

Context

AI-assisted coding has achieved **widespread adoption**:

- 30% of AI-suggested code accepted into production¹
- Projected **\$1.5T** GDP boost by 2030¹

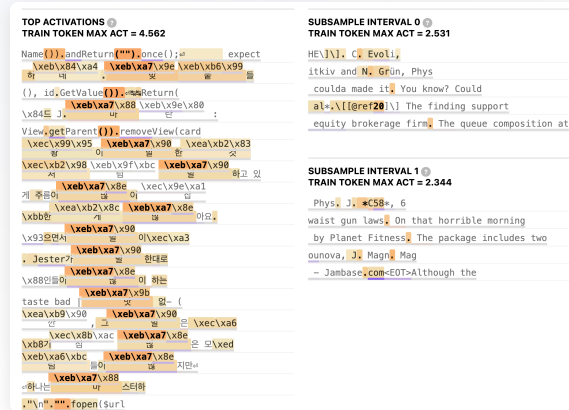
The Problem

LLMs' **internal mechanisms** for code correctness remain poorly understood.

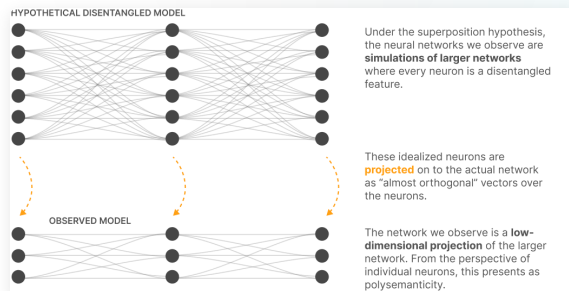
- 44% of LLM bugs identical to historical training errors²
- Only 12.27% accuracy in bug-prone contexts²
- Critical for **high-stakes systems** (healthcare, banking, military) demanding transparency

The Challenge

Understanding LLMs requires analyzing individual neurons—but neurons are **polysemantic**, responding to multiple unrelated concepts like academic citations, English dialogue, HTTP requests, and Korean text.

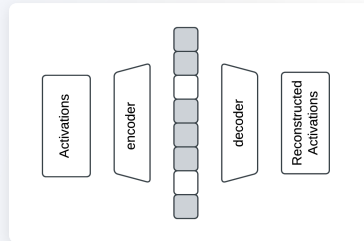


A hypothesized cause is **superposition**: networks encode more features than available dimensions. The observed model is a low-dimensional projection of a larger, idealized network where features would be disentangled.



Our Approach

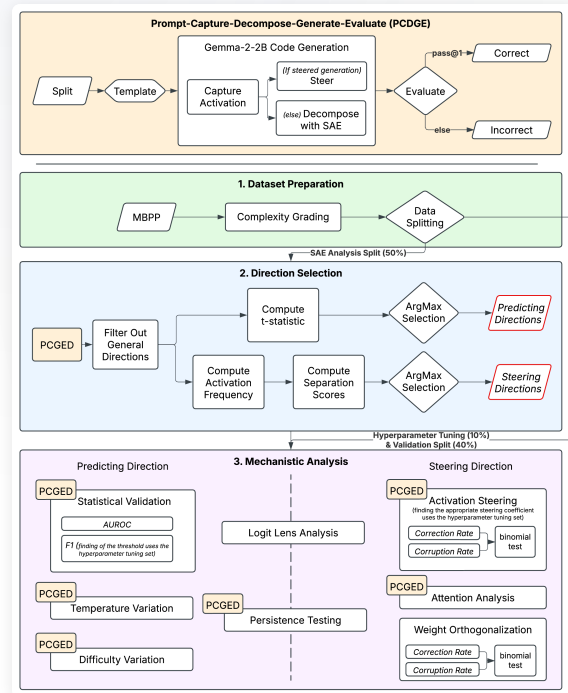
Sparse Autoencoders (SAEs) address superposition by expanding activations into a higher-dimensional sparse space, decomposing entangled representations into interpretable directions.



SAE: Activations → sparse latent space → reconstructed activations

Methodology Pipeline

Using 1,000 Python problems from MBPP, we capture residual stream activations at the final prompt token across all layers, then identify two predicting directions (correct/incorrect, via t-statistics) and two steering directions (correct/incorrect, via separation scores).



Key Discovery

Code correctness directions **EXIST** in LLM representations, revealing an **asymmetry**.

Identified Directions

Name	Direction	Result
Incorrect Predicting	L19-5441	F1=0.821 ✓
Correct Predicting	L16-14439	F1=0.504 ×
Correct Steering	L16-11225	4.04% > 0% ctrl (p<0.001) ✓
Incorrect Steering	L25-2853	64.66% < 100% ctrl (p=1.0) ×

*Note: Correct steering also corrupts 14.66% of initially correct code, suggesting selective application.

The asymmetry works in our favor: we can detect errors AND steer toward correctness

What does Incorrect-Predicting detect?

POSITIVE LOGITS @		
none	1.35	
None	1.28	
none	1.24	
None	1.21	
SourceChecksum	1.01	Anomalies: null, None, foreign tokens
NONE	0.96	
NONE	0.94	
autorytatywna	0.89	
なし	0.87	
بلاسيما	0.83	

Correct Steering in Action (L16-11225)

```
BEFORE STEERING ✗
def char_frequency(string):
    return dict.fromkeys(string,
0)

AFTER STEERING ✓
def char_frequency(string):
    frequency = {}
    for char in string:
        if char in frequency:
            frequency[char] += 1
        else:
            frequency[char] = 1
    return frequency
```

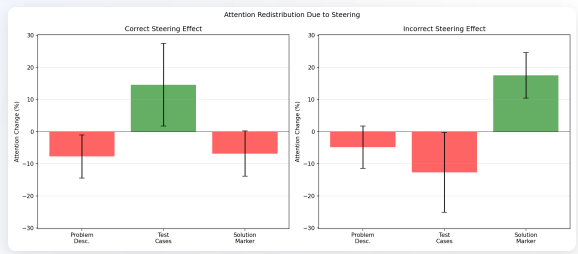
◆ Mechanistic Analysis

Attention Analysis

Our prompts contain three components:

```
Problem Description Write a function to find the minimum cost path to reach (m, n) from (0, 0) for the given cost matrix cost[][] and a position (m, n) in cost[][].
Test Cases "assert min_cost([[1, 2, 3], [4, 8, 2], [1, 5, 3]], 2, 2) == 8",
"assert min_cost([[2, 3, 4], [5, 8, 3], [2, 6, 4]], 2, 2) == 12",
"assert min_cost([[3, 4, 5], [6, 10, 4], [3, 7, 5]], 2, 2) == 16"
Code Initiator # Solution:
```

Where does the model focus attention when steering activates?



Correct-steering redirects attention to **test cases** (+15%), while incorrect-steering shifts away (-13%). *Implication: Prompting should prioritize test examples.*

Weight Orthogonalization

Is the direction merely correlated, or **necessary**? We surgically remove each from model weights.

Correct Steering Removal

CORRUPTION RATE

83.6%

vs 19% control (4.4x) ✓

Incorrect Steering Removal

CORRECTION RATE

2.2%

< 5.5% control ✗

Correct-steering is **necessary** for generation; incorrect-steering removal doesn't fix errors (asymmetry confirmed).

Persistence Across Fine-tuning

Do these directions persist from base to instruction-tuned models?

Incorrect-predicting

F1: 0.821 → 0.772

Correct-steering

4.04% → 2.93% (p<0.001)

Both directions persist through fine-tuning, confirming these are fundamental mechanisms.

◆ Research Contribution

- **First application** of Sparse Autoencoders to mechanistically interpret code correctness in LLMs
- **Adapted** mechanistic interpretability techniques from entity recognition⁶ to the code generation domain

◆ Practical Applications

1. Error Detection Systems

- Integrate incorrect-predicting directions into development workflows
- Flag AI-generated code before production deployment
- Implementation: IDE plugins, CI/CD pipeline checks, API services

2. Prompting Best Practices

- Prioritize test examples over problem descriptions when prompting
- Invest effort in crafting detailed test cases rather than lengthy descriptions
- No model retraining required—simple prompt restructuring

3. Selective Steering Pipeline

- Two-stage approach combining prediction and steering
- **Stage 1 - Predict:** Use incorrect-predicting direction (L19-5441) to identify likely errors
- **Stage 2 - Steer:** Apply correct-steering direction (L16-11225) only on flagged samples
- Avoids corrupting initially correct code through universal steering

◆ References

1. Dohmke et al. (2023). Sea Change in Software Development: Economic and Productivity Analysis of the AI-Powered Developer Lifecycle.
2. Guo et al. (2025). LLMs are Bug Replicators: An Empirical Study on LLMs' Capability in Completing Bug-prone Code.
3. Bricken et al. (2023). Towards Monosemanticity: Decomposing Language Models With Dictionary Learning.
4. Templeton et al. (2024). Scaling Monosemanticity: Extracting Interpretable Features from Claude 3 Sonnet.
5. Lieberum et al. (2024). Gemma Scope: Open Sparse Autoencoders Everywhere All at Once on Gemma 2.
6. Ferrando et al. (2024). Do I Know This Entity? Knowledge Awareness and Hallucinations in Language Models.

Contact

PROPOSER

Kriz Tahimic

kriz_tahimic@dlsu.edu.ph

ADVISER

Dr. Charibeth K. Cheng

charibeth.cheng@dlsu.edu.ph

Mechanistic Interpretability

of Code Correctness in LLMs

via Sparse Autoencoders



Kriz Tahimic

Adviser: Dr. Charibeth K. Cheng

College of Computer Studies
De La Salle University

Academic Year 2025–2026