more role-annotated variables than Hussain et al. (2023), but fewer steppers and walkers (Table 4). Coverage is uneven: Fixed Value accounts for 74.4% of role memberships, and Ruby for 53 files. Broader coverage does not imply more examples of each role or higher annotation quality.

C VARIABLE-ROLE EXTRACTION DETAILS

Section 8 describes how coding agents assisted implementation of these author-defined contracts.

EXTRACTION PSEUDOCODE 1/3

v: declaration identity; W: binding writes; I: initialization events; $U = W \setminus I$; M: observed state mutations; F: raw Fixed Values. Initialization partitions are role-specific; named checks require supporting source/CPG evidence. CFG witnesses are bounded; missing labels do not prove semantic absence.

Supplement: `src/role_extractor/scala/`: `facts/` builds evidence, `role_predicates.sc` classifies, `annotation_emission.sc` resolves labels. Frontend details and exact algorithms reside in this executable specification.

Fixed Value

if $Q(v)$: **return** $C(v) = \mathsf{Fixed}$
require $J(v)$ and $B(v)$
reject if any write violates A, or $Z(v)$ or $O(v)$ fails
accept

Key: Q: verified non-text collection; C: collection decision (third page). J: verified initialization, at most one per path (parameter entry; local initializer/exhaustive alternatives; member constructors). B: no later/external rebinding. A: plain, non-self-dependent writes. Z: no observed mutation. O: no loop initialization/rebinding.

Pattern: `limit = input_limit; use(limit)`

Temporary (raw)

require $L(v)$, $K_T(v)$, $W \neq \varnothing$
for each $w \in W$: **require** $D(w)$ and $1 \leq |R_w| \leq 5$
require $0 \leq d(w, r) \leq 5$ for every $r \in R_w$
return $H(v)$

Key: L: genuine callable-local; K_T: complete read associations. D: direct plain non-self assignment; RHS call/known-variable token, excluding strings/keywords and import/definition artifacts. R_w: reaching-definition reads with redefinition-free CFG paths. d: line difference; same-line event order; otherwise CFG instruction distance. H: removing writes disconnects loop-entry/back-edge paths to reads.

Pattern: `tmp = parse(item); emit(tmp)`

One-Way Flag

$b \leftarrow$ initial Boolean value; **require** $J_b(v)$, $U \neq \varnothing$, $M = \varnothing$
for each $w \in U$: **require** $T_b(w)$
return some $w \in U$ satisfies $N_b(w)$

Key: J_b: complete canonical identity/partition, Boolean-literal initialization b (also 0/1). T_b: complete direct write of $\neg b$, or $v \vee e$ from false / $v \wedge e$ from true; symmetric operands allowed. e: Boolean literal/reference/comparison or supported logical composition. N_b: terminal assignment or non-inert operand; structural compatibility, not guard satisfiability.

Pattern: `found = false; ...; found = found || match`

Stepper

$S \leftarrow N$
repeat $S \leftarrow S \cup \{v : P(v),\ D_v \subseteq F \cup S,\ G(v, F, S)\}$
until S stops changing; **return** $v \in S$

Key: N: implicit numeric iterators. P: complete repeated progression; every later write a direct self-update with unary $+/-$ or binary $+, -, \times, /$; exclude fixed $-$ v and proved Walkers. D_v: non-self update dependencies. $G(v, F, S)$: complete supported guards; dependencies in S or fixed inputs with predictable seeds (parameters allowed).

Pattern: `i = 0; while i < limit:  i += stride`

EXTRACTION PSEUDOCODE 2/3

Walker

1 **if** $E(v)$: **accept**
2 **require** $K_W(v)$ and $U \neq \varnothing$
3 **for each** $w \in U$: **require** $V(w)$
4 **accept**

Key: E: implicit element iterator, excluding numeric iterators. K_W: complete navigation and initialization partition. V: previous-position dependence; loop context; access/control use; resolved member, fixed-collection successor index, supported navigation call, or cursor increment/decrement with direct cursor use. An ordinary array index alone is insufficient.

Pattern: `while p:  use(p); p = p.next`

Most Recent Holder

1 **require** $K_R(v)$ and $A_v \neq \varnothing$
2 **reject if** Walker, Follower, or Most Wanted Holder
3 **if** member: **return** $P_R(v)$
4 **return** $L_R(v)$ and every $a \in A_v$ has witness $R(a)$

Key: K_R: complete acquisition/correction coverage of U, $M = \varnothing$, no holder-dependent acquisition guard. A_v: acquisitions (call, collection access, iterator projection). L_R: non-member loop/proved-iterator context. R: read outside own writes via one unambiguous stable-input correction chain; limits: depth 64, nodes 10,000, corrections 5, branches 128. P_R: same-owner member acquisitions in ordinary methods plus an owner use; no cross-call ordering proof.

Pattern: `while more:  current = read(); consume(current)`

Follower

1 **require** $K_F(v)$; **reject if** $X(v)$
2 $s \leftarrow$ unique master shared by every non-initial assignment
3 **require** $A_s(w)$ for every reassignment w
4 **return** $Y_s(v)$

Key: K_F: complete, nonempty reassignment set U_F. A_s: uniquely resolved canonical declaration/member/fixed-index master; RHS s, $s + k$, $k + s$, or $s - k$; k literal/raw Fixed Value. Y_s: every local loop has capture $\rightarrow$ master evolution $\rightarrow$ follower use $\rightarrow$ capture; members require ordered capture/evolution per writing method and same-owner use. X: self-dependence, mutation, follower-controlled write, or follower-dependent master evolution.

Pattern: `prev = current; current = next(current); use(prev)`

Most Wanted Holder

1 $\mathsf{Primary}(v) \leftarrow J_S(v)$ and $S(v)$ and $K_S(v)$
2 **clear primary status if** Stepper, Walker, Gatherer, or Follower
3 **accept if** $\mathsf{Primary}(v)$
4 **otherwise accept if** some primary h satisfies $B_h(v)$

Key: J_S: compatible seeds, nonempty loop-selection epochs, one direction. S: guarded holder–candidate comparison, min/max, or null-seeded first-satisfying candidate with structural exit. K_S: all writes explained (seed/selection/unset fallback), no mutation. B_h: nonempty later writes all coupled to one primary h by a unique guard in one context; nonempty fixed/stepper/walker dependencies, later read, no mutation.

Pattern: `if score > best:  best = score; winner = i`

EXTRACTION PSEUDOCODE 3/3

Gatherer

require $K_G(v)$, $J_G(v)$, $\mathcal{A}_v \neq \varnothing$
reject if Stepper or Walker
for each span $a \in \mathcal{A}_v$: **require** a self-update with D
require all resets, other transforms, and finalizations satisfy T
accept

Key: K_G: admitted numeric non-collection; complete phase coverage of U. J_G: parameter, stable numeric seed, or supported first contribution. $\mathcal{A}$: nonempty loop spans split by resets. SelfUpdate: uses the previous value. D: indexed/member/dereferenced data or a non-fixed/non-stepper dependency; calls through argument data. T: numeric/fixed resets; arithmetic transforms with numeric/fixed/stepper inputs; at most one stable arithmetic finalization per method outside loops.

Pattern: `score *= 5; score += points[item]`

Container

require $Q(v)$ and $H_c(v)$
$E_c \leftarrow$ post-construction collection changes
$P_c \leftarrow$ event IDs consumed by proved permutation spans Π
return $E_c \setminus P_c \neq \varnothing$

Key: Q: verified non-text collection; H_c: source-backed target, excluding synthetic/function bindings. E_c: changes after the identified construction phase: rebinding, updates, add/remove/clear, element/descendant writes, exact-target mutations. Π: proved permutations; P_c: event IDs they consume. Alias-only effects do not establish mutation.

Pattern: `items.append(x); items.sort()`

Organizer

require $Q(v)$, $H_c(v)$, and $J_c(v)$
require $\Pi_v \neq \varnothing$
return $E_c \setminus P_c = \varnothing$

Key: J_c: verified, complete collection initialization. Π: tuple swap; local dataflow swap; exact same-collection swap; known in-place sort/reverse/shuffle/rotate. Manual swaps require reaching-definition/CFG witnesses without transformation; ambiguous paths, indexes, or aliases fail. K_c: complete collection lifecycle.

Pattern: `items.reverse()`

Collection decision and final labels

$C(v) \leftarrow$ None; **require** $Q(v)$ and $H_c(v)$
if $E_c \setminus P_c \neq \varnothing$: $C(v) \leftarrow$ Container
else if $J_c(v)$ and $\Pi_v \neq \varnothing$: $C(v) \leftarrow$ Organizer
else if $J_c(v)$ and $K_c(v)$: $C(v) \leftarrow$ Fixed

Emission: retain raw multilabel results subject to exclusions above. Suppress Temporary if any role other than Fixed Value co-occurs; otherwise Temporary suppresses Fixed Value. Member subjects omit Temporary, Gatherer, and Most Wanted Holder. Raw Fixed Values remain available to dependency checks. Emit identity, labels, metadata, and exact source views; graph/alias/interprocedural limits follow the supplementary implementation.

D DATASET STATISTICS

The consolidated corpus combines the earlier role-enriched dataset with variable records from successful shards of the expanded CodeParrot annotation run. Of 150,000 new candidate files, thirteen shards completed and two failed parsing; failed shards contribute no new annotations. We merge byte-identical sources and identify variables by source hash, kind, name, and declaration coordinates, preserving both releases' annotation IDs. The corpus contains 126,926 source files, 14,624,580 physical lines, and 3,525,530 unique variables across five languages, retaining the earlier Ruby data. Of these, 1,737,758 have an extracted role and 1,787,772 have none; missing labels do not imply semantic absence of a role.

Table 5: Consolidated corpus coverage by partition and language. Role pairs count variable–role memberships; with role is the fraction of variables with at least one extracted role.

Partition	Files	Variables	Role pairs
Train	76,155	2,116,300	1,048,604
Validation	25,386	705,213	349,496
Test	25,385	704,017	349,527
All	126,926	3,525,530	1,747,627

Language	Files	Variables	With role
C	32,492	964,660	42.8%
C++	31,910	913,138	49.3%
JavaScript	29,673	951,288	45.1%
Python	32,798	695,422	64.0%
Ruby	53	1,022	69.6%

Five approximately equal file folds are stratified by role and language with source groups kept intact. For the baselines, three folds train, one validates, and one tests, yielding 2,116,300/705,213/704,017 variables (Table 5). The corpus includes 2,428,645 locals, 865,797 parameters, and 231,088 members.

Table 6: Role support in the consolidated corpus. Percentages use all 3,525,530 variables; test files counts files in each row. Extracted (total) sums variable–role and file–role memberships across rows, not unique variables or files. "No extracted role" is not a prediction target.

Role	Train	Valid.	Test	Total	%	Test files
Fixed value	780,012	259,968	260,007	1,299,987	36.87	24,387
Stepper	22,201	7,403	7,402	37,006	1.05	4,079
Gatherer	2,877	959	958	4,794	0.14	677
Most-recent holder	6,500	2,166	2,168	10,834	0.31	1,300
Most-wanted holder	901	298	300	1,499	0.04	217
One-way flag	4,321	1,439	1,437	7,197	0.20	1,132
Follower	435	144	145	724	0.02	125
Walker	25,013	8,337	8,336	41,686	1.18	4,250
Container	53,230	17,744	17,738	88,712	2.52	7,969
Organizer	448	150	149	747	0.02	127
Temporary	152,666	50,888	50,887	254,441	7.22	15,521
Extracted (total)	1,048,604	349,496	349,527	1,747,627	49.57	59,784
No extracted role	1,073,661	357,596	356,515	1,787,772	50.71	24,348

Fixed value contributes 74.4% of the 1,747,627 memberships, while follower and organizer have 724 and 747 variables (Table 6). There are 9,866 multi-role variables; the largest overlap is fixed value with walker, for 5,665 variables.

Integrity and scope. Exact-source matches, normalized whole-file/function matches, and earlier clone links define 122,458 indivisible source groups; none crosses folds. This grouping is not repository holdout or exhaustive semantic deduplication; 16,784 files have parser errors. We union existential boolean flags across duplicates and recover flags for variables with no extracted role from raw facts. The earlier corpus was used in model development; the new split does not erase that exposure.

E METADATA AND LEXICAL BASELINES

We use the full consolidated corpus (Appendix D), without subsampling. The split has 2,116,300 training, 705,213 validation, and 704,017 test variables. For each of eleven roles, variables lacking that annotation, including those with no extracted role, are negative. We evaluate extractor labels using original identifiers.

Predictors and selection. We use language/kind categories, six general boolean properties, and TF–IDF name character 1–4-grams or local-code token/operator 1–2-grams. Vocabularies retain at most 12,000 name and 30,000 code features, with minimum document frequency 3. Code windows cover up to eight occurrences with 100 source bytes per side; verified target identifiers are masked. The combined baseline concatenates name, code, and boolean features; role-specific metadata and target-derived flags are excluded. For each role, class-balanced L2 logistic regression selects $C \in \{0.1, 1, 10\}$ by validation AP and a threshold by validation F1. Vocabularies and weights are refitted on 2,821,513 train/validation variables; thresholds are unchanged.

Table 7: Macro surface-baseline performance over eleven roles. Brackets: 95% intervals from 500 source-group bootstrap resamples, conditional on fitted models; $\pm$: standard deviation over 500 random draws.

Predictor	Macro AP	Macro F1
Binary majority	0.045	0.000
Most frequent role	0.045	0.049
Always positive	0.045	0.071
Random (uniform)	0.0451 ± 0.0001	0.0598 ± 0.0001
Random (prior-matched)	0.0451 ± 0.0001	0.0451 ± 0.0004
Language + kind	0.100 [0.098, 0.101]	0.112 [0.109, 0.114]
Boolean properties	0.067 [0.065, 0.069]	0.102 [0.099, 0.104]
Variable name	0.142 [0.140, 0.146]	0.213 [0.207, 0.218]
Local code	0.487 [0.475, 0.503]	0.526 [0.515, 0.537]
Combined	0.533 [0.521, 0.547]	0.553 [0.541, 0.564]

Train/validation frequencies define binary majority (no roles) and most-frequent-role (fixed value) references; always positive predicts every role. Uniform random scores yield positive predictions with probability 0.5 or the train/validation role frequency. AP integrates precision over recall increments at distinct score thresholds; F1 is $2\mathrm{TP}/(2\mathrm{TP} + \mathrm{FP} + \mathrm{FN})$ at the frozen threshold. Constant-score AP equals test prevalence; macro averages weight roles equally.

Table 8: Per-role surface predictability on the consolidated corpus. Prior AP is test prevalence; code denotes local code, and combined adds names and boolean properties.

Role	Prior AP	Name AP	Code AP	Combined AP	Combined F1
Fixed value	0.3693	0.594	0.785	0.822	0.752
Stepper	0.0105	0.363	0.649	0.679	0.690
Gatherer	0.0014	0.075	0.423	0.479	0.535
Most-recent holder	0.0031	0.011	0.222	0.223	0.317
Most-wanted holder	0.0004	0.056	0.388	0.458	0.488
One-way flag	0.0020	0.052	0.516	0.665	0.658
Follower	0.0002	0.008	0.099	0.139	0.205
Walker	0.0118	0.114	0.882	0.893	0.840
Container	0.0252	0.115	0.588	0.635	0.620
Organizer	0.0002	0.009	0.272	0.299	0.378
Temporary	0.0723	0.170	0.538	0.569	0.597

Binary majority attains 95.5% accuracy with zero F1. Name, code, and combined macro AP are 0.142, 0.487, and 0.533 (Tables 7–8). These scores measure automatic-label predictability, without independent semantic validation. Probe comparisons require matched subjects, negatives, partitions, and accessible context; these windows can include code after a causal readout.

F SAE DETAILS

Table 9: Sparse autoencoder configurations used for each model.

Model	SAE release	SAE ID	Layers
Gemma 2 2B	`gemma-scope-2b-pt-res-canonical`	`layer_{l}/width_16k/canonical`	6, 13, 17, 22
Gemma 3 4B	`gemma-scope-2-4b-pt-res`	`layer_{l}_width_65k_l0_medium`	9, 17, 22, 29
Gemma 3 12B	`gemma-scope-2-12b-pt-res`	`layer_{l}_width_65k_l0_medium`	12, 24, 31, 41
Llama 3.1 8B	`llama_scope_lxr_32x`	`l{l}r_32x`	8, 16, 21, 27
Qwen3.5 9B	`qwen-scope-3.5-9b-base-w64k-l100`	`layer{l}`	8, 16, 21, 27

G Additional Analyses of Role Representations

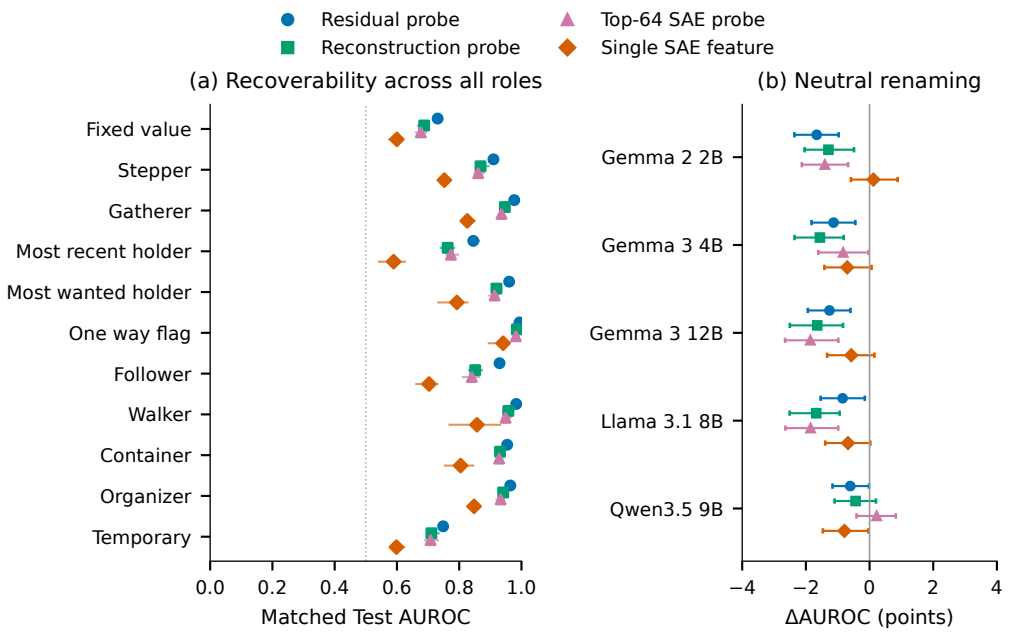


Figure 7: **Residual probes recover more role information than individual SAE features, with modest sensitivity to neutral target names.** (a) Matched-negative Test AUROC for all eleven roles; points average five models and their four prescribed layers, and lines span the five model means. (b) Change in AUROC after renaming both members of 100 positive/negative pairs per role, with fits and thresholds fixed. Intervals are paired 95% source-group bootstrap intervals; negative values indicate lower discrimination after renaming.

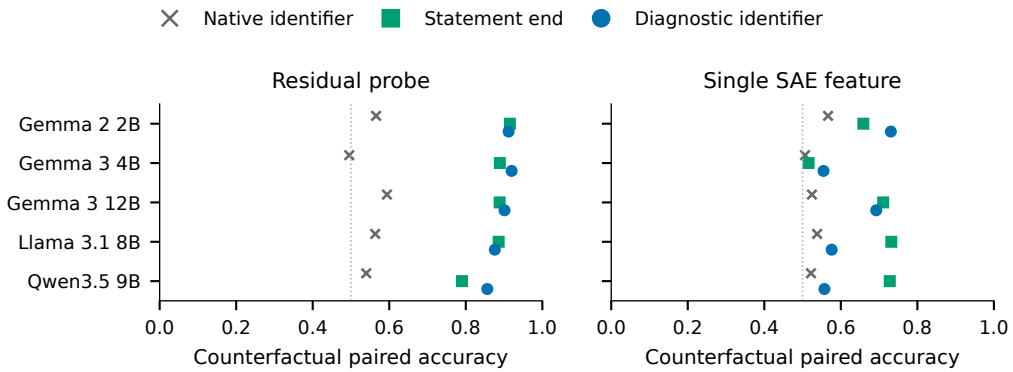


Figure 8: **Later readouts expose counterfactual changes that the native identifier often misses.** Reviewed-counterfactual paired accuracy at the second prescribed layer of each model, averaged over all eleven roles. Predictors are fitted separately at each readout with the same training policy and evaluation pairs; the comparison measures readout-specific decodability.

On matched pairs, residual probes recover automatic labels more accurately than individual SAE features, and surface cues also predict these labels. We next test how this discrimination changes with target names, language, implementation, and the observation position. These analyses use the fitted predictors and diagnostic readout from Section 4, averaging four layers except in the readout comparison. The principal comparisons use matched-negative training; the broader sweep trains separately on matched, neighboring-role, same-context, and hard-mined negatives and evaluates all six strategies in Appendix K. For thresholded results, validation fixes each decision threshold by

maximizing sensitivity minus false-positive rate. We use Gemma Scope for Gemma 2 2B (Lieberum et al., 2024), Gemma Scope 2 for Gemma 3 4B and 12B (McDougall et al., 2025), Llama Scope for Llama 3.1 8B (He et al., 2024), and Qwen-Scope for Qwen3.5 9B (Deng et al., 2026).

Reconstruction and counterfactual controls. To separate reconstruction loss from the restriction to one feature, we also fit ridge probes to SAE reconstructions and to the 64 training-ranked features. Across models, their matched-negative macro AUROC reaches 0.857–0.881 and 0.853–0.878, respectively, compared with 0.906–0.913 for residual probes (Table 2). Thus, combining features closes much of the single-feature gap. We also test 1,698 source-reviewed role-changing counterfactual pairs from 948 source groups, excluding rejected edits and incorrect original-role labels through the LLM review described in Appendix H. Paired accuracy measures how often an original ranks above its edited negative, assigning ties half credit. Residual paired accuracy is 0.813–0.877; reconstruction probes are 4.0–12.3 percentage points lower, with all five paired 95% intervals excluding zero. SAE reconstruction therefore reduces recoverability of these distinctions under the tested linear fits. Intervals use 1,000 source-group bootstrap draws, with equal role and layer weight, conditional on the fixed split and fits. At the same frozen validation thresholds, residual-probe false-positive rates on edited negatives are 0.684–0.713, with balanced accuracy 0.610–0.637. These decision metrics distinguish sensitivity of score rankings from successful rejection of edited negatives.

Target names contribute to discrimination. To isolate target-name sensitivity, we select 100 Test pairs per role without consulting scores and replace every occurrence of the resolved target binding with a neutral identifier. Both positive and negative variables are renamed, preserving their roles and the surrounding syntax; fitted choices are unchanged. Both name conditions use the same model, SAE, and inference configuration. Residual AUROC decreases by 0.6–1.7 percentage points across models (Figure 7b), remaining at 0.878–0.887 after renaming. Single-feature AUROC is lower, at 0.693–0.760, with changes between -0.8 and $+0.1$ points. On these same intervention pairs, name surface features fall from AUROC 0.602 to 0.492, whereas code surface features change from 0.740 to 0.723. The residual probe preserves more discrimination than either surface baseline, although its frozen decisions still change for 10.8% of positives and 12.4% of negatives. Its balanced accuracy decreases from 0.830 to 0.809, distinguishing stable ranking from stable thresholded decisions. The edit preserves other identifiers, comments, and implementation cues, so it measures sensitivity to the target name.

Language and implementation condition the measured signal. We supplement Test coverage toward 50 matched pairs per supported role/language and examine metadata-defined realizations (Figure 9). The fits retain their original multilingual training and validation populations, and evaluation covers C, C++, JavaScript and Python. Gatherer and one-way flag have high residual AUROC across all four languages, while stepper ranges from 0.682 in JavaScript to 0.972 in Python. Ranking and thresholded decisions also differ across implementations: single-feature stepper AUROC is similar for for-loop and other updates (0.728 versus 0.757), but recall drops from 0.895 to 0.624 at the validation thresholds. Within Python, its recall also falls from 0.904 to 0.750, on 23 and 30 positives respectively. Most-wanted-holder recall is high for both guarded replacement and min/max selection, especially for residual probes (0.994 and 0.966). Walker differences are entangled with language in the selected subtypes, and member-expression most-recent holders lack a supported diagnostic readout. These distinctions constrain which apparent subtype effects can be attributed to implementation.

The observation point determines which edits are visible. Causal attention makes the readout position consequential: an assignment's identifier can precede the expression that determines its role. At the second prescribed layer, residual paired accuracy on reviewed edits is 0.495–0.594 at the native identifier and 0.856–0.920 at the diagnostic identifier (Figure 8). Statement-end readouts recover much of this sensitivity, with residual paired accuracy of 0.790–0.915 on the unmodified input. These separately fitted readouts measure position-specific decodability: later positions expose source text that can be unavailable at the native identifier, and the diagnostic reference also changes the input. The resulting predictive associations motivate the intervention experiments in Section 5.

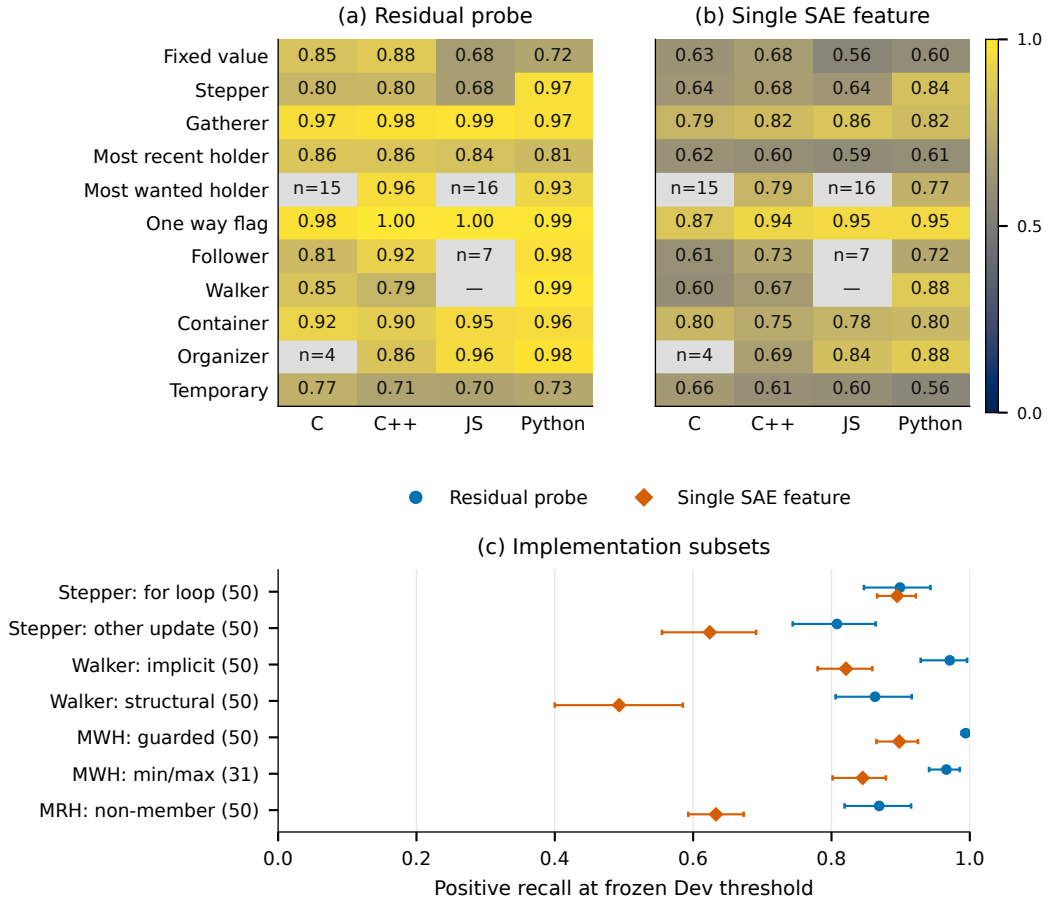


Figure 9: **Language-conditioned discrimination and implementation recall differ across roles and predictors.** (a–b) AUROC averaged over five models and four prescribed layers, targeting 50 matched pairs per role/language. Gray cells show cohorts below 20 pairs or ten positive source groups; a dash marks an unsupported cohort. (c) Recall on metadata-defined positive subsets, with sample counts and 95% source-group bootstrap intervals. Circles denote residual probes and diamonds single SAE features; MWH and MRH abbreviate most-wanted and most-recent holder. Implicit walkers are all Python, whereas 49 of 50 structural walkers are C/C++, so their pooled contrast also changes language.

H EVALUATION POPULATIONS, REVIEW, AND SELECTION

The full annotation corpus, representation benchmark, and generation tasks answer different questions. Table 10 identifies their populations and selection units.

Comparison	Population and split	Selection and outcome
Corpus baselines	3,525,530 variables; three training folds, one validation fold, one Test fold	Automatic annotations; metadata and lexical prediction on the full consolidated corpus.
Representation fits	5,321 training and 9,547 validation pairs; source groups separated from Test	Training fixes normalization and feature ranking; validation chooses regularization, single feature, and threshold.
Matched Test	4,238 positive/negative pairs after exclusions; four languages	Shared examples across predictors; AP and AUROC measure rankings against automatic labels.
Counterfactual Test	1,698 accepted pairs from 948 source groups	Static source review of original and edited behavior; paired ranking and frozen-threshold decisions.
Neutral-name Test	100 positive/negative pairs per role	Both bindings renamed, with fits fixed; paired change in AUROC and binary decisions.
Feature-scale steering	Union of 121 MBPP tasks; role-specific cohorts	Eleven SAE targets each for Gemma 2B/4B and nine for 12B; ten same-layer random features per target.
Normalized steering	Twelve calibration and twelve Test tasks per role; eleven completed model–role cells	Shared probe/SAE dose under a correctness constraint; random residual and SAE controls.

Table 10: Evaluation populations and fitted choices. Pair and target–task counts can reuse bindings or tasks; uncertainty resamples the corresponding source groups or unique tasks. The separate Gemma 12B stepper pilot uses 48 calibration and 144 Test tasks (Appendix P).

Frozen representation cohort. The representation experiments retain an earlier prepared cohort and its source/clone-group assignments; the expanded corpus's full-variable baselines are a separate evaluation. One rotation assigns folds 2/3/4 to training, fold 1 to validation, and fold 0 to Test. Training and validation retain their original multilingual populations. The original matched Test cohort contains 4,245 pairs; excluding Ruby pairs and the two source-role labels rejected by subsequent review leaves 4,238. The common tokenization support uses a 4,096-token ceiling. All methods and model–SAE systems share eligible pairs within a comparison; the four prescribed layers are averaged without Test-based layer selection.

Negative selection. Matched negatives share the positive's partition, programming language, and declaration kind, and lack the target annotation. Retrieval ranks candidates using standardized context length, occurrence count, name length and shape, counts of loops, branches, assignments and calls, parser status, and identifier spelling similarity. Scaling is fitted on training subjects; candidates from the positive's source group are excluded, with limits on negative reuse. Neighboring-role selection additionally requires a specified competing role; same-context selection chooses another resolved binding in the enclosing context. Hard-negative selection uses a logistic classifier on target-masked code, fitted with source-group-held-out folds within training. The broader sweep varies the training-negative policy, retaining matched-negative validation; the main comparisons use matched training throughout.

Predictor fitting. Residual, reconstruction, and top-64 probes use ridge regression with training-only centering and coordinate scaling. The coordinate standard-deviation floor is 10^{-6}, and standardized vectors are divided by the square root of their dimension. Validation AUROC selects the penalty from $\{10^{-6}, 10^{-4}, 10^{-2}, 1, 100\}$. Training AUROC ranks SAE coordinates by

$\max(\mathrm{AUROC}, 1 - \mathrm{AUROC})$ and fixes their predictive polarity. Validation selects one of the top 64 coordinates for the single-feature predictor; the multi-feature probe uses those same 64 candidates. For each predictor, validation selects the threshold maximizing sensitivity minus false-positive rate. The matched name and visible-prefix baselines fit TF–IDF/logistic models on the same training pairs and use validation for regularization and threshold selection. Name features use character 2–5-grams and a 12,000-feature cap; code prefixes use 3–5-grams and a 30,000-feature cap. Both use minimum document frequency two, case-sensitive sublinear term frequencies, and validation-selected $C \in \{0.1, 1, 10\}$. The code-prefix input ends at the diagnostic identifier; the full-corpus lexical windows in Appendix E use a different context policy.

Counterfactual review and exclusions. Counterfactuals are generated from eligible source patterns and reviewed through LLM assessments of the enclosing binding context and exact edit. Review records specify whether the original has the target role, whether the edit removes it, source-level validity, and assumptions about external inputs or calls. The rubric asks reviewers to inspect the binding lifetime and all writes, including mutation and aliasing, and to flag unresolved dependencies or role ambiguity. Source annotations and proposed edits are supplied as fallible evidence; the judgments are not blinded to those proposals. The final review uses GPT-5.6 Terra source reviewers to resolve 333 previously pending cases: 330 edits are accepted and three rejected. One rejected edit preserves the target role; two cases have incorrect original-role labels. Those two source-role labels are excluded from affected test panels; training and validation fits are unchanged. The resulting 1,698 accepted pairs include earlier reviewed cases and the final accepted additions after evaluation exclusions. These static LLM judgments are distinct from independent human adjudication or execution-based verification; the acceptance fraction of the selected queue is not an estimate of corpus-wide annotation accuracy. Review uncertainty and inter-annotator agreement are unmeasured.

Ranking, decisions, and uncertainty. Paired counterfactual accuracy is $\mathbb{E}[\mathbf{1}(s_{\mathrm{original}} > s_{\mathrm{edit}}) + \frac{1}{2}\mathbf{1}(s_{\mathrm{original}} = s_{\mathrm{edit}})]$. AUROC instead compares all positive/negative score combinations within a role–layer cell. Balanced accuracy is the mean of positive recall and negative recall at the frozen threshold. Model summaries average roles and layers equally. Paired 95% intervals use 1,000 joint source-group bootstrap resamples, retaining related examples and methods together; they condition on the split, fits, and review decisions. Layer ranges in Figure 3 and random-feature ranges in Figure 6 describe heterogeneity, not sampling intervals.

Neutral names and readout controls. Test-pair selection does not consult predictor scores. Neutral identifiers are derived from a role-independent binding hash; edits replace all annotated occurrences of the resolved binding, with source hashes, syntax shape, and non-target leaf bytes checked. Original and renamed comparisons use matched execution settings; stored scores are reused only after overlap checks. Readout controls fit separate predictors at the native identifier, statement end, and inserted identifier at each model's second prescribed layer. They compare position-specific decodability, with differences in accessible context and, for the inserted reference, input text.

Steering selection and provenance. The representative features in Figure 6 are a descriptive selection from the larger feature-scale experiment. The archived selection does not establish that these five representatives were chosen before intervention outcomes were inspected; aggregate conclusions therefore use all selected targets within each protocol. Addition strength is calibrated on annotation induction for the feature-scale experiments, while the normalized experiments calibrate a joint induction-and-passing objective subject to a correctness constraint. Appendices M and P report their different perturbation scopes, task overlap, denominators, and unresolved historical scale provenance.

I BEHAVIORAL ROLE RECOGNITION AND PROMPT SENSITIVITY

We evaluate Llama 3.2 1B/3B and 3.1 8B, Gemma 3 1B/4B/12B/27B, Qwen3 1.7B/4B/8B/14B Base, and Qwen3.5 9B Base. These twelve pretrained checkpoints share the protocol but vary in size, architecture, and training.

Evaluation and scoring. The same 9,553 matched positive/negative development pairs cover eleven roles and five languages: 17,770 distinct role/variable queries from 16,837 variables and 6,996 source groups. Labels are automatic annotations; a negative lacks the target annotation. Prompts include the role definition, full enclosing code with comments, and target identifier and location, asking for a judgment from initialization, updates, and use while allowing other roles to apply. Each role has one fixed training-only demonstration set (seed 17), with nested, class-balanced prefixes of 0, 2, 4, or 8 examples shared across models. Inference uses BF16 without quantization, chat templates, or source truncation. At the final `Answer:` position, $s = \log P(\texttt{Yes}) - \log P(\texttt{No})$ predicts positive when $s \geq 0$; average precision (AP) ranks the unrounded scores. We restore original pair multiplicities and average roles equally: constant predictions attain 50% accuracy and constant scores attain 50% AP.

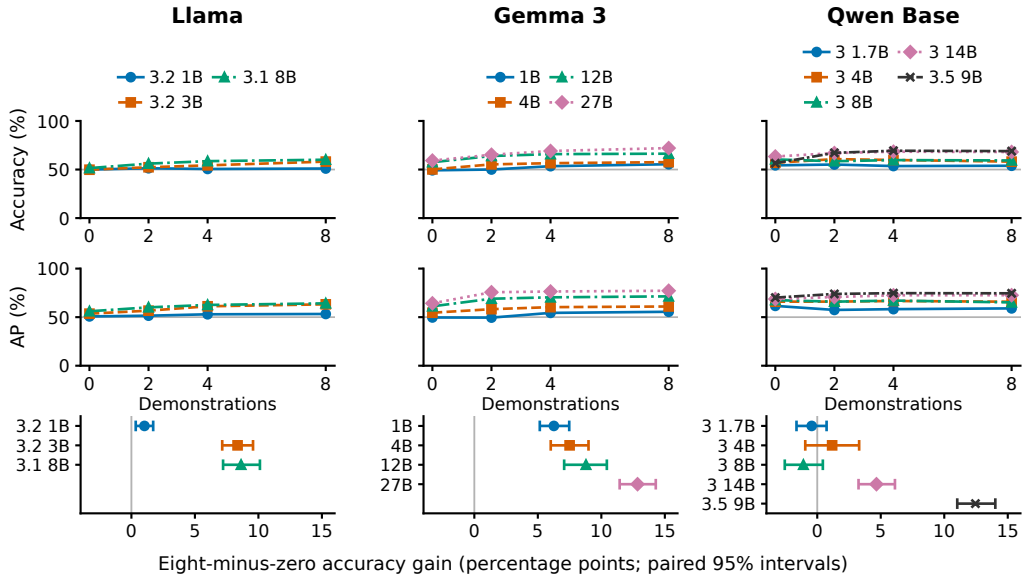


Figure 10: Demonstration gains differ across checkpoints. Top and middle: role-macro accuracy and AP on identical development pairs; gray lines mark 50%. Bottom: eight-minus-zero accuracy gains with paired 95% source-group bootstrap intervals (2,000 draws), conditional on the cohort, pair weights, and one demonstration set. Intervals are exploratory and unadjusted for multiple comparisons. Model labels specify version and nominal size within each family.

Recognition across checkpoints. Gemma 3 27B has the highest eight-shot macro accuracy/AP (72.1/77.2%; Figure 10); its accuracy exceeds Gemma 3 12B by 5.8 points (paired 95% interval: 4.6–6.9). Qwen3 14B reaches 68.1/72.6%, exceeding Qwen3 8B accuracy by 8.7 points (7.5–9.9). Eight-shot accuracy rises with tested size within each family, but demonstration gains vary: the eight-minus-zero accuracy intervals include zero for Qwen3 1.7B/4B/8B, and their AP does not improve. Llama 3.2 1B gains 1.0 accuracy point (0.3–1.7), with accuracy near chance.

Taxonomy-free descriptions. Prompts request at most three sentences on initialization, updates, and use, without role names or examples. Training-only format calibration reduces descriptions reaching the generation cap from 27/66 to 6/66 across three checkpoints; we fix this format for all models. Cap rates range from 0% to 78.2% across the expanded model set. We assess fidelity qualitatively: Gemma 3 27B invents a prior `None` initialization for `exit_code`, while Qwen3 14B identifies the shown assignment.

I.1 Role differences and free generation

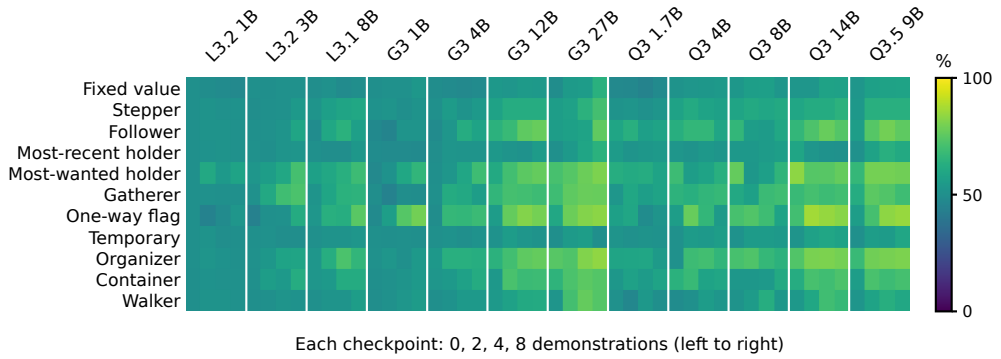


Figure 11: Role accuracy varies within these aggregate trends. Each checkpoint has four columns for 0/2/4/8 demonstrations: all 528 combinations, with original pair weights and a shared scale. L/G/Q denote Llama/Gemma/Qwen; numerals specify version and size.

Role and answer dependence. Fixed value, most-recent holder, and temporary are weaker than one-way flag and most-wanted holder (Figure 11). Greedy generation evaluates 100 distinct development cases per role (50 positive, 50 negative), shared across models: 1,100 cases and 39,600 outputs across descriptions and zero/eight-shot answers. The fixed seeded panel includes the original six cases per role; cases receive equal weight and share source groups. Leading Yes/No answers are valid; invalid outputs count as incorrect, with a 512-new-token budget. Three Llama 3.1 8B shards use A100/GB10 GPUs; the others use H200s, with matching software.

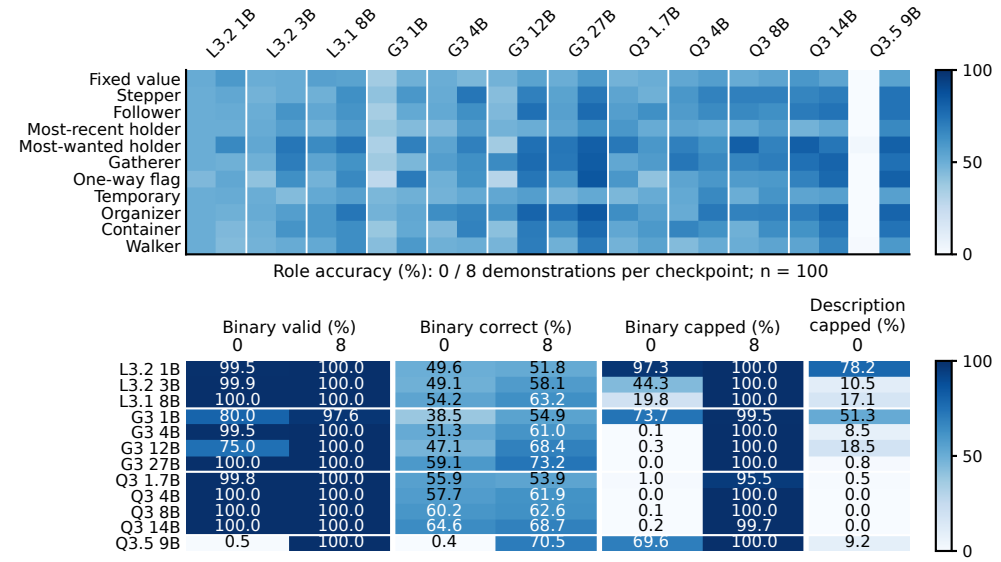


Figure 12: Valid answer beginnings coexist with frequent budget exhaustion. Top: all 264 role/model/shot accuracies, each on 100 cases. Bottom: aggregate percentages over 1,100 cases per checkpoint; columns give demonstration counts. Invalid answers count as errors; description caps measure stopping behavior, not semantic accuracy.

Eight-shot answer validity is 97.6–100%, yet 95.5–100% of outputs reach the cap (Figure 12). Gemma 3 27B reaches 73.2% free-answer accuracy. Qwen3 Base checkpoints yield 99.8–100% valid zero-shot starts, compared with 6/1,100 for Qwen3.5 9B despite its 70.1% zero-shot AP under constrained scoring.

J Extended related work

Table 11: Extended comparison of work on code-language-model behavior and representations. Groups follow Section 6, with a final group for explanation methods and human comparison; studies are placed by their primary emphasis. The analysis column distinguishes input perturbations, representation readouts, and activation interventions. AST: abstract syntax tree; SAE: sparse autoencoder. †: cited as a preprint or technical note.

Study	Analysis and target	Key finding
Behavioral robustness and semantic priors		
Yang et al. (2022)	Input edits: semantics-preserving names	Plausible renaming induces failures; adversarial training improves robustness.
Gao et al. (2023)	Input counterfactuals: names	Names provide semantics and shortcuts; counterfactual inference improves robustness.
Wang et al. (2023)	Input edits: ReCode	Generation degrades under identifier, syntax, formatting, and docstring perturbations.
Ahmed et al. (2023)†	Input edits: masked code	Masked-operator predictions reflect code edits beyond frequency effects.
Hooda et al. (2024)	Input counterfactuals: predicates	Counterfactual code edits expose inconsistent handling of control/data flow and names.
Li et al. (2024)	Token attribution: project bias	Regularization improves cross-project generalization and adversarial robustness.
Wang et al. (2024)	Input edits: random/misleading names	Both harm code search and clone detection; misleading names are often worse.
Orvalho & Kwiatkowska (2025)†	Input edits: equivalent programs	Mutations degrade output prediction; correct answers can accompany flawed explanations.
Liang et al. (2026)†	Input edits: test-specified behavior	Test-specified rule switching is incomplete, even for models that implement both rules when explicitly instructed.
Programming concepts in internal representations		
Karmakar & Robbes (2021)	Probes: code properties	English-pretrained BERT is competitive with code-pretrained models on some probes.
Karmakar & Robbes (2024)	Probes: INSPECT	Fifteen probes; structure-aware models often score higher, with task-dependent rankings.
Wan et al. (2022)	Probes/attention: program structure	Hidden states and attention encode recoverable syntactic relations.
Hernández López et al. (2022)	Structural probes: ASTs	Full ASTs recoverable from low-dimensional subspaces; middle layers most informative.
Troshin & Chirkova (2022)	Probes: syntax, names, semantics	Syntax and naming properties are decodable; semantic equivalence is harder.
Ma et al. (2024)	Probes: syntax and flow graphs	Syntax broadly recoverable; control/data dependencies vary across models.
Sharma et al. (2022)	Attention/pooling: identifiers	Attention favors identifiers and separators; identifier pooling improves clone detection.
Haji Mohammadkhani et al. (2023)	Attention: downstream code tasks	Token-category attention varies across tasks; models can fail on simple programs.
Anand et al. (2024)	Probes: token relations	Syntax–identifier links are poorly recovered; larger tested models often trail smaller ones.
Jin & Rinard (2024)	Probes: synthetic execution states	Intermediate states are decodable without supervision on execution traces.
Sharma et al. (2024)	Neuron selection: token concepts	Small neuron subsets preserve probe accuracy; selected neurons align with code concepts.
Haider et al. (2026)	MLP-key inspection/masking	Masking API-associated keys reduces targeted predictions, with smaller effects on other tokens.
Sharma et al. (2025)†	Clustering: CoCoA concepts	Lexical, syntactic, and semantic clusters; lexical/subword structure persists in deeper layers.

Continued on next page

Table 11 (continued)

Study	Analysis and target	Key finding
Shared representations across programming and natural languages		
Kargaran et al. (2025)	Decoding/ablation: shared neurons	Intermediate states and neurons share programming-language and English information.
North et al. (2025)	Probes/attention: variable types	Shared attention heads convey type information across three programming languages.
Yin et al. (2026)	Probes, anonymization, neuron ablation	Language-specific neurons coexist with more invariant middle-layer representations.
Naganna et al. (2026)†	Patching: arithmetic across formats	Shared neurons across symbols, text, and Python; cross-format patches rescue selected errors.
Causal and mechanistic analyses		
Heimersheim & Janiak (2023)†	Circuit tracing: Python docstrings	Compositional argument-copying circuit explains part of a small attention-only model's behavior.
Conmy et al. (2023)	Automated edge selection: ACDC	Recovers task-relevant subgraphs, including the docstring-copying setting.
Templeton et al. (2024)†	SAE clamping: code features	Feature clamping induces insecure code, backdoors, or false bug reports in selected examples.
Lucchetti & Guha (2025)	Activation steering: type prediction	Repairs induced failures; middle-layer effects and Python–TypeScript transfer vary by edit.
Rai et al. (2025)	Circuit analysis: parentheses	Faulty components override sound computations on synthetic parentheses tasks.
Tahimic & Cheng (2025)†	SAE readouts/steering: correctness	Separate features predict errors and steer generation; steering can repair failures or corrupt correct outputs.
He et al. (2026)†	Attribution graphs/intervention	Graphs predict correctness; a binary-search case demonstrates causal repair.
Abdelsalam et al. (2026)†	Patching: misleading names/comments	Traces cue-induced errors through selected residual-stream states in curated conflict cases.
Chen et al. (2026)†	Probes/context-stripped decoding	In short Python tasks, probe recovery can precede context-stripped decoding; later layers can lose correct answers.
Huang et al. (2026)	Probes/training/steering: security	Intermediate-layer vulnerability signals support improved secure generation with combined interventions.
Explanation methods and human comparison		
Chen et al. (2024)	Input perturbations: DeciX	Explanation accuracy, compactness, and stability improve over evaluated baselines.
Paltenghi et al. (2024)	Attention/gaze: code exploration	Processed follow-up attention captures aspects of developers' code-reading transitions.

K SELECTING AND GENERATING NEGATIVE EXAMPLES

Original positive

x advances by a fixed increment.

Positive variable role: stepper

```
def integrate(self, p, q, step):
    sum = 0.0
    x = p
    while x <= q:
        dy = self.blancmange_curve(x) - self.circle(x)
        sum += step * dy
        x += step
+       print(x)
    return sum
```

a) Matched properties

Match language and binding kind; minimize differences in surface features.

Negative variable role: container

```
x, y = [], []
for i in range(self.start, self.stop):
    x.append(i)
+   print(x)
    y.append(self.dict.get(i, 0))
return [x, y]
```

b) Neighboring roles

Select a walker to contrast iteration over data with fixed-increment progression.

Negative variable role: walker

```
def setUp(self):
... ...
    for x,y in self.data:
+       print(x)
        self.db.add(x,y)
```

c) Same context

Query sum (gatherer) in the same source function.

Negative variable role: gatherer

```
def integrate(self, p, q, step):
    sum = 0.0
    x = p
    while x <= q:
        dy = self.blancmange_curve(x) - self.circle(x)
        sum += step * dy
+       print(sum)
        x += step
    return sum
```

d) Counterfactual edits

Replace x's fixed increment with an increment computed from dy.

Counterfactual negative role: gatherer

```
x = p
while x <= q:
... ...
Δ       x += step * (1 +
↪   abs(dy))
+       print(x)
```

Counterfactual negative role: gatherer

```
x = p
while x <= q:
... ...
Δ       x += step + abs(dy)
+       print(x)
```

e) Hard-negative mining

Select negatives with high out-of-fold stepper probabilities.

Negative variable role: fixed value

```
pix = self.image.load()
+ print(pix)
for y in range(0, self.height):
    for x in range(0, self.width):
        self.ind[pix[x, y][0]] = self.ind[pix[x, y][0]] + 1
```

f) Misleading name

Rename x to total using a gatherer donor from a different context.

Renamed positive variable role: stepper

```
def integrate(self, p, q, step):
    sum = 0.0
Δ   total = p
Δ   while total <= q:
Δ       dy = self.blancmange_curve(total) - self.circle(total)
        sum += step * dy
Δ       total += step
+       print(total)
    return sum
```

Figure 13: **Strategies for selecting or creating negative examples, illustrated for the stepper role.** Bold gold marks targets; headers give their roles. Selected negatives lack the target annotation, counterfactuals change the role, and renaming preserves it. Illustrative `print` readouts (+) immediately follow the statement or loop header containing the occurrence of interest, making that text available to causal attention. Δ marks edits, ellipses omit lines, and arrows mark wrapping. Examples come from annotated Python training data.

Original positive

total accumulates successive values.

Positive variable role: gatherer

```
def benchmark(f):
    runs = 100
    total = 0.0
    for i in range(runs):
        result = benchmark_once(f)
        sys.stderr.write('Run {0}: {1}\n'.format(i, result))
        total += result
+       print(total)
    return total / runs
```

a) Matched properties

Match language and binding kind; minimize differences in surface features.

Negative variable role: container

```
staff_lines = []
for dummy in self._tuning:
    staff_lines.append([])
+   print(staff_lines)
self._staff_lines = tuple(staff_lines)
```

b) Neighboring roles

Select a most-wanted holder to contrast extremum tracking with accumulation.

Negative variable role: most-wanted holder

```
best_dist = euc(row, self.x_train[0])
...
for i in range(1, len(self.x_train)):
    dist = euc(row, self.x_train[i])
    if dist < best_dist:
        best_dist = dist
+       print(best_dist)
```

c) Same context

Query i (stepper) in the same source function.

Negative variable role: stepper

```
def benchmark(f):
    runs = 100
    total = 0.0
    for i in range(runs):
+       print(i)
        result = benchmark_once(f)
        sys.stderr.write('Run {0}: {1}\n'.format(i, result))
        total += result
    return total / runs
```

d) Counterfactual edits

Replace accumulation with direct assignment or a running maximum.

Counterfactual negative role: most-recent holder

```
total = 0.0
for i in range(runs):
...
Δ       total = result
+       print(total)
return total / runs
```

Counterfactual negative role: most-wanted holder

```
total = 0.0
for i in range(runs):
...
Δ       total = max(total,
↪ result)
+       print(total)
return total / runs
```

e) Hard-negative mining

Select negatives with high out-of-fold gatherer probabilities.

Negative variable role: stepper

```
def get_content_depth(content):
    depth = 0
    for parent in iter_parents(content):
        depth += 1
+       print(depth)
    return depth
```

f) Misleading name

Rename total to idx using a stepper donor from a different context.

Renamed positive variable role: gatherer

```
def benchmark(f):
    runs = 100
Δ   idx = 0.0
    for i in range(runs):
        result = benchmark_once(f)
        sys.stderr.write('Run {0}: {1}\n'.format(i, result))
Δ       idx += result
+       print(idx)
Δ   return idx / runs
```

Figure 14: **Strategies for selecting or creating negative examples, illustrated for the gatherer role.** The six strategies vary the queried binding, context, update, or name. Renaming preserves the role and all non-target code; donor and recipient role sets are disjoint and their source groups differ. The miner uses target-masked code and logistic regression in three source-group-held-out folds within training. Added `print` readouts, highlighting and edit markers follow Figure 13.

```python
best = None
for x in values:
    if valid(x):
        if best is None or
     score(x) > score(best):
            best = x
```

(a) Guarded MWH

```java
int min_val = a[0];

for (int i = 1; i < n; ++i)
     {
    if (a[i] < min_val)
        min_val = a[i];
}
```

(b) Min/max MWH

```javascript
let bestUser = null;
let bestScore = -Infinity;

for (const user of users) {
    const s = rank(user);
    if (s > bestScore) {
        bestScore = s;
        bestUser = user;
    }
}
```

(c) MWH with companion

Figure 15: **Different realizations of the most-wanted holder role.** A guarded MWH updates only when an additional eligibility condition holds; a min/max MWH retains the extremal value observed so far; an MWH with a companion variable stores the selected object while another variable tracks the selection criterion.

```java
boolean found = false;

for (Item x : items) {
    if (matches(x)) {
        found = true;
        break;
    }
}
```

(a) Direct terminal assignment

```python
valid = False

for x in items:
    valid = valid or check(x)
```

(b) Monotone update

Figure 16: **Different realizations of the one-way flag role.** The direct form uses a terminal assignment, typically inside a branch. The monotone form repeatedly combines the previous state with a new condition. Both allow state changes in only one direction.

L DIFFERENT REALIZATIONS OF VARIABLE ROLES

```for
for (int i = 0; i < n; ++i) {
    process(data[i]);
}
```

(a) Numeric loop iterator

```count
count = 0
for elt in arr:
    count += 1
```

(b) Predictive accumulator

Figure 17: **Different realizations of the stepper role.** A numeric iterator follows a regular progression controlling loop iteration. A predictive accumulator advances by a regular step to represent the position or state expected at the next iteration.

```for
for item in records:
    consume(item)
```

(a) Implicit iterator

```node
Node *node = head;

while (node != NULL) {
    visit(node);
    node = node->next;
}
```

(b) Structural iterator

Figure 18: **Different realizations of the walker role.** An implicit iterator takes successive elements exposed by the loop construct. A structural iterator follows explicit links in the data structure.

```let
let current = start;
let previous = null;

while (current) {
    current = current.next;
    previous = current;
}
```

(a) Assigned after master change

```node
Node* current = head;
Node* previous = nullptr;

while (current) {
    previous = current;
    current = current->next;
}
```

(b) Assigned before master change

Figure 19: **Different realizations of the follower role.** A follower derives its value from a changing master variable. Assignment order determines whether it captures the master's value before or after a transition.

```last_member
last_member = nil

group.each do |member|
  last_member = member
end
```

(a) Member holder

```latest
latest = None

for event in events:
    if event.accepted:
        latest = event
```

(b) Loop-updated holder

Figure 20: **Different realizations of the most-recent holder role.** A member holder receives successive collection elements. A loop-updated holder records the latest value satisfying an update condition. The stored value is replaced at every element or only at selected events, respectively.

M MATCHED PROBE AND SAE STEERING

We compare probe and SAE directions under the same intervention budget, alongside random residual and random SAE directions. The exploratory panel covers stepper, gatherer, most-wanted holder, and walker in Gemma 2 2B (layer 13), Gemma 3 12B (layer 24), and Llama 3.1 8B (layer 16). Each role has twelve calibration tasks and twelve Test tasks, balanced by role presence in the reference solution and shared across models. These are intact batches selected from the original MBPP cohort without consulting generation outcomes. Eleven model–role cells pass the generation and evaluation checks; Llama walker is excluded because its zero-dose output differs from the unsteered baseline.

A common budget and calibration criterion. We add each normalized direction at generated-token residual positions in one layer, with magnitudes of 1%, 3%, or 10% of the median calibration residual norm. The zero-strength option is also eligible. Probe and SAE share the strength maximizing mean calibration induction accompanied by passing tests, subject to each losing at most five percentage points of correctness; ties favor the smaller strength. The selected strength is fixed for Test and applied to both random controls. With twelve calibration tasks, the correctness constraint permits no net loss of a passing program. All conditions retain the same four-task batches, prompts, and greedy 256-token budget, and the zero-dose control must reproduce baseline tokens exactly. The completed comparison contains 3,696 evaluated generations.

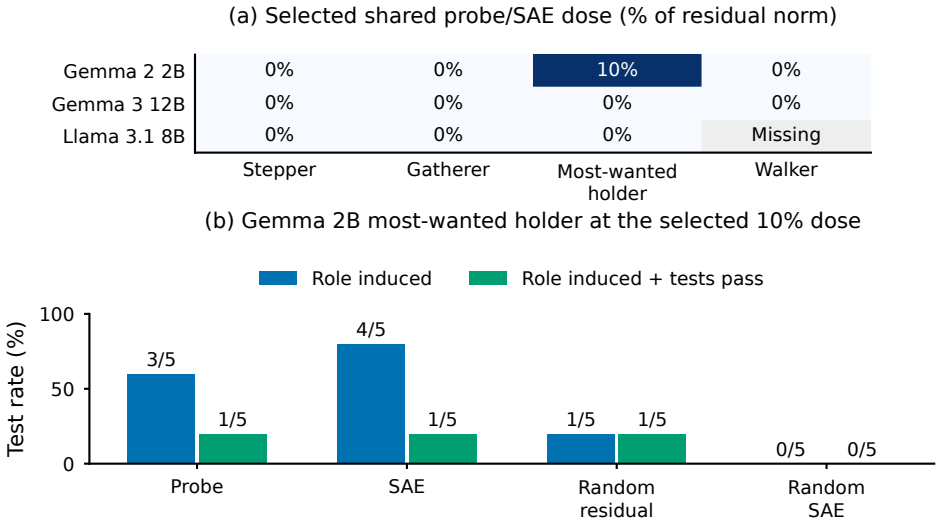


Figure 21: Correctness-constrained calibration usually selects no intervention. (a) Ten of eleven completed cells select zero; the gray cell is unavailable after a failed zero-dose check. (b) In the sole nonzero cell, probe and SAE directions induce most-wanted holder on three and four of five eligible Test tasks, respectively, but each produces only one induction that passes the supplied tests. Bars include random controls on the same tasks; fractions show the small conditional denominators.

Role induction often accompanies test failure. Gemma 2 2B most-wanted holder selects the 10% dose (Figure 21). Its probe and SAE induction rates are 3/5 and 4/5, compared with 1/5 for the random residual direction and 0/5 for the random SAE direction. Induction accompanied by passing tests is 1/5 for the probe, SAE, and random residual direction, and 0/5 for the random SAE direction. Overall Test correctness remains 4/12 for both target directions and falls to 3/12 for both controls. The remaining ten cells select zero because nonzero strengths provide no better feasible calibration objective. This result distinguishes the ability to alter role annotations from reliable control under a correctness constraint. These estimates concern positive steering at one layer in small, overlapping task cohorts conditioned on baseline role presence.

N ACTIVATION CONTEXTS OF ROLE-ASSOCIATED SAE FEATURES

The displays below cover all eleven roles with validation-selected single-feature predictors from Gemma 3 4B layers 9 and 17. Role colors and symbols follow Figure 2. Each row gives a Neuronpedia description, corpus code (left and "Also"), and public prose (right). Highlight intensity is linear from zero to the displayed bound, fixed per latent; compare magnitudes within a latent. "Lower predicts role" identifies negative polarity in the original probe selection, so strong highlighting need not occur on the target variable.

Stepper · i — 4B L17 #4648 · 0–800

"loop variables and iterators" — Higher predicts role

```
while i < len(L):↵
    result.append(L[i])↵
    i = i + 3
```

Now for the i'th color we need to count

Also: `for i in range(20):`

Walker · index — 4B L17 #5142 · 0–900

"code structures and loops" — Higher predicts role

```
index = goal↵
while index is not None:↵
    result.append(index)↵
    index = path[index]
```

each of which was several MBs in size.

Also: `for tag in tags:`

Follower · old_time — 4B L9 #34976 · 0–200

"assignment with value" — Lower predicts role

```
while True:↵
    new_time = time.time()↵
    delta_time = new_time - old_time↵
    old_time = new_time
```

your request must mention that condition

Also: `temp = F[1]`

Fixed value · value — 4B L17 #6 · 0–900

"function or method returns" — Lower predicts role

```
v = float(value)↵
d = float(divisor)↵
↵
return v/d
```

returns the newly created resource.

Also: `return F.relu(h)`

Most-recent holder · error — 4B L9 #34976 · 0–200

"assignment with value" — Higher predicts role

```
while error > emax and i < imax:↵
    error = self.step()↵
    i = i + 1
```

your request must mention that condition

Also: `tests = loader.loadTestsFromTestCase(test_class)`

Figure 22: **Selected features for progression, traversal, retention, and fixed values.** Row titles name the role and corpus variable; 4B denotes Gemma 3 4B. Follower and most-recent holder share feature 34976 with opposite predictive signs. Return marks denote newline tokens, and "Also" introduces a separate corpus context.

All ten distinct latents were measured on 41 hand-selected training snippets. Code crops retain their standalone-input activations; public windows retain their original Neuronpedia contexts. These examples illustrate activity, not prevalence.

DESCRIPTIONS AND ACTIVATION CONTEXTS (CONTINUED)

Most-wanted holder · min — 4B L17 #3404 · 0–900

"highest or lowest extremes"

Higher predicts role

```
min = 1e20↵
for x in arr:↵
    if x < min:↵
        min = x
```

Also: `result = max(result, last);`

to find the lowest fares all on one site.

Gatherer · result — 4B L17 #1377 · 0–900

"initialization to zero"

Higher predicts role

```
result = 0↵
for k in d:↵
    result += d[k]
```

Also: `length += f.repeat`

we add 1 to i, which will be 2.

One-way flag · ret — 4B L17 #11436 · 0–1000

"boolean data types"

Higher predicts role

```
ret = False↵
for p in processList:↵
    if p.is_alive():↵
        ret = True
```

Also: `result = True`

A boolean indicating whether a downloaded file still exists

Container · frequencies — 4B L9 #10912 · 0–200

"list operations"

Higher predicts role

```
frequencies = defaultdict(int)↵
for letter in sentence:↵
    frequencies[letter] += 1
```

Also: `result.append(eq)`

the list of words on the right isn't really "user-friendly"

Organizer · methods — 4B L17 #6996 · 0–800

"lists and arrays"

Higher predicts role

```
methods = list(http_methods)↵
methods.sort()
```

Also: `filters.reverse()`

the Text objects are in the right order in the Space List array.

Temporary · a — 4B L17 #3423 · 0–800

"training data, passwords, dates, books"

Higher predicts role

```
a = y * y↵
b = x * x↵
c = math.sqrt(a + b)
```

Also: `delta = now - date`

Combine the beans with the garlic, the chopped cilantro

Figure 23: **Extrema, accumulation, flags, collections, and intermediate computations.** The displays continue Figure 22 with the same conventions and per-latent scales. Descriptions can name broad operations or disparate contexts, rather than the variable role itself.

Matched and reviewed-counterfactual AUROCs are 0.848/0.474 for most-wanted holder, 0.949/0.354 for one-way flag, and 0.764/0.536 for stepper. Contextual activity alone does not establish role specificity.

O QUALITATIVE OUTCOMES OF SAE STEERING

The larger Gemma 2 2B and Gemma 3 4B experiments from Section 5 also permit inspection of complete generated programs. Across 292 evaluated target–task pairs, eight failing-to-passing transitions correspond to six distinct model–task pairs. They include most-frequent-item search, counting sort, largest row sum, cuboid volume, product divided by length, and maximum subarray sum. We show selected repairs and failures alongside an unfiltered seeded random sample in Figure 5. All displayed test outcomes were reproduced on the complete programs using the three supplied assertions.

The selected repairs can reveal useful changes, such as replacing a stub with an explicit search, without establishing that the targeted role caused the improvement. In the broader gallery, cuboid and product repairs pass all three tests while retaining a zero target-role annotation. Conversely, an unchanged correct program may already implement the role semantically even when its original annotation is absent. The six random cases drawn across the full gallery all fail the supplied tests; the draw was not repeated to obtain favorable examples. These observations motivate reading role annotations, complete code, and correctness together.

P CONSOLIDATED QUANTITATIVE ANALYSIS OF CAUSAL INTERVENTIONS

We connect the selected-feature results in Figure 6 and the structural changes in Table 3 to all tested SAE features and the normalized probe/SAE experiments. The two displays summarize complementary measurements from the same Gemma 2 2B and Gemma 3 4B causal runs. The analysis adds the completed Gemma 3 12B feature-scale run, a five-fit Gemma 3 12B stepper pilot, and the eleven model–role cells in Appendix M.

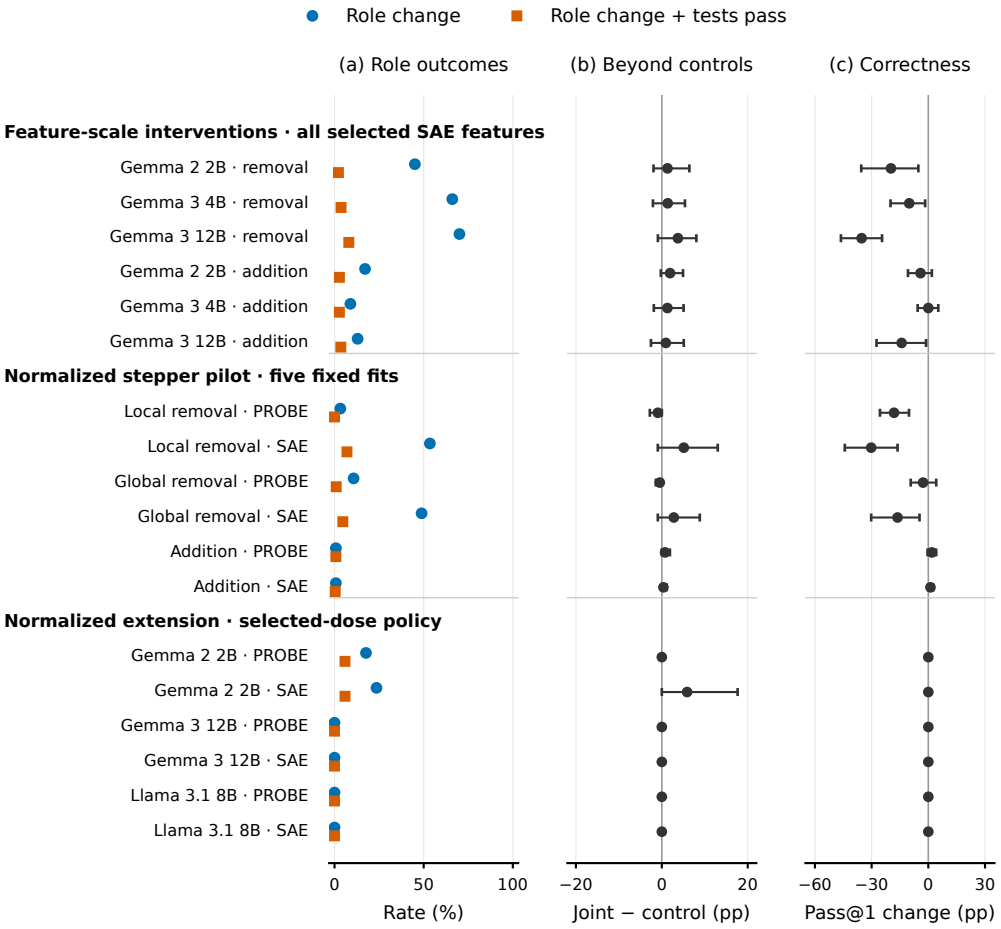


Figure 24: **Annotation changes substantially exceed changes accompanied by passing tests.** (a) Role disappearance for removal and induction for addition, with the joint passing outcome on the same baseline-defined cohort. (b) Joint outcome minus the matched random-control mean: random residual directions for probes and random SAE directions for SAE targets. (c) Intervention-minus-baseline correctness on that cohort. Rates weight target–task enrollments within each protocol; 95% intervals resample unique tasks 4,000 times, retaining all associated features, fits, and controls together. Normalized addition averages include selected zero doses; the stepper pilot uses Gemma 3 12B. The three protocol groups use different cohorts and perturbation scales.

Across all historical features, removal changes 41/91, 72/109, and 131/187 target–task outcomes for Gemma 2 2B, Gemma 3 4B, and Gemma 3 12B, respectively. Only 2/91, 4/109, and 15/187 also pass the supplied tests; requiring both outputs to pass leaves 0, 3, and 13 outcomes. The joint outcome exceeds matched random controls by only 1.3, 1.4, and 3.7 percentage points for removal, and 1.9, 1.3, and 0.9 points for addition; all six corresponding intervals include zero. These enrollment counts include repeated tasks across features and are not independent sample sizes. The removal cohorts contain 32, 32, and 63 unique tasks, respectively; reference-positive addition uses 53, 49, and 45 unique Test tasks.

P.1 Comparison units, calibration, and structural survival

Protocol-specific comparisons. We test eleven SAE directions in Gemma 2 2B and Gemma 3 4B and nine in Gemma 3 12B, covering five roles. They share a union of 121 MBPP tasks with role-specific enrollment; removal uses both splits, while induction uses Test only. Six positive multipliers are calibrated for annotation induction, with ten same-layer random SAE controls per target. The baseline passing counts are 33/121, 14/121, and 67/121. The archived feature scales are not normalized across models, and twenty of the shared package's twenty-two scales differ from the corresponding selection artifact's training-positive maximum. They are treated as executed settings with unresolved scale provenance. Consequently, these results describe the recorded interventions without attributing cross-model differences to model size or a common dose.

The normalized stepper pilot uses Gemma 3 12B layer 31, five fitted probes, and the same SAE feature in every fit. It has 48 calibration and 144 Test tasks, including 43 eligible removal and 53 induction tasks. Four fits select zero and one selects a 1% residual-norm dose under the correctness constraint. In that nonzero fit, probe and SAE directions each induce stepper on 2/53 tasks; two probe outputs and one SAE output pass, including one probe change that preserves baseline correctness. Repeated SAE outcomes and overlapping probe training sets are not independent replications. Local projection affects one generated-token layer; global projection also alters residual writers and prompt processing. The smaller extension in Appendix M instead covers four roles in twelve-task splits and selects zero in ten of eleven completed cells.

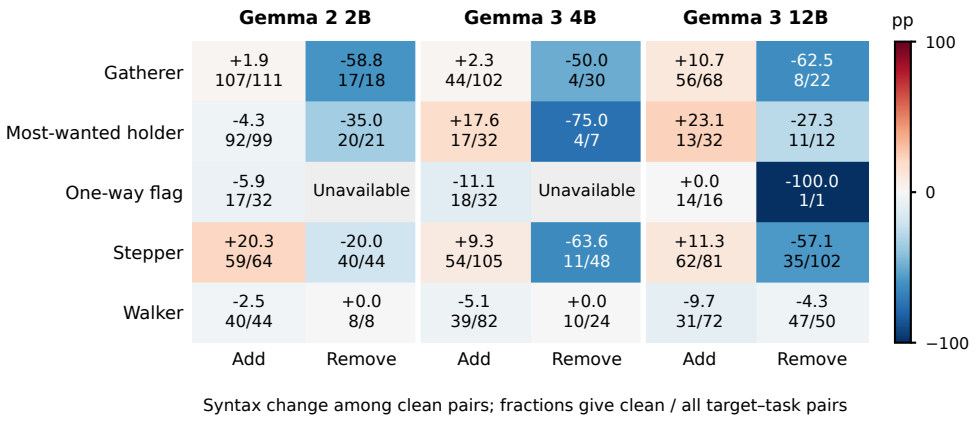


Figure 25: **Large structural shifts can describe a small surviving subset.** Values reproduce Table 3 and extend it to Gemma 12B: syntax-frequency changes in percentage points among pairs with two clean Tree-sitter parses. Fractions show retained clean pairs over all target–task pairs. Markers follow Table 3; addition combines Test sufficiency and reference-negative cohorts. Gray cells have no eligible removal cohort.

Syntax and correctness condition different subsets. For Gemma 3 4B removal, gatherer syntax uses 4/30 pairs from two distinct tasks, and stepper syntax uses 11/48 pairs from five tasks. The reported shifts therefore describe parseable survivors, not all interventions. Among the three clean stepper pairs where both outputs pass, the `range`-call frequency does not change.

Scope of inference. Within each protocol, target and control outcomes use the same eligible tasks and retained denominator. The joint metric requires the intervention output to pass; preserving an already correct program additionally requires baseline correctness. Intervals condition on selected features, fitted probes, calibration, and sampled controls; all-zero samples can yield degenerate intervals. The evaluated records share twelve task IDs between historical runs and the smaller extension, four between the pilot and extension, and none between historical runs and the pilot. No effect is pooled across protocols. Finite, prompt-visible tests, generated-code truncation, and missing historical zero-dose/batch-parity audits limit claims of semantic control.