Identifying and Steering Program Validity Awareness Latent Directions in LLMs: A Sparse Autoencoder Analysis of Code Hallucinations

Kriz Royce A. Tahimic
Department of Software Technology
College of Computer Studies
De La Salle University
kriz_tahimic@dlsu.edu.ph

Abstract

Large Language Models (LLMs) generate code with varying degrees of accuracy, and recent studies have identified hallucinations in over 2,000 instances across 3,000 code samples, indicating substantial room for improvement. Yet, approximately 30% of AI-suggested code is being integrated into production systems. Despite the risk and wide adoption, the mechanisms by which these models generate between valid and invalid codes remain poorly understood. Hence, we will mechanistically examine how LLMs internally represent their *program validity awareness* through a three-component methodology: (1) dataset building using MBPP (Mostly Basic Python Problems) dataset paired with Gemma 2 2B generated solutions, (2) Sparse Autoencoder analysis using Gemma Scope to identify latent directions corresponding to program validity awareness, and (3) validation through both statistical methods and causal experiments. Our statistical validation employs AUROC and F1 scores, while causal evidence comes from model steering experiments. By analyzing residual stream activations at the final token position, we expect to identify specific latent dimensions that discriminate between correct and incorrect code implementations. This research advances the understanding of code generation mechanisms while offering practical implications for enhancing software development reliability, potentially reducing economic costs of code hallucinations, and improving safety in high-risk domains.

1 Introduction

The integration of Large Language Models (LLMs) into software development has sparked a fundamental shift in how code is written and maintained across the industry. A comprehensive study by Dohmke et al. (2023) revealed out of nearly one million developers, approximately 30% of all AI-suggested code from GitHub Copilot is accepted and integrated into production systems. The same paper also projected that these productivity enhancements could contribute significantly to economic growth, potentially boosting global GDP by over USD 1.5 trillion by 2030 as these technologies help meet the growing worldwide demand for software development.

However, the widespread adoption of code-capable LLMs brings critical concerns regarding their reliability and safety. These concerns are validated by compelling research from Liu et al. (2024), which documented a troubling prevalence of hallucinations—identifying 2,119 instances across just 3,084 code samples. What makes these findings particularly alarming is the fundamental difference between hallucinations in code versus natural language. Unlike hallucinations in natural language generation, which might result in incorrect but plausible text, code hallucinations can produce non-functional, insecure, or potentially harmful implementations. Code hallucinations become particularly concerning in high-risk domains such as healthcare, banking, and the military, where code reliability is paramount.

Despite the widespread adoption and associated risks, the internal mechanisms of these models remain poorly understood, operating largely as black boxes. This challenge has led researchers to explore various interpretability methods to understand and improve these models. Among these approaches, Mechanistic Interpretability has emerged as a promising direction, focusing on revealing how models represent and process information internally. This approach is especially valuable because it helps identify why models make certain mistakes or errors. The insights gained from mechanistic interpretability can inform better architectural choices for neural networks, making it particularly impactful for safety-critical applications.

In their pursuit to mechanistically understand neural networks, researchers discovered a critical challenge: superposition. This phenomenon re-

veals that neural networks can encode more features than they have available neurons, which might have been the cause of polysemantic neurons. To untangle these overlapping features, researchers turned to Sparse Autoencoders (SAE), which have proven effective at separating these intertwined features into more interpretable components.

After deciding to use SAE to disentangle polysemantic features, researchers face the crucial task of labeling and understanding each latent direction. Some researchers, like Bricken et al. (2023) and Templeton et al. (2024), employ a top-down approach that uses a larger LLM to generate labels for each latent direction based on tokens they activate strongly. In contrast, this paper follows what Ferrando et al. (2024) implements, which is a more bottom-up approach that starts with a label or behavior of interest. Their methodology begins with constructing a dataset to elicit a specific latent direction, which will be inputted into the model. The model's internals will then be observed to identify what latent directions activate in response to the feature-specific dataset.

Ferrando et al. (2024) study uncovered specific latent directions in the model's representation space that encode the model's ability to recognize entities. Using datasets spanning various entity types, including people, movies, cities, and songs, the researchers identified distinct latent directions - some that activate specifically for entities the model can recall facts about and others that activate for entities the model doesn't recognize. Through model steering of these entity recognition latent directions, they demonstrated causal effects on model behavior, showing they could influence whether models would refuse to answer questions about known entities or generate hallucinated information about unknown ones. Their analysis revealed these entity recognition latent directions emerged in the middle layers of the network and regulated how attention mechanisms extracted factual information, providing crucial insights into how models determine whether to generate information or refuse to answer.

While these approaches have successfully revealed latent directions in natural language generation, the application of Sparse Autoencoders to understand program validity awareness latent directions in code-capable LLMs remains unexplored. This represents a significant opportunity to advance our understanding of how models internally represent program validity awareness — the model's self-knowledge if it will generate correct code or not — a critical aspect for improving the reliability and safety of AI-assisted software development.

2 Methodology

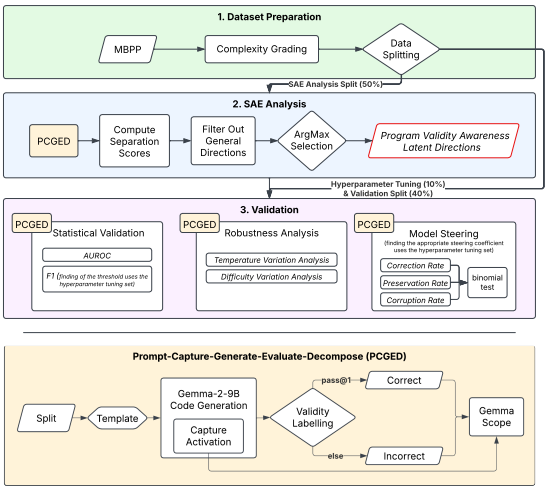

Figure 1: Overview of the three-component methodology: dataset building, SAE analysis, and validation

This chapter outlines the methodological approach used in our study, drawing significant inspiration from the work of Ferrando et al. (2024) in their investigation of entity recognition latent in language models. We adapt their framework while introducing key modifications to address our distinct research objectives.

Our methodology consists of three main components: dataset building to collect and categorize program solutions, SAE activation analysis to identify latent direction, and validation to verify the found program validity awareness latent direction, as illustrated in Figure 1.

2.1 Dataset Preparation

This research utilizes the MBPP (Mostly Basic Python Problems) dataset, which consists of 1,000 Python programming problems designed to evaluate fundamental programming competencies. We enhance the original MBPP dataset by adding a cyclomatic complexity column using McCabe's cyclomatic complexity metric (McCabe, 1976) to enable complexity-based analysis.

Cyclomatic complexity quantifies the structural complexity of program code by measuring the number of linearly independent paths through the program's control flow graph. The metric is computed using the formula:

$$M = E - N + 2P \tag{1}$$

where M represents the cyclomatic complexity, E denotes the number of edges in the control flow graph, N indicates the number of nodes in the graph, and P represents the number of connected components. In the control flow graph representation, nodes correspond to the smallest executable code segments, while directed edges connect these segments based on the program's execution flow. This mathematical foundation allows us to systematically categorize programming problems based on their algorithmic complexity, enabling stratified analysis of how program validity awareness varies across different complexity levels.

The enhanced dataset undergoes stratified random sampling for data splitting to ensure fair representation across complexity levels:

- **Stratification**: Problems are divided into complexity strata using equal-width bins based on complexity scores
- **Randomization**: Task IDs are shuffled within each complexity stratum to ensure fair sampling
- **Interleaving**: A round-robin pattern is applied across all strata simultaneously to prevent complexity bias

This approach results in three balanced sets: SAE analysis (50%), hyperparameter tuning (10%), and validation (40%) that maintain proportional complexity distribution. The hyperparameter tuning set is used to optimize classification thresholds for F1 score calculation and determine appropriate steering coefficients for model interventions.

2.2 SAE Analysis

To analyze program validity awareness latent direction in language models, we employ a systematic approach using the PCGED pipeline (detailed in Section 2.4) to generate labeled code samples with corresponding model activations. The captured activations are then processed through pre-trained Sparse Autoencoders (SAEs) from GemmaScope (Lieberum et al., 2024), which project model representations into higher-dimensional interpretable spaces. The technical architecture of these SAEs is described in the Decompose stage of the PCGED pipeline. This section focuses on our analytical methodology for identifying latent directions that demonstrate program validity awareness.

2.2.1 Separation Score Analysis

We analyze the residual stream at the final token for each code sample in our dataset to capture the model's representation during the input phase when the model consumes the programming problem prompt.

We compute activation statistics for each latent dimension across correct and incorrect code samples. For a given layer l and latent dimension j, we calculate the fraction of activations on correct and incorrect code, respectively:

$$f_{l,j}^{\text{correct}} = \frac{\sum_i^{N^{\text{correct}}} \mathbf{1}[a_{l,j}(\mathbf{x}_{l,i}^{\text{correct}}) > 0]}{N^{\text{correct}}}, \tag{2}$$

$$f_{l,j}^{\text{incorrect}} = \frac{\sum_i^{N^{\text{incorrect}}} \mathbf{1}[a_{l,j}(\mathbf{x}_{l,i}^{\text{incorrect}}) > 0]}{N^{\text{incorrect}}} \tag{3}$$

To identify latent dimensions that effectively distinguish between correct and incorrect code, we compute separation scores:

$$s_{l,j}^{\text{correct}} = f_{l,j}^{\text{correct}} - f_{l,j}^{\text{incorrect}}, \tag{4}$$

$$s_{l,j}^{\text{incorrect}} = f_{l,j}^{\text{incorrect}} - f_{l,j}^{\text{correct}} \tag{5}$$

2.2.2 General Language Filtering

To isolate Python-specific validity features from general language patterns, we apply a filtering step using the pile-10k dataset[1], the first 10,000 samples from the Pile (Gao et al., 2020). We process these diverse text samples through the same SAE pipeline, extracting activations at random word positions. For each SAE feature, we compute its activation frequency on general text:

$$f_{l,j}^{\text{pile}} = \frac{\sum_{i=1}^{N^{\text{pile}}} \mathbf{1}[a_{l,j}(\mathbf{x}_{l,i}^{\text{pile}}) > 0]}{N^{\text{pile}}} \tag{6}$$

Features with $f_{l,j}^{\text{pile}} > 0.02$ are excluded as they represent general language patterns rather than code-specific features. This 2% threshold effectively removes ubiquitous language features while preserving Python-relevant directions.

2.2.3 Latent Direction Selection

After filtering, we identify the Program Validity Awareness (PVA) latent directions by selecting features with the highest separation scores:

[1] https://huggingface.co/datasets/NeelNanda/pile-10k

$$\text{PVA}^{+} = \arg\max_{l,j}\{s_{l,j}^{\text{correct}} : f_{l,j}^{\text{pile}} \leq 0.02\} \quad (7)$$

$$\text{PVA}^{-} = \arg\max_{l,j}\{s_{l,j}^{\text{incorrect}} : f_{l,j}^{\text{pile}} \leq 0.02\} \quad (8)$$

The identified correct and incorrect code directions are the Program Validity Awareness Latent Directions which will be validated and steered.

2.3 Validation

2.3.1 Statistical Analysis

To validate the identified program validity awareness latent direction of the topmost correct code latent and incorrect code latent, we employ two statistical measures that assess different aspects of the latent direction's effectiveness and generalizability.

First, we evaluate the latent direction's statistical significance using the Area Under the Receiver Operating Characteristic (AUROC) curve. This metric comprehensively assesses the model's ability to distinguish between correct and incorrect code implementations across various classification thresholds. The AUROC analysis offers insights into the latent direction's robustness and reliability across different decision boundaries. The AUROC score is calculated as follows:

$$\text{AUROC} = \int_0^1 \text{TPR}(\text{FPR})\, d\text{FPR} \quad (9)$$

The integral formulation captures the relationship between true and false positive rates across all possible classification thresholds. The True Positive Rate (TPR) measures the proportion of correct code implementations that are accurately identified (TP/(TP + FN)), while the False Positive Rate (FPR) indicates the proportion of incorrect implementations mistakenly classified as correct (FP/(FP + TN)). This mathematical framework quantifies the model's discriminative capability independent of any specific threshold, where a score closer to 1.0 indicates superior discrimination ability, and 0.5 suggests performance equivalent to random chance.

Second, we compute the F1 score, which provides a balanced measure between precision and True Positive Rate (TPR) in identifying program validity. The F1 score is calculated as the harmonic mean:

$$F_1 = \frac{2 \cdot \text{precision} \cdot \text{TPR}}{\text{precision} + \text{TPR}} \quad (10)$$

where precision = TP/(TP + FP) represents the proportion of correctly identified valid programs among all programs classified as valid. The multiplication by 2 in the numerator and the division by the sum in the denominator creates a harmonic mean, ensuring equal weighting between precision and TPR.

To compute the F1 score, we must first determine an optimal classification threshold for our classifier that converts our continuous model outputs into binary decisions of valid or invalid programs. We use our hyperparameter tuning set to evaluate multiple threshold values, calculating each threshold's resulting TP, FP, TN, and FN counts. These counts produce different precision and TPR values, yielding different F1 scores. We select the threshold that maximizes the F1 score on the development set and then apply this optimized threshold to our validation dataset for final evaluation. The resulting F1 score ranges from 0 to 1, where 1 indicates perfect precision and recall (TPR), while 0 indicates complete failure in either metric. This systematic threshold optimization ensures that our latent direction maintains high accuracy while minimizing misclassifications in both directions, providing a comprehensive metric that captures the latent direction's overall performance in distinguishing between valid and invalid programs.

2.3.2 Robustness Analysis

To ensure the reliability and generalizability of our findings, we conduct two types of robustness analyses to evaluate the stability of the identified program validity awareness latent directions under varying conditions.

First, the Temperature Variation Analysis examines how varying temperature affects the statistical significance of our program validity awareness latent directions. Crucially, we capture the model's internal activations only once using deterministic sampling (temperature = 0), as the underlying representations remain invariant to sampling temperature—only the token selection probabilities are affected by temperature scaling. We then systematically evaluate performance across temperature values of 0.0, 0.2, 0.4, 0.6, 0.8 and 1.0, where higher temperatures introduce stochasticity solely in the token sampling process. For each non-zero temperature, we generate three distinct code solutions per problem by sampling from the same activation-derived probability distributions with different random seeds. The statistical validation metrics (AU-

ROC and F1 score) are computed for each temperature setting, revealing how sampling variability in the decoding process affects the apparent statistical significance of the identified PVA latent directions while the underlying neural representations remain constant.

Second, the Complexity Variation Analysis examines how the identified latent directions generalize across problems of varying complexity. We employ McCabe's cyclomatic complexity (McCabe, 1976) to categorize MBPP problems into three difficulty groups: easy (complexity = 1), medium (complexity = 2-3), and hard (complexity $\geq$ 4). These thresholds were chosen to ensure approximately equal distribution of samples across the three groups, maximizing statistical power for each difficulty category. This stratified analysis, computing AUROC and F1 scores for each complexity group, reveals whether the program validity awareness mechanism operates uniformly across problem difficulties or exhibits complexity-dependent effectiveness.

2.3.3 Model Steering

To validate our discovery of program validity awareness latent in the model's representations, we employ a technique called model steering using the identified Sparse Autoencoder (SAE) latents. This approach leverages the fundamental property of SAEs: their ability to reconstruct a model's internal representations as interpretable latent directions. Mathematically, an SAE reconstructs a model's representation $\mathbf{x}$ as a combination of learned latent directions: $\mathbf{x} \approx a(\mathbf{x})\mathbf{W}_{\text{dec}} + \mathbf{b}_{\text{dec}}$. This reconstruction can be understood more intuitively by expanding it into its components: $\mathbf{x} \approx \sum_j a_j(\mathbf{x})\mathbf{W}_{\text{dec}}[j, :]$. Here, each row of the decoder matrix $\mathbf{W}_{\text{dec}}$ represents a learned latent direction, and $a_j(\mathbf{x})$ represents how strongly that latent direction is present in the input. Modifying these activation values allows us to control how much each latent direction contributes to the model's processing. Building on this understanding, we can steer the model's behavior by adjusting the activation value of specific SAE latent directions. This process, known as model steering, involves updating the model's residual stream according to the equation:

$$\mathbf{x}^{\text{new}} \leftarrow \mathbf{x} + \alpha \mathbf{d}_j \tag{11}$$

where α controls the strength of the intervention and $\mathbf{d}_j$ is the latent direction we wish to accentuate. This allows us to systematically test how the identified program validity awareness latent directions influence the model's behavior, validating our findings empirically.

To determine optimal steering coefficients α, we employ an adaptive binary search strategy on the hyperparameter tuning set. We begin with a coarse search across exponentially spaced values (1, 10, 100, 1000) to identify the active range where steering effects emerge without saturation. Within the identified active range, we conduct binary search to maximize a composite score tailored to each steering type. For validity steering, we combine the correction rate with the preservation rate, ensuring the intervention both fixes incorrect code and avoids breaking already-correct solutions. For invalidity steering, we combine the corruption rate with code similarity, as effective steering should introduce bugs while maintaining syntactic structure rather than producing incoherent output. This dual-metric approach prevents degenerate solutions where steering either fails to affect the target samples or damages output quality. The strategy evaluates each coefficient on the complete hyperparameter tuning set of approximately 97 problems, providing statistically reliable selection.

To quantify the impact of our model steering, we introduce three evaluation metrics. Let steered_i^+ denote a sample steered with the correct latent direction and steered_i^- denote a sample steered with the incorrect latent direction.

For the correct latent direction, we measure the Correction Rate, which quantifies the proportion of initially incorrect code samples that become correct after steering:

$$\text{Correction Rate} = \frac{1}{N^{\text{incorrect}}} \sum_{i=1}^{N^{\text{incorrect}}} \mathbf{1}[\text{IsCorrect}(\text{steered}_i^+) = \text{True}] \tag{12}$$

Additionally, to assess whether the correct latent direction preserves validity in already-correct code, we measure the Preservation Rate:

$$\text{Preservation Rate} = \frac{1}{N^{\text{correct}}} \sum_{i=1}^{N^{\text{correct}}} \mathbf{1}[\text{IsCorrect}(\text{steered}_i^+) = \text{True}] \tag{13}$$

For the incorrect latent direction, we measure the Corruption Rate, which captures the proportion of

initially correct code that becomes incorrect after steering:

$$\text{Corruption Rate} = \frac{1}{N^{\text{correct}}} \sum_{i=1}^{N^{\text{correct}}} \mathbf{1}[\text{IsCorrect}(\text{steered}_i^-) = \text{False}] \tag{14}$$

To determine whether these observed rates represent statistically significant effects, we apply Binomial Testing to each rate. For a given test with n trials, k successes, and baseline probability p_0, the probability of observing k or more successes under the null hypothesis is:

$$P(X \geq k) = \sum_{i=k}^{n} \binom{n}{i} p_0^i (1-p_0)^{n-i} \tag{15}$$

For the Correction Rate with the correct latent direction, we test:

$$H_0 : \text{Correction Rate} = p_0 \tag{16}$$
$$H_1 : \text{Correction Rate} > p_0 \tag{17}$$

For the Preservation Rate, we test whether correct code remains correct:

$$H_0 : \text{Preservation Rate} = p_0 \tag{18}$$
$$H_1 : \text{Preservation Rate} > p_0 \tag{19}$$

For the Corruption Rate with the incorrect latent direction, we test:

$$H_0 : \text{Corruption Rate} = p_0 \tag{20}$$
$$H_1 : \text{Corruption Rate} > p_0 \tag{21}$$

To establish valid baseline flip rates (p_0) for our statistical tests, we steer latent directions picked from the same layer of program validity awareness latent direction through simple random sampling using the same steering coefficient α. We apply these control interventions to both initially correct and incorrect samples to measure the background rate of state changes that occur due to non-specific perturbations. These baseline rates serve as our null hypothesis values, allowing us to determine whether our targeted latent direction interventions produce statistically significant effects beyond what would be expected from arbitrary model steering. We reject the null hypothesis when p-values fall below 0.05, allowing us to conclude with statistical confidence that our identified latent directions causally influence code correctness.

2.4 Prompt-Capture-Generate-Evaluate-Decompose (PCGED) Pipeline

The PCGED pipeline represents a systematic workflow that underlies each of our four main analysis components: SAE Analysis, Statistical Analysis, Robustness Analysis, and Model Steering. The pipeline is applied across two main phases: first, SAE Analysis uses the SAE split (50% of data) and captures activations from all model layers to identify optimal latent directions. Second, the validation components use the validation and hyperparameter tuning splits, capturing only the specific layers where the best program validity awareness latent directions were identified. This unified approach ensures consistent data generation and processing across all experimental phases.

2.4.1 Pipeline Stages

The PCGED pipeline consists of five sequential stages:

Prompt Stage: Each programming problem from the MBPP dataset is formatted using a standardized template structure to ensure consistent input formatting.

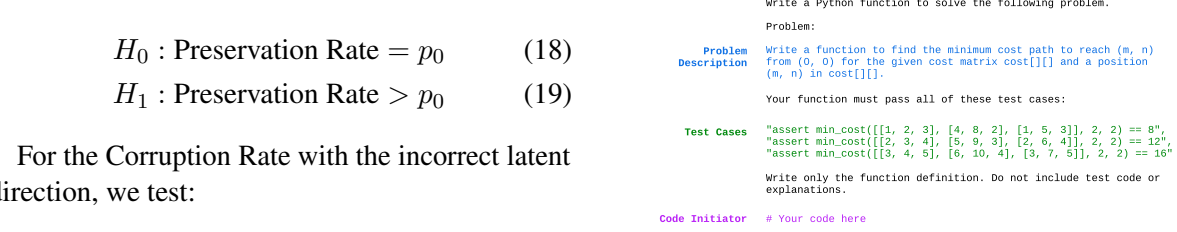

Figure 2: Standardized prompt template containing three components: problem description, test cases, and code initiator.

The template contains three essential components: the problem description provides the fundamental problem statement, test cases demonstrate expected input-output behavior with function signatures, and the code initiator ensures proper code generation from the base model. Context annotations clarify structure and purpose, ensuring consistent formatting while making the template's organization explicit for both human readers and the language model.

Capture Stage: During model inference, we capture residual stream activations at the final token position, providing the neural representations that will be decomposed through SAEs.

Generate Stage: Google's Gemma 2

language model with 2 billion parameters (`google/gemma-2-2b`) (Team et al., 2024) generates code solutions. Temperature is set to 0 for deterministic, reproducible outputs that focus on the model's base behavior in code generation tasks.

Evaluate Stage: Each generated code solution is evaluated against MBPP test cases using pass@1 criterion. Solutions passing all test cases are classified as correct, while any failures (compilation errors, runtime exceptions, or incorrect outputs) result in incorrect classification.

Decompose Stage: The labeled samples and their corresponding activations undergo Sparse Autoencoder (SAE) decomposition using pre-trained models from GemmaScope. This stage transforms the raw neural activations into interpretable latent representations that can reveal program validity awareness patterns.

We employ the JumpReLU SAE architecture, which projects model representations $\mathbf{x} \in \mathbb{R}^d$ into a larger dimensional space $a(\mathbf{x}) \in \mathbb{R}^{d_{\text{SAE}}}$ to enable more granular analysis of neural activations. The encoding process applies a linear transformation followed by a JumpReLU activation function:

$$a(\mathbf{x}) = \text{JumpReLU}_\theta(\mathbf{x}\mathbf{W}_{\text{enc}} + \mathbf{b}_{\text{enc}}) \tag{22}$$

where JumpReLU implements a threshold activation defined as:

$$\text{JumpReLU}_\theta(\mathbf{x}) = \mathbf{x} \odot H(\mathbf{x} - \theta) \tag{23}$$

Here, H represents the Heaviside step function, and θ is a learnable threshold vector. The decoder reconstructs the original representation through:

$$\text{SAE}(\mathbf{x}) = a(\mathbf{x})\mathbf{W}_{\text{dec}} + \mathbf{b}_{\text{dec}} \tag{24}$$

The autoencoder training minimizes a combined loss function that balances reconstruction accuracy with sparsity:

$$\mathcal{L}(\mathbf{x}) = \underbrace{\|\mathbf{x} - \text{SAE}(\mathbf{x})\|_2^2}_{\mathcal{L}_{\text{reconstruction}}} + \underbrace{\lambda\|a(\mathbf{x})\|_0}_{\mathcal{L}_{\text{sparsity}}} \tag{25}$$

This decomposition addresses the superposition problem in neural networks, where multiple features are entangled within individual neurons. By expanding the captured activations into a higher-dimensional sparse space, the SAE disentangles these overlapping representations, enabling cleaner identification of program validity awareness (PVA) latent directions with reduced interference from other linguistic and computational features. The resulting sparse activations are then analyzed using the separation score methodology detailed in the SAE Analysis section.

2.5 Calendar of Activities

Table 1 shows a Gantt chart of the activities. Each bullet represents approximately one week's worth of activity.

	2025							
Activity	**Jan**	**Feb**	**Mar**	**Apr**	**May**	**Jun**	**Jul**	**Aug**
Dataset Building				****	****			
SAE Activation Analysis					****	****		
Statistical Analysis						****	****	
Steering Analysis							****	****
Documentation	****	****	****	****	****	****	****	****

Table 1: Gantt Chart of Activities

References

Trenton Bricken, Adly Templeton, Joshua Batson, Brian Chen, Adam Jermyn, Tom Conerly, Nick Turner, Cem Anil, Carson Denison, Amanda Askell, Robert Lasenby, Yifan Wu, Shauna Kravec, Nicholas Schiefer, Tim Maxwell, Nicholas Joseph, Zac Hatfield-Dodds, Alex Tamkin, Karina Nguyen, and 6 others. 2023. Towards monosemanticity: Decomposing language models with dictionary learning. *Transformer Circuits Thread*. Https://transformer-circuits.pub/2023/monosemantic-features/index.html.

Thomas Dohmke, Marco Iansiti, and Greg Richards. 2023. Sea change in software development: Economic and productivity analysis of the ai-powered developer lifecycle. *arXiv preprint arXiv:2306.15033*.

Javier Ferrando, Oscar Obeso, Senthooran Rajamanoharan, and Neel Nanda. 2024. Do i know this entity? knowledge awareness and hallucinations in language models. *arXiv preprint arXiv:2411.14257*.

Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, and 1 others. 2020. The pile: An 800gb dataset of diverse text for language modeling. *arXiv preprint arXiv:2101.00027*.

Tom Lieberum, Senthooran Rajamanoharan, Arthur Conmy, Lewis Smith, Nicolas Sonnerat, Vikrant Varma, János Kramár, Anca Dragan, Rohin Shah, and Neel Nanda. 2024. Gemma scope: Open sparse autoencoders everywhere all at once on gemma 2. *arXiv preprint arXiv:2408.05147*.

Fang Liu, Yang Liu, Lin Shi, Houkun Huang, Ruifeng Wang, Zhen Yang, Li Zhang, Zhongqi Li, and Yuchi

Ma. 2024. Exploring and evaluating hallucinations in llm-powered code generation. *arXiv preprint arXiv:2404.00971*.

Thomas J McCabe. 1976. A complexity measure. *IEEE Transactions on software Engineering*, (4):308–320.

Gemma Team, Morgane Riviere, Shreya Pathak, Pier Giuseppe Sessa, Cassidy Hardin, Surya Bhupatiraju, Léonard Hussenot, Thomas Mesnard, Bobak Shahriari, Alexandre Ramé, and 1 others. 2024. Gemma 2: Improving open language models at a practical size. *arXiv preprint arXiv:2408.00118*.

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, and 3 others. 2024. Scaling monosemanticity: Extracting interpretable features from claude 3 sonnet. *Transformer Circuits Thread*.