Exploring Coding Spot: Understanding Parametric Contributions to LLM Coding Performance

Anonymous ACL submission

Abstract

Large Language Models (LLMs) have demonstrated notable proficiency in both code generation and comprehension across multiple programming languages. However, the mechanisms underlying this proficiency remain underexplored, particularly with respect to whether distinct programming languages are processed independently or within a shared parametric region. Drawing an analogy to the specialized regions of the brain responsible for distinct cognitive functions, we introduce the concept of *Coding Spot*, a specialized parametric region within LLMs that facilitates coding capabilities. Our findings identify this *Coding Spot* and show that targeted modifications to this subset significantly affect performance on coding tasks, while largely preserving non-coding functionalities. This compartmentalization mirrors the functional specialization observed in cognitive neuroscience, where specific brain regions are dedicated to distinct tasks, suggesting that LLMs may similarly employ specialized parameter regions for different knowledge domains.

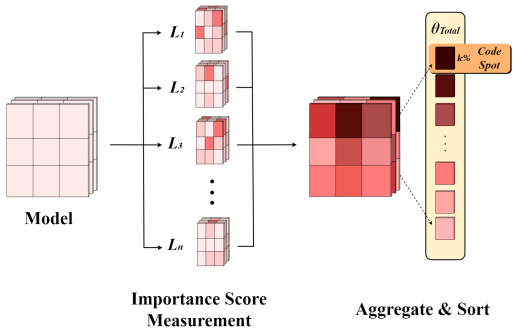

Figure 1: Overview of the framework for extracting and analyzing the *Coding Spot* within LLMs. The model undergoes importance scoring independently for n programming languages; per-language scores are aggregated per-parameter and sorted. The top $k\%$ parameters form the *Coding Spot*.

1 Introduction

Large Language Models (LLMs) have initiated a significant transformation in computational code processing, showcasing advanced capabilities in tasks such as code generation and comprehension across a wide range of programming languages (Chen et al., 2021; Austin et al., 2021; Li et al., 2022). Models such as Llama 3 (Dubey et al., 2024), GPT-5 (OpenAI, 2025), Gemini 2.5 (Comanici et al., 2025), and Claude 4.5 Sonnet (Anthropic, 2025) have achieved considerable success, establishing themselves as essential tools for automating programming tasks and enhancing developer productivity.

Despite these successes, a fundamental question remains: how do these models internally represent and organize the coding knowledge necessary for such tasks? More specifically, is the knowledge required for programming tasks uniformly distributed across the model's parameters, or do certain parameter subsets exhibit specialization for coding-related functionalities? Prior human studies suggest code comprehension draws strongly on domain-general control systems (e.g., the multiple-demand network) rather than classic language areas (Ivanova et al., 2020), and the broader language neuroscience literature emphasizes distributed processing (Binder, 2015). Motivated by these observations, we hypothesize that LLMs may exhibit a *parameter-localized* substrate for coding.

In this study, we introduce the concept of the *Coding Spot*, a subset of parameters within LLMs that are particularly critical for code-related capabilities. We localize this subset using a simple, well-motivated importance score (first-order Taylor, cf. pruning literature (Molchanov et al., 2019)) aggregated across multiple programming languages. We then validate its role via *targeted ablations* and compare against *random ablations* at the same sparsity.

2 Coding Spot

Our methodology aims to identify and analyze the *Coding Spot*, a specialized and *very sparse* subset of parameters crucial for coding proficiency in LLMs. At a high level, we (i) compute per-parameter importance from short, language-specific code fine-tuning runs (for gradients only), (ii) aggregate these scores across languages to obtain a language-agnostic ranking, and (iii) causally validate the subset via targeted vs. random ablations.

Methodological Framework The core of our approach is a simple, reproducible importance estimator grounded in first-order Taylor analysis of loss change with respect to each parameter. We deliberately opt for this classical estimator because it is widely used in pruning/importance literature and scales easily to full models without layer-specific heuristics. To test whether the identified parameters encode *language-agnostic* coding competence, we aggregate importance across multiple non-Python languages but *hold out* Python for evaluation (Section 3). This lets us validate the subset on HumanEval (Python) even though Python contributed no gradients to scoring. Holding out Python when scoring importance but evaluating on HumanEval (Python) tests cross-lingual *generalizability* of the identified subset.

Parameter Importance Scoring We estimate the effect of zeroing a parameter via a first-order Taylor approximation around the current parameters θ. Let $\theta^{(-j)}$ denote θ with θ_j set to 0. Then

$$L(D_l, \theta^{(-j)}) \approx L(D_l, \theta) - \frac{\partial L(D_l, \theta)}{\partial \theta_j}\, \theta_j.$$

This motivates a saliency score proportional to the gradient–weight product:

$$I_j^{(l)}(\theta) \;=\; \left|\tfrac{\partial L(D_l,\theta)}{\partial \theta_j}\right| \cdot |\theta_j|,$$

with elementwise absolute values (L1); results are similar with L2.

Normalization and Stability To mitigate dataset-size effects, we optionally normalize $I^{(l)}$ by $|D_l|$ (or z-score within each l). We retain the original model tokenizer for all languages; our *language-specific tokenization* refers only to language-aware preprocessing/filtering that removes comments and near-natural-language spans. We found that (i) gradient accumulation over a few thousand code tokens per language and (ii) L1 norms provided stable rankings across random seeds.

Aggregating Importance Across Languages We aggregate per-language importance to obtain a language-agnostic score:

$$I_j^{\text{total}}(\theta) \;=\; \sum_{l \in \mathcal{L}} I_j^{(l)}(\theta). \qquad (1)$$

We then sort parameters by I_j^{total} in descending order. This simple sum works well empirically and avoids introducing tunable weights per language; we verified that weighting by dataset size or using a robust mean yields nearly identical top-k sets. The *Coding Spot* is the set of indices corresponding to the top $k\%$ of this ranking.

Defining k and Sensitivity The fraction k controls sparsity. Rather than fixing k a priori, we sweep a small grid $K = \{0.0025, 0.01, 0.09, 0.25\}\%$ and report full curves (Table 1). For interpretability, we define an onset threshold $k^\star$ as the smallest evaluated k for which HumanEval pass@1 $< 1\%$.

3 Experiments

Our evaluation quantifies the role of the *Coding Spot* in both coding and general tasks by *deactivating* (zeroing) small fractions of the identified subset.

3.1 Experimental Setup

Datasets We filter `nampdn-ai/tiny-codes` (Nam Pham, 2023) to only contain *pure code* for fine-tuning. We remove comments and near-natural-language content to avoid conflating instruction-following or English abilities with coding. HumanEval (Chen et al., 2021) evaluates coding; GSM8K (Cobbe et al., 2021), HellaSwag (Zellers et al., 2019), MMLU (Hendrycks et al., 2020), TruthfulQA (Lin et al., 2021), and WinoGrande (Sakaguchi et al., 2021) evaluate general capabilities.

Models We evaluate CodeLlama 7B Instruct (Roziere et al., 2023), Llama 3.1 8B Instruct, and Llama 3.2 3B Instruct (Dubey et al., 2024). Importantly, *Python is **excluded** during fine-tuning* to test whether the *Coding Spot* generalizes across languages (we then evaluate on *Python*-based HumanEval).

	General						Code
	GSM8K	HellaSwag	MMLU	TruthfulQA	WinoGrande	Avg. GTC (%)	HumanEval
CodeLlama 7B Instruct							
Original	18.12	48.32	39.54	39.21	64.56	41.95	87.2
💥 **0.0025%**	1.90	36.01	24.63	39.32	53.51	31.07 (-25.93%)	22.56 (-74.13%)
💥 **0.01%**	2.27	35.08	23.99	36.91	52.41	30.13 (-28.17%)	16.46 (-81.12%)
💥 **0.09%**	0.23	27.23	22.94	49.72	51.38	30.30 (-27.77%)	1.83 (-97.90%)
💥 **0.25%**	0.00	25.85	22.93	51.52	50.51	30.16 (-28.10%)	0.00 (-100.00%)
Llama 3.1 8B Instruct							
Original	76.72	59.10	67.96	54.08	73.64	66.30	97.56
💥 **0.0025%**	2.35	44.47	42.32	43.61	60.14	38.58 (-41.81%)	20.12 (-79.38%)
💥 **0.01%**	2.65	38.83	29.47	41.55	58.41	34.18 (-48.44%)	9.76 (-90.00%)
💥 **0.09%**	0.61	26.82	23.57	49.58	49.17	29.95 (-54.83%)	0.00 (-100.00%)
💥 **0.25%**	0.15	26.36	22.95	51.03	48.46	29.79 (-55.07%)	0.00 (-100.00%)
Llama 3.2 3B Instruct							
Original	64.67	52.22	60.42	49.77	67.56	58.93	94.51
💥 **0.0025%**	2.35	41.39	38.71	45.07	55.96	36.70 (-37.73%)	15.24 (-83.87%)
💥 **0.01%**	2.20	36.31	30.30	45.66	53.75	33.64 (-42.91%)	6.71 (-92.90%)
💥 **0.09%**	1.29	26.66	23.36	50.53	49.49	30.27 (-48.64%)	0.00 (-100.00%)
💥 **0.25%**	1.21	26.21	22.94	49.54	49.96	29.97 (-49.14%)	0.00 (-100.00%)

Table 1: Performance comparison of three LLMs on general and coding tasks after deactivating a percentage of the model identified as *Coding Spot*. GTC represents General Task Change, and 💥 marks models with deactivated parameters.

Preprocessing and Metrics We retain the models' original tokenizers. *Language-specific tokenization* refers to *language-aware preprocessing and filtering* using token-level rules; no new tokenizers are trained. Code performance uses HumanEval pass@1; general tasks use accuracy. We sweep k over $\{0.0025, 0.01, 0.09, 0.25\}\%$.

Counterfactual Random Ablation Alongside *targeted* Coding-Spot ablation, we perform a *random* ablation that zeros the same number of parameters sampled uniformly *without replacement* from all other parameters (i.e., outside the Coding Spot). We report the $k = 0.25\%$ comparison in Table 2, mirroring Table 1.

4 Results

4.1 Main Results

Deactivating even a very small fraction of Coding-Spot parameters causes large and often catastrophic drops on coding, with comparatively smaller effects on general tasks (Table 1). On HumanEval, pass@1 reaches ≈ 0 at $k^\star$=$\{0.09\%, 0.09\%, 0.25\%\}$ for Llama 3.1 8B, Llama 3.2 3B, and CodeLlama 7B, respectively.[1] At k=0.25%, the same models score 0.00, 0.00, 0.00 on HumanEval (down from 97.56, 94.51, 87.20), while the average accuracy on general tasks falls to 29.79, 29.97, 30.16 (from 66.30, 58.93, 41.95), corresponding to changes of -55.1%, -49.1%, and -28.1%, respectively. Because Python was held out when constructing the *Coding Spot*, the collapse on *Python-based* HumanEval supports a *language-agnostic* localization of coding capability (Section 3).

Because Python was explicitly held out when constructing the *Coding Spot*, the failure on *Python-based* HumanEval supports a *language-agnostic* localization of coding capability (Section 3).

Task-wise effects (general benchmarks). The Coding-Spot ablation also affects non-coding tasks, but heterogeneously: GSM8K is most sensitive; MMLU and HellaSwag drop substantially; WinoGrande declines moderately; TruthfulQA sometimes increases (CodeLlama). See details in the main text and Table 1.

Onset thresholds for catastrophic coding failure. Let $k^\star$ denote the smallest tested k that yields HumanEval ≈ 0. We observe $k^\star \approx 0.09\%$ for Llama 3.1 and Llama 3.2, and $k^\star \approx 0.25\%$ for CodeLlama, consistent with greater redundancy from code-heavy pretraining.

Language-agnosticity check. Because Python is *not* used to construct the *Coding Spot*, the collapse on HumanEval (Python) indicates the *Coding Spot* captures *language-agnostic* coding competence rather than language-specific idiosyncrasies.

[1] For brevity we report pass@1.

	GSM8K	HellaSwag	MMLU	General TruthfulQA	WinoGrande	Avg. GTC (%)	Code HumanEval
CodeLlama 7B Instruct							
Original	18.12	48.32	39.54	39.21	64.56	41.95	87.20
💥 **Coding Spot (0.25%)**	0.00	25.85	22.93	51.52	50.51	30.16 (-28.10%)	0.00 (-100.00%)
💥 **Random (0.25%)**	17.44	47.31	39.37	39.81	64.64	41.71 (-0.60%)	34.76 (-60.10%)
Llama 3.1 8B Instruct							
Original	76.72	59.10	67.96	54.08	73.64	66.30	97.56
💥 **Coding Spot (0.25%)**	0.15	26.36	22.95	51.03	48.46	29.79 (-55.07%)	0.00 (-100.00%)
💥 **Random (0.25%)**	75.21	58.78	66.98	52.85	73.16	65.40 (-1.40%)	56.71 (-41.90%)
Llama 3.2 3B Instruct							
Original	64.67	52.22	60.42	49.77	67.56	58.93	94.51
💥 **Coding Spot (0.25%)**	1.21	26.21	22.94	49.54	49.96	29.97 (-49.14%)	0.00 (-100.00%)
💥 **Random (0.25%)**	59.29	51.95	59.59	49.13	67.64	57.52 (-2.40%)	45.12 (-52.30%)

Table 2: **Counterfactual baseline at matched sparsity** (k=0.25%)**.** Randomly zeroing the same number of parameters outside the *Coding Spot* produces negligible changes on general tasks and substantially smaller degradation on HumanEval than targeted Coding-Spot ablation. The 💥 icon marks ablated settings.

	Avg. GTC (%)	Code Change (%)	M_s
CodeLlama 7B Instruct			
Original	41.95	87.20	-
💥 **0.0025%**	-25.93%	-74.13%	5.44
💥 **0.01%**	-28.17%	-81.12%	5.52
💥 **0.09%**	-27.77%	-97.90%	6.75
💥 **0.25%**	-28.10%	-100.00%	6.82
Llama 3.1 8B Instruct			
Original	66.30	97.56	-
💥 **0.0025%**	-41.81%	-79.38%	2.70
💥 **0.01%**	-48.44%	-90.00%	2.65
💥 **0.09%**	-54.83%	-100.00%	2.61
💥 **0.25%**	-55.07%	-100.00%	2.60
Llama 3.2 3B Instruct			
Original	58.93	94.51	-
💥 **0.0025%**	-37.73%	-83.87%	3.41
💥 **0.01%**	-42.91%	-92.90%	3.34
💥 **0.09%**	-48.64%	-100.00%	3.19
💥 **0.25%**	-49.14%	-100.00%	3.15

Table 3: Comparison of LLM performance after *Coding Spot* parameter deactivation. GTC (%) and Code Change (%) show changes in accuracy for general and coding tasks. M_s measures task-specific impact. 💥 marks models with deactivated parameters.

4.2 Specificity: Targeted vs. Random Ablation

Table 2 compares targeted ablations to random ablations that zero the *same number* of parameters sampled from the *complement* of the *Coding Spot*. Targeted ablation reduces HumanEval by 100% for all models at k=0.25%, while random ablation reduces it by 60.1% (CodeLlama), 41.9% (Llama 3.1), and 52.3% (Llama 3.2); general-task averages under random ablation remain near baseline (-0.6%, -1.4%, -2.4%), confirming specificity.

4.3 Coding Spot Impact and Model Differences

We summarize specialization using the monosemanticity score

$$M_s = \frac{\Delta\text{Coding}}{1 + \Delta\text{General}}, \tag{2}$$

computed from Table 3, where Δ denotes relative change from the original accuracy. Llama 3.1/3.2 peak at tiny k=0.0025%, indicating a high density of task-critical parameters, whereas CodeLlama's M_s increases through k=0.25%, suggesting a broader, more redundant *Coding Spot*. This aligns with our pretraining hypothesis: code-heavy pretraining distributes coding-relevant information across a wider subset of weights, yielding a larger onset threshold $k^\star$ before catastrophic HumanEval collapse and smaller general-task degradation at matched k.

5 Conclusion

We identified the *Coding Spot*, a sparse, language-aggregated subset of parameters, whose targeted ablation collapses coding performance far more than random ablation at matched sparsity, while modestly impacting general tasks. The *Coding Spot* suggests (i) *targeted PEFT* (biasing adapters/LoRA toward layers with higher Coding-Spot density), (ii) *controlled editing/safety* for code-restricted deployments, and (iii) *compression/distillation* pathways focused on code-relevant parameters.

Limitations

We empirically select $k\%$; more principled selection criteria are future work. Our intervention zeros parameters; alternative perturbations (e.g., noise injection) may yield different dynamics. We study three Llama-family models; broader families are left for future work.

Ethics Statement

Our fine-tuning and evaluation datasets are publicly available; however, the *base LMs* we analyze may include proprietary pretraining sources beyond our control. We therefore avoid claims about full pretraining data provenance. We note that code generation can raise safety and security concerns; future applications should assess risks accordingly.

References

Anthropic. 2025. Claude sonnet 4.5 system card. Technical report, Anthropic.

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

Elad Ben Zaken, Shauli Ravfogel, and Yoav Goldberg. 2021. Bitfit: Simple parameter-efficient fine-tuning for transformer-based masked language models. *arXiv preprint arXiv:2106.10199*.

Jeffrey R. Binder. 2015. The wernicke area: Modern evidence and a reinterpretation. *Neurology*, 85(24):2170–2175.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. 2021. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*.

Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, et al. 2025. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv preprint arXiv:2507.06261*.

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

Nelson Elhage, Tristan Hume, Catherine Olsson, Nicholas Schiefer, Tom Henighan, Shauna Kravec, Zac Hatfield-Dodds, Robert Lasenby, Dawn Drain, Carol Chen, et al. 2022. Toy models of superposition. *arXiv preprint arXiv:2209.10652*.

Leo Gao, Tom Dupré la Tour, Henk Tillman, Gabriel Goh, Rajan Troll, Alec Radford, Ilya Sutskever, Jan Leike, and Jeffrey Wu. 2024. Scaling and evaluating sparse autoencoders. https://cdn.openai.com/papers/sparse-autoencoders.pdf. OpenAI Technical Report.

Mor Geva, Roei Schuster, Jonathan Berant, and Omer Levy. 2021. Transformer feed-forward layers are key-value memories. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics*.

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. 2020. Measuring massive multitask language understanding. *arXiv preprint arXiv:2009.03300*.

Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. 2019. Parameter-efficient transfer learning for NLP. In *International Conference on Machine Learning*, pages 2790–2799. PMLR.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Lu Wang, and Weizhu Chen. 2022. Lora: Low-rank adaptation of large language models. In *International Conference on Learning Representations*.

Anna A. Ivanova, Shashank Srikant, Yotaro Sueoka, Hope H. Kean, Riva Dhamala, Una-May O'Reilly, Marina U. Bers, and Evelina Fedorenko. 2020. Comprehension of computer code relies primarily on domain-general executive brain regions. *eLife*, 9:e58906.

Dongjun Kim, Gyuho Shim, Yongchan Chun, Minhyuk Kim, Chanjun Park, and Heuiseok Lim. 2025. Benchmark profiling: Mechanistic diagnosis of llm benchmarks. *arXiv preprint arXiv:2510.01232*.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with AlphaCode. *Science*, 378(6624):1092–1097.

Stephanie Lin, Jacob Hilton, and Owain Evans. 2021. Truthfulqa: Measuring how models mimic human falsehoods. *arXiv preprint arXiv:2109.07958*.

Pavlo Molchanov, Arun Mallya, Stephen Tyree, Iuri Frosio, and Jan Kautz. 2019. Importance estimation for neural network pruning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 11264–11272.

Nam Pham. 2023. tiny-codes (revision c13428e).

OpenAI. 2025. Gpt-5 system card. Technical report, OpenAI.

Abhishek Panigrahi, Nikunj Saunshi, Haoyu Zhao, and Sanjeev Arora. 2023. Task-specific skill localization in fine-tuned language models. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 27011–27033. PMLR.

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

Keisuke Sakaguchi, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. 2021. Winogrande: An adversarial winograd schema challenge at scale. *Communications of the ACM*, 64(9):99–106.

Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.

Mukund Sundararajan, Ankur Taly, and Qiqi Yan. 2017. Axiomatic attribution for deep networks. In *International Conference on Machine Learning*, pages 3319–3328. PMLR.

Transformer Circuits Team. 2024. Scaling monosemanticity: Extracting interpretable features from claude 3 sonnet. https://transformer-circuits.pub/2024/scaling-monosemanticity/.

Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. 2019. Hellaswag: Can a machine really finish your sentence? *arXiv preprint arXiv:1905.07830*.

A Related Work

LLMs for code and evaluation. Large language models have rapidly advanced on code generation and understanding tasks (Chen et al., 2021; Austin et al., 2021; Li et al., 2022), with open families such as Llama and Code Llama reporting strong results across multiple programming languages (Dubey et al., 2024; Roziere et al., 2023). While these results demonstrate practical competence, the internal organization of coding knowledge (i.e., where in the parameters coding capability resides) remains under-characterized and motivates our study. Complementary to our parameter-localization focus, recent work introduces *Benchmark Profiling* to mechanistically diagnose what abilities a benchmark truly measures (Kim et al., 2025).

Parameter importance and pruning. A large body of work estimates per-parameter importance to prune or compress networks. Of particular relevance is Taylor-style saliency, which scores parameters via gradient–weight products and has proven effective for structured and unstructured pruning (Molchanov et al., 2019). We adopt this classical estimator for its scalability and clear first-order interpretation: the score approximates the loss change when a scalar weight is zeroed. Unlike pruning whose goal is to remove redundant parameters while preserving average performance, we aggregate importance across *multiple programming languages* to isolate a *coding-specific* subset and then test its *causal* role via matched-sparsity ablations.

Capability localization: parameters, features, circuits, and directions. Mechanistic interpretability (MI) investigates how specific capabilities are implemented at different granularities. Parameter-level work studies *task-specific skill localization* and sparse subnetworks disproportionately causal for a behavior (Panigrahi et al., 2023). Feature-level methods use sparse autoencoders to recover monosemantic features and connect them to behavior (Gao et al., 2024; Transformer Circuits Team, 2024), while theory and toy models analyze superposition and its implications for interpretability (Elhage et al., 2022). Circuit-level analyses map causal pathways across attention and MLP components, and direction/subspace methods discover activation steering axes. Our contribution sits at the *parameter* level: a language-aggregated saliency procedure that localizes a very sparse subset of weights and a direct, whole-model causal validation showing that ablating this subset disproportionately harms coding compared with matched random ablations.

Interpretability tools for language and code. Foundational attribution techniques (saliency, integrated gradients) have long been used to study neural predictions (Simonyan and Zisserman, 2014; Sundararajan et al., 2017). For transformers, components such as feed-forward layers can function as key–value memories, shaping representational structure (Geva et al., 2021). Code-focused analyses examine neuron/feature selectivity, syntax-aware attention, and steering

along code-relevant directions (Transformer Circuits Team, 2024). Rather than explaining *what* units represent, our study asks *where* coding ability is parameterized and quantifies how *sparse* that parameterization can be while remaining causally necessary.

Parameter-efficient fine-tuning and editing. Selective adaptation methods such as adapters (Houlsby et al., 2019), BitFit (Ben Zaken et al., 2021), and LoRA (Hu et al., 2022), show that modifying a small fraction of parameters can yield strong task performance. Our findings link localization with practice: the *Coding Spot* indicates where PEFT capacity may be most effective (e.g., biasing LoRA/adapters toward layers with high Coding-Spot density) and where surgical editing could safely adjust code behavior.

Human cognitive neuroscience (scope and analogy). Classic language neuroanatomy emphasizing discrete "language areas" has shifted toward distributed accounts (Binder, 2015). For programming, neural evidence indicates substantial engagement of domain-general multiple-demand systems rather than classical perisylvian language regions (Ivanova et al., 2020). We therefore use neuroscience only as *high-level inspiration* for specialization and localization, not as a direct mapping: all claims and evidence here are internal to model parameters and their causal roles.

Positioning and novelty. Relative to MI and pruning work, we contribute: (i) a *language-aggregated*, Taylor-style importance procedure tailored to code that yields a consistent, *very sparse* parameter subset across models; (ii) *causal validation* via targeted zeroing with a matched-size *random ablation* counterfactual, showing large differential effects on coding; and (iii) an out-of-distribution design (Python held out for scoring) that supports a *language-agnostic* interpretation of the localized substrate.

B Experimental Setup

Datasets

To evaluate the performance of our methodology, we use the `nampdn-ai/tiny-codes` (Nam Pham, 2023) dataset, which includes Bash, C#, C++, Go, Java, JavaScript, Julia, Ruby, Rust, and TypeScript. This dataset offers a diverse environment for fine-tuning LLMs on language-specific benchmarks, allowing a comprehensive evaluation of coding capabilities. During preprocessing, non-code content was filtered out, and language-aware preprocessing was applied to retain syntactic and semantic integrity for each language.

Languages and Benchmarks

We employ HumanEval (Chen et al., 2021) for code generation and GSM8K (Cobbe et al., 2021), HellaSwag (Zellers et al., 2019), MMLU (Hendrycks et al., 2020), TruthfulQA (Lin et al., 2021), and WinoGrande (Sakaguchi et al., 2021) for general capabilities.

Models

We study CodeLlama 7B Instruct (Roziere et al., 2023), Llama 3.1 8B Instruct (Dubey et al., 2024), and Llama 3.2 3B Instruct (Dubey et al., 2024).