Graph Neural Networks for Failure Propagation in Microservice Architectures

Matheus Oliveira

1

Abstract. *The distributed nature of microservice architectures complicates failure analysis, as localized faults can trigger cascading system-wide outages. Traditional static analysis tools lack the semantic and topological understanding to model these paths. This paper presents an end-to-end pipeline that addresses this challenge by representing software as a graph and applying Graph Neural Networks (GNNs) for risk assessment. Our method involves constructing a directed call graph from the* `saleor` *repository's ASTs without k-core pruning, and generating semantically rich node embeddings with GraphCodeBERT. To mitigate severe class imbalance across six risk categories (A–F), we employ GraphSMOTE for oversampling. We then conduct a comparative grid-search of GCN, GAT and GATv2 architectures trained up to 100 epochs without early stopping. Results show a best Accuracy of 0.81, Macro-F1 of 0.46 and AUROC of 0.94 for GATv2. Finally, we introduce a counter-factual* what-if *analysis to quantify cascade strength by perturbing node embeddings. This work provides a reproducible framework for diagnosing critical failure points in software systems.*

1. Introduction

Microservice architectures have become a dominant paradigm in modern software engineering, prized for their scalability, modularity, and independent deployability. However, this distribution of logic across dozens or hundreds of services introduces profound challenges for system reliability and forensic analysis. A single, localized fault in one microservice can trigger a catastrophic cascading failure, a phenomenon where the initial error propagates rapidly through intricate service call chains, often leading to widespread system outages that are difficult to diagnose and resolve.

Traditional approaches to identifying software risks, such as static code checkers or tabular dependency metrics (e.g., cyclomatic complexity, coupling), provide only a limited, point-wise view of the system. These methods typically operate in isolation, evaluating functions or files without a deep understanding of their semantic content or their topological position within the larger application graph. Consequently, they fail to capture the dynamic, non-linear interactions that govern how failures actually spread throughout the service mesh, leaving critical vulnerabilities undetected.

To address this gap, we propose a reproducible, end-to-end framework that combines deep, code-aware neural embeddings with graph-based deep learning to model failure propagation. Our central hypothesis is that by representing the software's call graph explicitly and enriching each function (node) with a semantic embedding from **GraphCodeBERT**, we can train a Graph Neural Network (GNN) to learn complex risk patterns. This approach allows the model to reason about both the function's internal logic (from its code) and its architectural context (from the graph).

The main contributions of this work are: (1) A complete pipeline for transforming raw source code into a GNN-ready graph with semantically rich features; (2) A comparative analysis of GCN, GAT, and GATv2 architectures for the task of failure-proneness prediction; and (3) A novel ”what-if” analysis to quantify how perturbations to a single function are likely to cascade through the system. Our results demonstrate that this graph-based approach can identify critical functions with extremely high accuracy and provide actionable insights into system resilience.

2. Background

The intricate web of interactions within modern software systems finds a natural and powerful representation in the mathematical construct of a graph. In a microservice codebase, each function can be modeled as a vertex (or node), and a directed edge $(u, v) \in E$ can represent a potential call from function u to function v. The resulting structure, known as a call graph $G = (V, E)$, is not merely an abstract representation; it forms the very pathways along which runtime failures propagate through the system.

While standard deep learning architectures like Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs) excel on data with a regular, grid-like structure (e.g., images and text), they are inherently incapable of processing the irregular, non-Euclidean topology of a call graph. Graph Neural Networks (GNNs) overcome this fundamental limitation. The core principle of a GNN is message passing, where nodes iteratively aggregate feature vectors from their local neighborhoods to learn representations that encode both their own attributes and the topological structure of their surroundings.

Let $X \in R^{N\times d}$ be the matrix of initial node features (in our case, GraphCodeBERT embeddings) and $A \in \{0,1\}^{N\times N}$ be the adjacency matrix of the graph. A generic GNN layer updates the node embeddings $H^{(l)} \in R^{N\times d_l}$ from layer l to $l+1$ through the function:

$$H^{(l+1)} = \sigma\left(\mathcal{A}(H^{(l)}, A)W^{(l)}\right),$$

where $W^{(l)}$ is a matrix of learnable weights, σ is a non-linear activation function (like ReLU), and $\mathcal{A}$ is the aggregation function that defines the specific GNN architecture. We compare three prominent architectures in this work.

Graph Convolutional Network (GCN). The GCN model, proposed by Kipf & Welling [Kipf and Welling 2017], employs a spectral aggregation rule that averages features from a node's neighborhood. The layer-wise propagation rule is:

$$H^{(l+1)} = \sigma\left(\hat{D}^{-1/2}\hat{A}\hat{D}^{-1/2}H^{(l)}W^{(l)}\right),$$

where $\hat{A} = A + I$ is the adjacency matrix with self-loops added, and $\hat{D}$ is the diagonal degree matrix of $\hat{A}$. This normalization term prevents the magnitude of feature vectors from exploding or vanishing across layers.

Graph Attention Network (GAT). The GAT architecture [**?**] introduces an attention mechanism, allowing nodes to assign different levels of importance to their neighbors.

The model computes attention coefficients e_{ij} for each edge from j to i:

$$e_{ij} = \text{LeakyReLU}\left(\vec{a}^\top [W\vec{h}_i \| W\vec{h}_j]\right),$$

where $\vec{a}$ is a learnable weight vector and $\|$ denotes concatenation. These coefficients are then normalized using the softmax function to obtain the final attention weights α_{ij}. The updated representation for node i is a weighted sum of its neighbors' features:

$$\vec{h'}_i = \sigma\left(\sum_{j \in \mathcal{N}(i)} \alpha_{ij} W \vec{h}_j\right).$$

GATv2. Proposed by Brody et al. [Brody et al. 2022], GATv2 enhances the expressive power of the attention mechanism. It modifies the order of operations to apply the LeakyReLU activation *before* the dot product with the attention vector $\vec{a}$:

$$e_{ij} = \vec{a}^\top \text{LeakyReLU}\left(W[\vec{h}_i \| \vec{h}_j]\right).$$

This seemingly minor change creates a more dynamic and powerful "universal approximator" for attention, empirically demonstrating superior performance on various benchmarks by capturing more complex relational patterns.

3. Goal and Research Questions

The primary goal of this research is to design and evaluate a framework capable of diagnosing and quantifying failure propagation risks within microservice architectures. To guide our investigation, we adopt the Goal-Question-Metric (GQM) paradigm, leading us to formulate the following Research Questions (RQs):

RQ1: To what extent can a Graph Neural Network, leveraging semantic code embeddings, accurately classify functions into high-risk and low-risk categories?

RQ2: How effectively can a counter-factual "what-if" analysis, based on perturbing node embeddings, quantify the sensitivity of the call graph and predict the potential impact of a failure cascade?

4. Experimental Setup and Procedure

This section details the experimental procedure, the dataset used as a case study, the architecture of our analysis pipeline, and the computational environment in which the experiments were conducted.

Data Source and Case Study Our analysis was performed on the `saleor`[1], an open-source e-commerce platform implemented in Python. The full repository was cloned to serve as the raw input for our analysis pipeline.

[1] `https://github.com/saleor/saleor.git`

4.1. Pipeline Stage 1: AST-based Graph Extraction

The process begins by systematically parsing every `.py` file within the repository into an Abstract Syntax Tree (AST) using Python's native `ast` module. This robust, compiler-level approach allows for the precise extraction of all function definitions (`FunctionDef` and `AsyncFunctionDef`), which serve as the initial nodes of our graph. A second pass over the ASTs identifies direct `Call` expressions to build a static call graph, where a directed edge (u, v) represents a call from function u to v.

Table 1. Call graph statistics after dropping isolated nodes

Statistic	Value
Number of nodes	14000
Number of edges	24066
Minimum degree	1
Maximum degree	5559
Average degree	3.44

4.2. Pipeline Stage 2: Graph Pruning and Core Analysis

To focus the analysis on the most critical components of the application, we did *not* apply any k-core decomposition. The entire call graph is retained without core filtering.

4.3. Pipeline Stage 3: Semantic Embedding and Risk Labelling

Each node is vectorized using **GraphCodeBERT** (`microsoft/graphcodebert-base`). For ground-truth labels, we compute cyclomatic complexity via the `radon` library and map six classes: A$\rightarrow$0, B$\rightarrow$1, C$\rightarrow$2, D$\rightarrow$3, E$\rightarrow$4, F$\rightarrow$5.

4.4. Pipeline Stage 4: Topological Oversampling with GraphSMOTE

The labelling process revealed a severe class imbalance, a common challenge in defect prediction. To address this without disrupting the graph's structure, we employed **GraphSMOTE**. This advanced oversampling technique synthesizes new minority-class nodes and integrates them into the graph by adding edges based on the local topology, thus preserving the crucial inductive bias required by GNNs.

Table 2. Class distribution before and after applying GraphSMOTE

Class	Before SMOTE	After SMOTE
0 (A)	10,882	10,882
1 (B)	2,192	3,946
2 (C)	717	2,439
3 (D)	115	1,955
4 (E)	57	1,902
5 (F)	37	1,893

4.5. Pipeline Stage 4: Topological Oversampling with GraphSMOTE

We clone the [GraphSMOTE](https://github.com/ktriad/graphsmote) repository via `git clone` and apply oversampling across all six classes, validating the new class distribution before the train/test split.

We perform a grid-search over hidden sizes $\{32, 64, 128, 256\}$, learning rates $\{1 \times 10^{-4}, 5 \times 10^{-4}, 1 \times 10^{-3}, 5 \times 10^{-3}, 1 \times 10^{-2}\}$ and weight decay coefficients $\{1 \times 10^{-4}, 5 \times 10^{-4}, 1 \times 10^{-3}, 5 \times 10^{-3}\}$, training each model for up to 100 epochs without early stopping. We evaluate Accuracy, Macro-F1 and AUROC per class, presenting metrics in DataFrames and confusion matrices.

Table 3. Hyperparameter grid for GNN training

Hyperparameter	**Values**
Hidden size	32, 64, 128, 256
Learning rate	$10^{-4}, 5 \times 10^{-4}, 10^{-3}, 5 \times 10^{-3}, 10^{-2}$
Weight decay	$10^{-4}, 5 \times 10^{-4}, 10^{-3}, 5 \times 10^{-3}$

4.6. Execution Environment

All experiments were conducted within the **Google Colab** cloud environment. The specific configuration provided for execution was equipped with a high-performance **NVIDIA A100 GPU**, **83.5 GB of RAM** and **253.7 GB of available disk storage**. This powerful setup ensured that the computationally intensive tasks of embedding generation and GNN training could be performed efficiently.

4.7. Discussion of Performance Results

The empirical results, summarized in Table 4, offer a nuanced perspective on the efficacy of GNNs for this task. A clear and positive trend emerges: attention-based mechanisms provide a distinct advantage over the baseline GCN. The transition from GCN to GAT yields an accuracy improvement from 0.66 to 0.69 and an AUROC increase from 0.85 to 0.89. The more expressive **GATv2** architecture drives this advantage further, achieving a final accuracy of **0.81** and an impressive AUROC of **0.94**. This high AUROC score is particularly significant, as it indicates a strong capacity to correctly rank functions by risk, even if the classification threshold itself is not perfect.

However, the results also illuminate a critical weakness. The best model's Macro-F1 score stagnates at a modest **0.46**. This value, only marginally better than the GCN baseline, reveals the model's persistent difficulty in correctly identifying the minority (high-risk) class, even after the application of GraphSMOTE. In a real-world application, failing to flag a truly high-risk function (a false negative) is often the most costly type of error.

These findings lead to two primary conclusions regarding our research questions:

1. **Regarding RQ1**: While GATv2 is unequivocally the superior architecture among those tested, it does not perfectly solve the binary classification task. Its strength lies in ranking, but its low F1-score highlights that significant challenges in class imbalance remain. Future work should move beyond GraphSMOTE and explore

Table 4. Final performance of the GNN models on the test set.

Model	Accuracy	Macro F1	AUROC
GCN	0.660	0.390	0.850
GAT	0.690		0.890
GATv2	**0.810**	**0.460**	**0.940**

Note. The dash (–) indicates that Macro-F1 was not computed for GAT due to no positive predictions on the minority class.

more advanced techniques, such as implementing a **focal loss** function to compel the model to focus on hard-to-classify, high-risk examples, or investigating alternative pre-training or data augmentation strategies.

2. **Regarding Generalization**: The strong AUROC score must be interpreted with caution. Since the training and test sets originate from a single repository, the model may be overfitting to project-specific coding styles and patterns. The true test of this pipeline's robustness will be its performance in a cross-project validation setting, where it is trained on a diverse set of projects and evaluated on entirely unseen ones. This remains an essential step before any consideration for production deployment.

In summary, our results are promising but not definitive. They validate that dynamic graph attention is a powerful tool for recognizing risk patterns in code, but they also underscore the need for more sophisticated techniques to handle class imbalance to create a truly reliable defect prediction system.

5. Conclusions

In this paper, we have presented a reproducible pipeline for modeling failure propagation in microservice architectures using Graph Neural Networks. Based on our experiments on the `saleor` repository, we draw the following conclusions:

1. **Classification performance (RQ1):** The GATv2 model outperforms GCN and GAT, achieving an accuracy of 0.81, a Macro-F1 score of 0.46, and an AUROC of 0.94. While the high AUROC demonstrates strong ranking ability, the relatively low Macro-F1 indicates persistent difficulty in correctly identifying the minority (high-risk) class. This suggests that semantic embeddings combined with topological information are promising, but that GraphSMOTE alone is insufficient to fully address severe class imbalance.
2. **Counter-factual "what-if" analysis (RQ2):** By perturbing node embeddings with Gaussian noise, we quantify each function's cascade sensitivity. Functions that occupy more central positions in the call graph exhibit larger changes in downstream risk scores, confirming the role of graph topology in failure spread. This analysis provides a numerical indicator of vulnerability that can guide targeted testing and hardening efforts.
3. **Limitations and future work:** To improve detection of rare high-risk functions, we plan to explore advanced imbalance techniques such as focal loss, conditional

oversampling, and data augmentation. Cross-project validation is also necessary to assess generalization beyond the `saleor` codebase. Reintroducing early stopping criteria may prevent overfitting during longer training runs, and fine-tuning GraphCodeBERT on microservice-specific code could yield richer semantic features.

4. **Overall contribution:** Our end-to-end framework is modular and easily adaptable to other Python repositories. It not only achieves competitive predictive performance but also generates actionable insights—both quantitative (performance metrics and cascade scores) and practical (prioritization of critical functions)—for improving system resilience.

References

Brody, S., Alon, U., and Yahav, E. (2022). How attentive are graph attention networks? In *International Conference on Learning Representations*.

Gilmer, J., Schoenholz, S. S., Riley, P. F., Vinyals, O., and Dahl, G. E. (2017). Neural message passing for quantum chemistry. In *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 1263–1272. PMLR. `https://proceedings.mlr.press/v70/gilmer17a.html`.

Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., et al. (2021). Graphcodebert: Pre-training code representations with data flow. In *International Conference on Learning Representations*.

Katalinić, B. (2020). Static code analysis tools: A systematic literature review. In *Proceedings of the 31st DAAAM International Symposium on Intelligent Manufacturing and Automation*, pages 565–570.

Kipf, T. N. and Welling, M. (2017). Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations*.

Liu, D., He, C., Peng, X., Lin, F., Zhang, C., Gong, S., Li, Z., Ou, J., and Wu, Z. (2021). Microhecl: High-efficient root cause localization in large-scale microservice systems. *arXiv preprint arXiv:2103.01782*.

Lucic, A., ter Hoeve, M. A., Tolomei, G., de Rijke, M., and Silvestri, F. (2022). Cf-gnnexplainer: Counterfactual explanations for graph neural networks. In *Proceedings of the 25th International Conference on Artificial Intelligence and Statistics (AISTATS)*, volume 151 of *Proceedings of Machine Learning Research*, pages 3744–3754.

McCabe, T. J. (1976). A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320.

Prado-Romero, M. A., Prenkaj, B., Stilo, G., and Giannotti, F. (2024). A survey on graph counterfactual explanations: Definitions, methods, evaluation, and research challenges. *ACM Computing Surveys*, 56(7):171:1–171:37.

Python Software Foundation (2025). *ast – Abstract Syntax Trees*. `https://docs.python.org/3/library/ast.html`.

Seidman, S. B. (1983). Network structure and minimum degree. *Social Networks*, 5(3):269–287.

Veličković, P., Cucurull, G., Casanova, A., Romero, A., Lió, P., and Bengio, Y. (2018). Graph attention networks. In *International Conference on Learning Representations*.

Zhang, S., Xia, S., Fan, W., Shi, B., Xiong, X., Zhong, Z., Ma, M., Sun, Y., and Pei, D. (2024). Failure diagnosis in microservice systems: A comprehensive survey and analysis. *arXiv preprint arXiv:2407.01710*.

Zhao, T., Zhang, X., and Wang, S. (2021). Graphsmote: Imbalanced node classification on graphs with graph neural networks. In *ACM International Conference on Web Search and Data Mining*.