Capturing Semantic Flow of ML-based Systems

Shin Yoo
KAIST
Daejeon, Korea
shin.yoo@kaist.ac.kr

Robert Feldt
Chalmers University
Gothenburg, Sweden
robert.feldt@chalmers.se

Somin Kim
KAIST
Daejeon, Korea
somin.kim@kaist.ac.kr

Naryeong Kim
KAIST
Daejeon, Korea
naryeong.kim@kaist.ac.kr

Abstract

ML-based systems are software systems that incorporates machine learning components such as Deep Neural Networks (DNNs) or Large Language Models (LLMs). While such systems enable advanced features such as high performance computer vision, natural language processing, and code generation, their *internal* behaviour remain largely opaque to traditional dynamic analysis such as testing: existing analysis typically concern only what is observable from the outside, such as input similarity or class label changes. We propose semantic flow, a concept designed to capture the internal behaviour of ML-based system and to provide a platform for traditional dynamic analysis techniques to be adapted to. Semantic flow combines the idea of control flow with internal states taken from executions of ML-based systems, such as activation values of a specific layer in a DNN, or embeddings of LLM responses at a specific inference step of LLM agents. The resulting representation, summarised as *semantic flow graphs*, can capture internal decisions that are not explicitly represented in the traditional control flow of ML-based systems. We propose the idea of semantic flow, introduce two examples using a DNN and an LLM agent, and finally sketch its properties and how it can be used to adapt existing dynamic analysis techniques for use in ML-based software systems.

CCS Concepts

• **Software and its engineering → Software creation and management**.

Keywords

Program Analysis, Large Language Models, Agents

ACM Reference Format:
Shin Yoo, Robert Feldt, Somin Kim, and Naryeong Kim. 2025. Capturing Semantic Flow of ML-based Systems. In *Proceedings of 33rd ACM International Conference on the Foundations of Software Engineering (FSE Companion '25).* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3696630.3728507

1 Introduction

Recent rapid advances in machine learning, particularly in Deep Neural Networks (DNNs) [16] and Large Language Models (LLMs) [3, 20] have led to the emergence of ML-based software system, a new type of software systems that uses machine learning components such as DNNs or LLMs as part of the system. ML-based systems range from safety-critical systems that incorporate vision-related DNNs [19] to agentic systems driven by LLMs [10, 24]. We expect such ML-based systems to be a prevalent form of software systems in the future, thanks to the unsurpassed capabilities of DNNs (in computer vision [5, 6] and speech recognition [7]) and LLMs (in various natural language processing tasks [3], text-based games [2, 23], language education [1], as well as software engineering tasks [4]).

With the increasingly wider adoption of ML-based systems, the need to analyse and verify their behaviour also increases, so that we can ensure the quality of service provided by these systems. However, it is not straightforward to apply existing program analysis techniques, such as testing and debugging, to ML-based systems, because the internal behaviour of the ML components (i.e., DNNs or Transformer models of LLMs) are fundamentally different from traditional software and therefore remain opaque. In machine learning literature, attempts to understand the internal behaviour of ML components tend to focus on identifying features of inputs [8, 27, 28]. While these work help us understand ML components themselves better, they fall short of providing a perspective for a whole ML-based software system, which involves parts written as traditional software as well as ML components.

We propose a new representation of ML-system behaviour called semantic flow. Similarly to control and data flow for traditional software, semantic flow captures how semantic information in the latent space used by ML components changes as the system executes. It is different from existing attempt to capture input features, as semantic flow also captures the idea of system execution, traditionally represented in control flow. We present semantic flow using examples of both a DNN and an LLM-based agent system, show connections to existing work in testing of ML systems, and finally discuss future work on how traditional dynamic analysis techniques such as testing and fault localisation can be adapted to ML-based systems using semantic flow as a base representation.

2 Motivation

Let us first consider a simple image classification DNN, whose source code is listed in Figure 1. Both the model structure defined in `__init__` and the actual computation defined in `forward` are linear, i.e., lack any branching. Regardless of the classification results, all inputs follow the same execution path (i.e., the sequence of layers). However, classification expressed in traditional programming language would inherently based on branching, as shown in Figure 2, because we use control flow to define program behaviour. In contrast, the CNN model in Figure 1 performs the classification solely along the dataflow, i.e., by changing the distributions of activation values in the latent space.

Another motivating example we present is of AutoFL [10], an agentic Fault Localisation (FL) technique that is driven by an LLM. At its core, AutoFL uses a ReAct [25] like function calling feature of GPT-3.5 and GPT-4 to overcome the limitations of context window

FSE Companion '25, June 23–28, 2025, Trondheim, Norway

ACM ISBN 979-8-4007-1276-0/25/06
https://doi.org/10.1145/3696630.3728507

```
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = torch.flatten(x, 1)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

Figure 1: A Convolutional Neural Network for Image Classification written in PyTorch

```
def classify(input):
    if features(input) == features["class_A"]:
      return "class_A"
    elif features(input) == features["class_B"]:
      return "class_B"
    elif features(input) == features["class_C"]:
      return "class_C"
    ...
```

Figure 2: A Classifier Logic expressed in Python

lengths in LLMs. After being presented with the source code of the failing test case, the LLM is given autonomy over which function call to invoke. The LLM instance respond to the initial prompt with a series of requests to invoke one of the four available functions:

- `get_class_covered`: returns the list of classes covered by the failing test case
- `get_method_covered`: given a class signature, returns the list of its methods covered by the failing test case
- `get_code_snippet`: given a method signature, returns the method body source code
- `get_comments`: given a method signature, returns the docstring that accompanies the method

Authors report that AutoFL can autonomously navigate the code repository, starting from the failing test case and gradually narrowing down to the location of the fault: it tends to start by looking at the list of covered classes, then the list of methods, and finally code snippets and comments of various methods, which is not unreasonable to human eyes. However, details of such behaviour can only be discerned in the semantic contents of LLM generated responses. From the perspective of traditional control flow, AutoFL simply repeats a loop between the source code that implements the function call invocations, and the LLM that makes such requests, as shown in Figure 3. The control flow alone does not reveal the rich semantic information, which is included in the contents of the function call requests made by the LLM.

Based on these two examples, we argue that we need a new representation that combines the traditional control flow and the semantic information that drives ML components such as DNNs

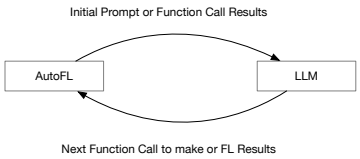


Figure 3: Outline of AutoFL, an LLM-based FL agent [10]

or LLMs. Such a representation would essentially serve as an alternative to the traditional concept of *execution traces*, and allow us to investigate the behaviour of ML-based systems in a way that is more similar to traditional software.

3 Semantic Flow

Semantic flow describes the sequential progression through one or more latent spaces within an ML component or ML-based system, analogous to how control flow describes the order of execution of program elements in traditional software. When given an input, an ML component (e.g., a deep neural network) or system (e.g., a large language model acting as an agent) traverses multiple latent spaces as part of its internal processing. These latent spaces represent the internal states and transformations that occur during execution.

We define a *semantic state*, s_i, as a point in a latent space, encapsulating the representation of the system's internal state at a specific moment. A semantic flow is a sequence of such states, $s_1, s_2, \ldots, s_n$ where each s_i resides in a latent space LS_i corresponding to a distinct phase or step in the execution. For a DNN, LS_i might represent the activations of a particular layer, while for an agentic system, LS_i could denote the reasoning state at a specific step.

To generalise and analyse multiple executions, we aggregate semantic states into semantic clusters, forming a *semantic flow graph* (SFG). In this graph, nodes represent clusters of semantically related states within one of the latent spaces, and edges capture transitions between these clusters observed—or theoretically possible—across a set of executions.

To construct semantic flows, three core elements are required:

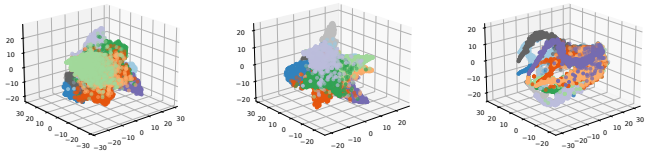
- *Unit of Analysis*: Define the specific execution steps to model and identify the relevant data structures for the internal state at each step. Let e_i represent the execution data at step i.
- *Latent Mapping*: Specify a function $embed(i, e_i)$ that maps execution data e_i at step i into a semantic state s_i in the latent space LS_i.
- *Semantic aggregation*: Specify a semantic aggregation function, $aggregate(i, s_i, S_i)$, where S_i represents all states mapped to LS_i. This function groups semantically related states into clusters and assigns each state to a node in the semantic flow graph. These clusters provide high-level semantic abstractions, improving interpretability and aiding analysis.

By formalizing semantic flows, we can better understand and visualize how ML systems progress through latent spaces during execution. This framework facilitates the comparison of system behaviour under different inputs or configurations, revealing similarities and differences in their logical and semantic decisions. We argue such insights can be useful for testing, optimization, and debugging, enabling targeted improvements in system behaviour.

In the following, we present concrete examples of semantic flows and explore their practical applications.

3.1 Semantic Flow of Image Classifiers

Consider the visualisation of semantic flow of a Convolutional Neural Network (CNN) trained to classify the CIFAR-10 benchmark [15]. Figure 4 visualizes the progression of semantic states that we extracted from the network's executions, with dots representing semantic states and their colour selected based on the network's predicted class such as birds, cats, ships, and aeroplanes.



(a) FC Layer 1 (b) FC Layer 2 (c) FC Layer 3

Figure 4: Semantic clusters observed at different layers of a CNN CIFAR-10 Classifier, visualised in 3D with t-SNE. Note that different classes are increasingly separated from each other as they pass layers, gradually forming more visually separated semantic clusters. We argue that such separations can be likened to decision branching in programs.

The unit of analysis is the layers of the CNN and the semantic states are derived from the activation values in a layer during classification. For each input image, the network generates activations at successive layers, which reflect the system's internal state at each step. We use the t-SNE algorithm to map these layer-wise activation values (execution data) into semantic states. Figures 4a, 4b, and 4c show the semantic states from the three final fully connected layers of the network, each one reduced to (3D) latent spaces using t-SNE. As the input progresses through the layers, semantic states for each class become increasingly distinct, forming clusters by class label in the latent spaces. This separation can be seen to correspond to branching in the network's classification logic, as shown for traditional code in Figure 2. Note that here we did not visualise the flow between the three latent spaces that would be needed to show the flow graph. Similarly, the aggregation function was trivial since it is only based on ground truth labels.

The clustering of semantic states we can observe aligns with findings by Rauber et al. [17], who observed that training improves class separation in latent space. Yosinski et al. [26] similarly demonstrated how visualizing activations can help reveal the role of convolutional layers. We build on these insights, and posit that the progression of internal activations through layers mirrors branching logic in traditional programs, where semantic flow captures how the network separates image classes for accurate classification.

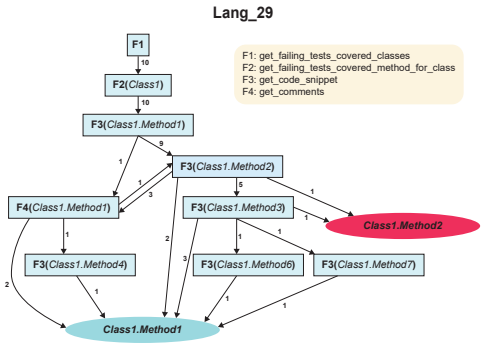


Figure 5: LLM Inference Graph by Kim et al. [14] of FL inferences of AutoFL [11] for the bug Lang-29 of Defects4J [9]

3.2 Semantic Flow of LLM Agents

LLM agents provide an intuitive context for semantic flow analysis. Their agentic workflows naturally define the ordering of information (akin to control flow), while their generated outputs encapsulate semantic information. The AutoFL system [10], previously introduced, can serve as an illustrative case.

In AutoFL, semantic states are derived from the system's inference steps, where each step corresponds to a function call invoked by the LLM during fault localization. The system's execution sequence forms the basis for semantic flow analysis. Each function call is represented as a semantic state within a single latent space (shared between all steps), consistent with the LLM Inference Graph (LIG) framework by Kim et al. [14]. Semantic states are encoded discretely using one-hot representations, ensuring identical function calls with the same arguments map to the same state. Figure 5 visualizes an LIG derived from 10 AutoFL executions, showing how function calls progress through the workflow. The semantic states corresponding to identical function calls across executions are merged, with edge weights reflecting the frequency of transitions between them. For example, AutoFL's multiple executions for the same bug aggregate results for self-consistency [21]. In Figure 5, some executions correctly localize the bug (reaching the blue node), while others fail (reaching the red node). These clusters and transitions clarify the logical pathways taken by the system.

The LIG depicted in Figure 5 highlights AutoFL's decision-making at each step, contrasting with traditional control flow (Figure 3). The LIG is a specific instance of a semantic flow graph, where semantic information is discretely encoded by function call types and the graph has been further compacted by counting and annotating with the number of node transitions. While in this case the semantic flow embedding function was manually selected we could alternatively use an external LLM encoder model to embed the natural language inputs/outputs into a latent space [18] or directly using the hidden states of the LLM of the agentic system. This analysis demonstrates how Semantic Flow Graph (SFG) analysis can provide a structured, interpretable view of LLM agent workflows.

4 Properties and Applications of Semantic Flow

Fundamentally, semantic flow aims to capture and represent the executions of ML-based systems at multiple latent spaces. Once we

capture semantic flow of a set of accepted executions, it can be used to measure various properties of executions of ML-based systems. For example, we can imagine a diversity-aware testing technique for LLM-based agents that can select and prioritise inputs that result in the most diverse set of executions. We discuss potential applications of semantic flow in this section.

4.1 Semantic and Control Flow Graphs (SaCFGs)

Semantic Flow Graphs (SFGs) represent execution flows as graphs, where nodes correspond to clusters of related states in a latent space, and edges indicate their sequential appearance. While these graphs can be helpful tools in themselves we also propose hybridizing SFGs with traditional Control Flow Graphs (CFGs) to create Semantic and Control Flow Graphs (SaCFGs) for analyzing hybrid systems that combine conventional software and machine learning (ML) components, such as large language models (LLMs). As one example, SaCFGs can enable defining and evaluating coverage criteria that span diverse components in complex systems.

Unlike static CFG analysis, constructing SaCFGs requires a multi-step process due to the stochastic nature of semantic state clustering, which depends on system executions. Once clusters are formed, new states can be mapped either as discrete assignments (e.g., ϵ-coverage: states within distance ϵ of a cluster) or using finer-grained distances to measure partial coverage across multiple clusters. This hybrid framework offers a powerful tool for exploring semantic and control flow interactions in ML-software systems, advancing both analysis and practical applications.

4.2 Measuring Out-of-Distribution-ness of Executions

The statistical nature of semantic flows, particularly semantic clusters, provides a foundation for quantifying the out-of-distribution-ness of ML-based system executions. We argue that Surprise Adequacy (SA) [12, 13], a widely studied test adequacy metric for deep neural networks (DNNs), represents a specific instance of such a measure. SA evaluates how much of an outlier a new semantic cluster is relative to a reference cluster, with research showing that higher levels of surprise in inputs correlate with a greater likelihood of unexpected or buggy behavior in DNNs.

Extending this concept, semantic flow analysis enables out-of-distribution measurements for longer and more complex executions, such as those from systems involving large language model (LLM) agents. By considering multiple semantic states sampled at various points during execution, we can evaluate the overall degree of out-of-distribution-ness for an entire system run. Similar to the hybrid SaCFG framework, there is flexibility in choosing whether to discretize states into clusters or to leverage continuous distances in the latent space, allowing for nuanced modeling and analysis.

4.3 Debugging ML-based Systems

Once we connect out-of-distribution-ness, unexpected behaviour, and coverage in SFG (or SaCFG), we can consider debugging techniques for traditional programs. For example, we may be able to perform something similar to Spectrum Based Fault Localisation [22]: if buggy executions of an LLM-based agent tend to *cover* a specific semantic cluster, while normal executions do not, we may expect that the specific reasoning step relevant to that semantic cluster is the root cause of the problem, and that the related prompt needs to be improved. Alternatively, specific patterns of flow, over multiple clusters, might indicate faulty (multi-step) "reasoning".

4.4 Predicting Execution Results

Kim et al. [14] initially proposed LLM Inference Graph as a way to predict whether a set of LLM-based agent executions can produce a correct answer: authors have trained a Graph Convolutional Network (GCN) that takes an LIG as an input and predicts the correctness of the final answer, achieving precisions of over 0.8. Such predictions are made feasible because LIG, a special case of SFG, encapsulates the entire behaviour of the agent. Since LLMs require very large amount of resources, we argue that such predictions can be very valuable as long as they are reasonable accurate. Further, if accurate predictions can be made using partial SFGs (i.e., SFGs constructed from incomplete executions), we may be able to force early-termination of executions that are not likely to succeed.

4.5 Interpretability and Explainability

The choice of latent spaces, $L_1, \ldots, L_n$, can significantly enhance the interpretability and explainability of ML-based system behavior. Semantic embeddings map specific execution steps, such as an LLM-based agent's response, into latent spaces. We argue that these embeddings can also be tailored to capture domain-specific abstract properties of the responses. For example, in an LLM-based agent providing customized health advice, user inputs could be embedded using a generic language model or into a specially designed latent space. Such a space might distinguish whether the input represents self-reflection, factual information, or a rebuttal to the agent's response. By designing latent spaces to highlight such high-level properties, we can potentially improve the interpretability and explainability of the system's behavior.

5 Conclusion

We introduce the concept of semantic flow: the flow of semantic information, represented as vectors in (a) latent space(s), traversed by an ML-based system during execution. This framework applies to systems ranging from individual deep neural networks (DNNs) to complex large language model (LLM) agents, with executions captured and visualized through Semantic Flow Graphs (SFGs).

We discuss properties of this new concept and highlight how semantic flow enables the adaptation of dynamic analysis techniques for ML-based systems. By using semantic flow to represent executions, we aim to improve the quality, reliability, and understanding of these systems, advancing their analysis and development.

Acknowledgements

Shin Yoo, Somin Kim, and Naryeong Kim are supported by the National Research Foundation of Korea grant (RS-2023-00208998), the Engineering Research Center Program (RS-2021-NR060080), and the Institute of Information & Communications Technology Planning & Evaluation (IITP) grant (RS-2022-II220995), all funded by Korean Government (MSIT). Robert Feldt acknowledges support from the Swedish Scientific Council (No. 2020-05272) and from the WASP-funded 'BoundMiner' project.

References

[1] 2024. Duolingo Max with GPT-4. https://blog.duolingo.com/duolingo-max/

[2] 2024. GPTRPG. https://gptrpg.net/en/discover

[3] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[4] A. Fan, B. Gokkaya, M. Harman, M. Lyubarskiy, S. Sengupta, S. Yoo, and J. M. Zhang. 2023. Large Language Models for Software Engineering: Survey and Open Problems. In *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*. IEEE Computer Society, 31–53. https://doi.org/10.1109/ICSE-FoSE59343.2023.00008

[5] Clement Farabet, Camille Couprie, Laurent Najman, and Yann LeCun. 2013. Learning hierarchical features for scene labeling. *IEEE transactions on pattern analysis and machine intelligence* 35, 8 (2013), 1915–1929.

[6] D. Feng, C. Haase-Schütz, L. Rosenbaum, H. Hertlein, C. Gläser, F. Timm, W. Wiesbeck, and K. Dietmayer. 2020. Deep Multi-Modal Object Detection and Semantic Segmentation for Autonomous Driving: Datasets, Methods, and Challenges. *IEEE Transactions on Intelligent Transportation Systems* (2020), 1–20.

[7] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, and B. Kingsbury. 2012. Deep Neural Networks for Acoustic Modeling in Speech Recognition: The Shared Views of Four Research Groups. *IEEE Signal Processing Magazine* 29, 6 (Nov 2012), 82–97. https://doi.org/10.1109/MSP.2012.2205597

[8] Robert Huben, Hoagy Cunningham, Logan Riggs Smith, Aidan Ewart, and Lee Sharkey. 2024. Sparse Autoencoders Find Highly Interpretable Features in Language Models. In *The Twelfth International Conference on Learning Representations*. https://openreview.net/forum?id=F76bwRSLeK

[9] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA 2014)*. ACM, New York, NY, USA, 437–440. https://doi.org/10.1145/2610384.2628055

[10] Sungmin Kang, Gabin An, and Shin Yoo. 2024. A Quantitative and Qualitative Evaluation of LLM-based Explainable Fault Localization. *Proceedings of the ACM on Software Engineering* 1, FSE (July 2024), 64:1424–1446.

[11] Sungmin Kang, Juyeon Yoon, Nargiz Askarbekkyzy, and Shin Yoo. 2024. Evaluating Diverse Large Language Models for Automatic and General Bug Reproduction. *IEEE Transactions on Software Engineering* 50, 10 (2024), 2677–2694.

[12] Jinhan Kim, Robert Feldt, and Shin Yoo. 2019. Guiding Deep Learning System Testing using Surprise Adequacy. In *Proceedings of the 41th International Conference on Software Engineering (ICSE 2019)*. IEEE Press, 1039–1049. https://doi.org/10.1109/ICSE.2019.00108

[13] Jinhan Kim, Robert Feldt, and Shin Yoo. 2022. Evaluating Surprise Adequacy for Deep Learning System Testing. *ACM Transactions on Software Engineering and Methodology* 32, 2 (June 2022), 1–29.

[14] Naryeong Kim, Sungmin Kang, Gabin An, and Shin Yoo. 2025. Lachesis: Predicting LLM Inference Accuracy using Structural Properties of Reasoning Paths. In *Proceedings of the 6th International Workshop on Deep Learning for Testing and Testing for Deep Learning (DeepTest 2025)*.

[15] Alex Krizhevsky. 2009. *Learning Multiple Layers of Features from Tiny Images*. Technical Report. University of Toronto.

[16] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *Nature* 521, 7553 (2015), 436.

[17] Paulo E. Rauber, Samuel G. Fadel, Alexandre X. Falcão, and Alexandru C. Telea. 2017. Visualizing the Hidden Activity of Artificial Neural Networks. *IEEE Transactions on Visualization and Computer Graphics* 23, 1 (2017), 101–110. https://doi.org/10.1109/TVCG.2016.2598838

[18] Nils Reimers and Iryna Gurevych. [n. d.]. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, Kentaro Inui, Jing Jiang, Vincent Ng, and Xiaojun Wan (Eds.). Hong Kong, China.

[19] Sepehr Sharifi, Donghwan Shin, Lionel C. Briand, and Nathan Aschbacher. 2023. Identifying the Hazard Boundary of ML-Enabled Autonomous Systems Using Cooperative Coevolutionary Search. *IEEE Transactions on Software Engineering* 49, 12 (2023), 5120–5138. https://doi.org/10.1109/TSE.2023.3327575

[20] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems*, I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc.

[21] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023. Self-Consistency Improves Chain of Thought Reasoning in Language Models. *CoRR* abs/2203.11171 (2023).

[22] W. E. Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A Survey on Software Fault Localization. *IEEE Transactions on Software Engineering* 42, 8 (August 2016), 707.

[23] Ming Yan, Ruihao Li, Hao Zhang, Hao Wang, Zhilan Yang, and Ji Yan. 2023. Larp: Language-agent role play for open-world games. *arXiv preprint arXiv:2312.17653* (2023).

[24] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. https://arxiv.org/abs/2405.15793

[25] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. In *Proceedings of the International Conference on Learning Representation (ICLR 2023)*.

[26] Jason Yosinski, Jeff Clune, Anh Mai Nguyen, Thomas J. Fuchs, and Hod Lipson. 2015. Understanding Neural Networks Through Deep Visualization. *CoRR* abs/1506.06579 (2015). http://arxiv.org/abs/1506.06579

[27] Zeyu Yun, Yubei Chen, Bruno A Olshausen, and Yann LeCun. 2021. Transformer visualization via dictionary learning: contextualized embedding as a linear superposition of transformer factors. *arXiv preprint arXiv:2103.15949* (2021).

[28] Quanshi Zhang, Ying Nian Wu, and Song-Chun Zhu. 2018. Interpretable Convolutional Neural Networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.