PECAN - Programming Encoder Language Classification Analysis Network

by

Ibrahim Alam

Undergraduate honors capstone under the direction of

Dr. James Ghawaly

Division of Computer Science & Engineering

Submitted to the LSU Roger Hadfield Ogden Honors College in partial fulfillment of the Upper Division Honors Program.

April, 2026

Louisiana State University
& Agricultural and Mechanical College
Baton Rouge, Louisiana

Abstract

Programming language identification is essential for code analysis, security scanning, and developer tooling, but existing methods fail across three broad categories: environments where file metadata is unavailable or unreliable, such as digital forensics and data recovery; situations where only partial code segments exist, such as snippets embedded in web content and documentation; and applications that require automatic analysis outside of traditional repository structures, such as malware analysis and large-scale data mining from unstructured sources. Current tools like GitHub Linguist rely on file extensions and heuristics that assume full-length and formatted files within repository contexts, while ML-based approaches, like Guesslang, were trained on complete source files and a limited number of languages. PECAN (Programming Encoder Classification Analysis Network) addresses these limitations by using a fine-tuned encoder-only transformer (CodeBERT) to classify programming languages directly from raw code fragments. By collecting over 42 million samples across over 300 languages from The Stack v1.1, PECAN evaluates a fine-tuned CodeBERT model with respect to accuracy, efficiency, and robustness, offering systematic insights into accuracy-efficiency tradeoffs for metadata-free deployment in resource-constrained environments.

1. Introduction

Programming language identification (PL-ID) is a foundational capability for software tooling, enabling static analysis, syntax-aware editing, code search and indexing (Allamanis et al., 2018). Accurate language labeling is also important in security and forensic workflows for malware triage and security scanning where analysts need to quickly identify unknown and potentially dangerous code before further investigation (Roussev et al., 2013). As code increasingly appears outside curated repositories, PL-ID systems must reliably operate on noisy and diverse inputs rather than assuming ideal full-file contexts (Baltes et al., 2018; Pimentel et al., 2019; Bavota, 2016).

Current PL-ID systems are frequently challenged in three recurring settings. The first is environments where metadata may be unavailable or unreliable, such as in corrupted file systems or extension-stripped data (Mittal et al., 2020). The next is situations where only partial code fragments exist, such as chat platforms or clipped log snippets (Dietrich et al., 2019). Third is applications requiring automatic analysis of code outside traditional repository structures such as large-scale mining from unstructured data sources (Bavota, 2016). These challenge settings require metadata-free PL-ID systems that remain reliable and consistent under context loss.

Existing methods often assume conditions that do not hold in these environments. Widely used practical systems such as GitHub Linguist and Guesslang are effective in repository-centric workflows. Prior work has shown that metadata-based and heuristic approaches degrade under noisy or metadata-free inputs (Fratantonio et al., 2025). Similarly, pretrained code representation models such as CodeBERT, GraphCodeBERT, and UniXcoder exhibit strong performance in tasks such as code search, summarization, translation, and code detection (Feng et al., 2020; Guo et al., 2020; 2022), but they also do not target metadata-free programming language identification on fixed length byte fragments. To our knowledge, large-scale evaluation of encoder-only transformers for programming language identification in this exact

setting remains limited.

This paper presents PECAN (Programming Encoder Classification Analysis Network), a metadata-free framework for programming-language identification from fixed-length raw code fragments. Lightweight encoder-only transformers provide robust PL-ID performance in non-ideal, real-world settings when trained and evaluated under consistent procedures. We construct supervised PL-ID benchmarking using The Stack v1.1 at scale with 42M+ samples across 300+ languages (Kocetkov et al., 2022). We evaluate multiple encoder families under identical training and evaluation conditions along with reporting transparency metrics such as raw/canonical scoring, abstention reliability with unknown and answered rates, and accuracy-efficiency trade-offs relevant to deployment. We benchmark PECAN against heuristic (GitHub Linguist) and application-specific deep learning (Guesslang) PL-ID baselines as well as GPT-OSS under a unified protocol.

2. Related Works

2.1. Pretrained code representation models

Pretrained code models have considerably improved performance on code intelligence tasks (Niu et al., 2023). CodeBERT uses bimodal NL-PL representations for code search and code documentation generation (Feng et al., 2020). GraphCodeBERT utilizes data-flow-aware pretraining to improve tasks including code search, clone detection, translation, and refinement (Guo et al., 2020). UniXcoder builds upon this line of work with a unified architecture leveraging AST and comment modalities for understanding and generation tasks (Guo et al., 2022). Despite these advances and capabilities, these works do not focus on metadata-free PL-ID from fixed-length byte-level fragments, which is the setting that PECAN targets.

2.2. Programming language identification literature

Current PL-ID systems span practical rule/heuristic tools and learned models. In particular, GitHub Linguist (heuristic) and Guesslang (application specific deep learning) are widely used practical baselines, while research systems such as DeepSCC and PLangRec explore snippet-oriented identification, and FiFTy studies fragment classification in an adjacent forensic setting (GitHub, Inc., 2024; Somda, 2021; Mittal et al., 2020). Snippet-oriented identification such as DeepSCC or PlangRec show the short-context identification as feasible, while FiFTy demonstrates fragment classification in an adjacent forensic setting with CNN-based methods (Yang et al., 2021; Rodriguez-Prieto et al., 2025). However, broad-language transformer benchmarking for metadata-free fixed-length byte-level code fragments remains relatively unexplored, especially at the scale and language variety used in PECAN.

2.3. Data and benchmark context

Primarily, The Stack has been used as a large-scale corpus for code model pretraining, but its breadth also enables supervised benchmarking when paired with a control split and labeling pipelines (Kocetkov et al., 2022). PECAN contributes this supervised PL-ID evaluation concept by combining large-scale data organization with a uniform training and evaluation framework with a large set of languages and samples rather than relying on unrelated task-specific benchmarks.

2.4. LLM inference reliability

For large instruction-reliant baselines such as GPT-OSS, inference reliability is a methodological concern along with raw accuracy (Agarwal et al., 2025). In constrained-label classification settings, responses may fail due to truncation, token limits, or formatting inconsistencies unless structured decoding is enforced (Tam et al., 2024;

Willard & Louf, 2023). We therefore treat reliability as part of the evaluation protocol design by enforcing structured outputs, recording abstentions explicitly, and report coverage-aware metrics (`unknown_rate`, `answered_rate`, and answered-only accuracy) along with conventional accuracy and F1 scores. This separates model classification quality from output-format compliance artifacts.

3. Methodology

3.1. Task Definition

We construct programming language identification (PL-ID) as a multiclass classification problem on fixed-length code fragments. Given an input code snippet x, the model or tool predicts one label y in the set Y, where Y is the full programming-language label set. Our setting is explicitly metadata-free, meaning the model only receives raw code content as input with no file name, extension, repository path, or other contextual metadata. To mimic realistic partial-observation environments such as logs, recovered/corrupted fragments, or embedded snippets, each sample is treated as a fixed-length code fragment rather than a complete source file.

3.2. Dataset Origin, Splitting, and Leakage Control

For PECAN experiments, we construct our dataset from The Stack v.1.1, a large-scale permissively licensed source code corpus (Kocetkov et al., 2022). We used a pinned commit hash to ensure the corpus is fixed and reproducible. The Stack v1.1 provides deduplicated source files spanning 358 programming languages. Of these, we retained 319 languages that had sufficient data after applying a minimum chunk count threshold.

We generate deterministic, precomputed split artifacts from the source corpus using a download and partitioning split. The splitter loads the dataset source partition used for experimentation (split="train"), then performs a two-stage deterministic split with seed 42. First, 20% of the data is held out as a test set (test_size=0.2). Second, the remaining training bucket (80% of the original) is split into new training and validation sets, with 15% allocated towards validation (val_fraction=0.15). This yields an overall 68/12/20 train/validation/test ratio.

Splits are stratified by language when feasible (stratify_threshold=1000) and fall back to deterministic non-stratified splitting if stratification splits are not satisfiable. The resulting artifacts are saved to the disk as train_split, val_split, and test_split, along with metadata and row-index files. These artifacts are reused across all runs to ensure reproducibility and prevent train/test contamination that could occur at runtime re-splitting.

In split artifacts used for this study, the final counts are 28,623,869 train, 5,051,271 validation, and 8,418,785 test samples (42,093,925 total).

3.3. Model Family

We fine-tune and evaluate an encoder-only transformer model commonly used in code and multilingual representation learning, CodeBERT (Feng et al., 2020). The model is adapted to a multiclass classification interface using a task-specific classification head over the shared programming language label dataset Y. Inputs are tokenized to fixed-length sequences (max_length=512) with truncation and max-length padding for stable batched training. We do not alter the model backbone, but rather we use the standard sequence-classification formulation (task head + label mapping) to compare model families under a common PL-ID protocol, input setting, and evaluation pipeline.

3.4. Training Protocol

Model training is performed with the Hugging Face Trainer API in a unified pipeline using model-specific sweep-selected hyperparameters. We first run W&B sweeps on benchmark subsets to identify strong configurations for each archi-

tecture (e.g. learning rate, weight decay, gradient accumulation). For full-scale training, we use the best hyperparameter sweep configuration per model rather than re-sweeping at full scale to preserve computational resources while maintaining consistent model selection.

We perform convergence-oriented training by combining a maximum epoch cap with early stopping. Specifically, we set a maximum number of epochs as well as a stop when validation performance fails to improve beyond a minimum threshold for a fixed-patience window. This prevents both undertraining (too few fixed epochs) and overtraining (continuing after a plateau). We load the best checkpoint at the end-of-training according to validation accuracy, and we log training hyperparameters and runtime metrics (optimizer schedule, batch configuration, seed, and evaluation outputs) for reproducibility. When W&B logging is enabled, system/hardware metadata is also captured automatically.

Because hardware differs across model sweeps, we preserve each model's sweep validated hyperparameters for primary comparisons rather than forcing identical batch size across all architectures. This reflects each model's best performance under its own stable operating point. Primary comparisons use each model's sweep-selected stable operating point under the same data split and evaluation protocol.

3.5. Evaluation Pipeline

All evaluators run through a shared pipeline on a common held-out set. By default, evaluation loads the precomputed test artifact mentioned in 3.3 rather than re-splitting at runtime, which helps avoid train/test leakage, preserves compatibility across runs, and avoids additional hardware overhead. Results are exported as machine-readable summary CSV files along with per-model artifacts.

The primary metrics used to measure effectiveness are accuracy, macro-F1, and weighted-F1. Accuracy is retained and used for standard comparison between models and comparability with prior classification benchmarks. Macro-F1 is emphasized because it reflects balanced performance across the full label space, including low-support and less-represented languages. Weighted-F1 captures aggregate behavior under class-frequency bias.

Metric reporting (accuracy, macro precision/recall/F1, and weighted precision/recall/F1) is two-pronged. First we report canonical (alias-aware) metrics after label normalization through a runtime alias map that accounts for naming differences among models and the dataset. Second, we report raw metrics which use strict lowercase label matching before any normalization. In this approach, the field accuracy represents canonical accuracy while raw metrics are reported separately as accuracy_raw. Reporting both views makes label-mapping assumptions clear and avoids distortion from normalization.

For LLM-metric reporting (GPT-OSS) the pipeline also reports abstention and coverage statistics (unknown_rate, unknown_count, answered_rate, answered_count, accuracy_answered_only) to capture reliability beyond top-line accuracy. Predictions normalized to unknown are treated as abstention and answered-only accuracy is calculated only using non-abstained outputs. We also log runtime efficiency metrics, including avg_inference_time and total_inference_time, so effectiveness, reliability, and latency are evaluated together and analyzed during a single evaluation period.

In addition to encoder and LLM baselines, the same evaluation pipeline supports traditional language-identification baselines, including Guesslang and GitHub Linguist, allowing all methods to be compared on the same held-out split and metric suite.

3.6. Large-LLM Baseline Protocol

Our large-LLM baseline utilizes GPT-OSS (120B) served through a vLLM backend in OpenAI-compatible server mode and evaluated through the shared PECAN pipeline via a base vLLM URL. Evaluation is processed in dataset chunks (e.g. 1000 samples per chunk), and requests are sent through the OpenAI-compatible chat path with client-side concurrency. At serving time, the vLLM backend performs its own internal batching and scheduling on incoming requests by the pipeline.

Inference uses schema-constrained structured decoding by default so outputs follow compact JSON with a single language field. Reliability and latency controls include an openai-max-tokens parameter, per-request-timeout parameter, as well as optional prompt truncation and reasoning hints. Constraints on output tokens are enforced to combat a high volume of GPU usage from long model outputs while allowing the model to reason at an efficient level. Because reasoning-capable models can abstain under these strict constraints, we optionally run an unknown-only structured retry pass with separate retry token, timeout, and concurrency limits.

GPT-OSS results are reported with the same canonical/raw scoring and abstention-aware reliability metrics used for all models (unknown_rate, answered_rate, accuracy_answered_only), enabling a direct comparison to the encoder-only baselines under a unified evaluation protocol.

3.7. Statistical and Practical Comparison Strategy

Primary comparison is based on canonical accuracy and macro-F1, with weighted-F1 and raw metrics as supporting views. Efficiency is reported using average inference time and observed throughput. Reliability is jointly assessed via unknown/answered metrics to avoid overstating performance when abstentions are non-negligible. We report model rankings under this multi-metric perspective to reflect deployment-relevant trade-offs among correctness, speed, and output reliability.

3.8. Error Analysis Protocol

To complement aggregate benchmark metrics, this study includes a structured post-hoc error analysis stage. Error analysis operates on per-sample prediction records exported by the shared evaluation pipeline, allowing each model's mistakes to be examined under the same fixed test artifact and label normalization policy used for primary scoring. This ensures that diagnostic findings are directly comparable across baseline families.

The quantitative component computes confusion statistics from error cases, including top confusion pairs (true_label $\rightarrow$ predicted_label) and per-language error-rate summaries. In addition, cross-model summaries are generated to compare how concentrated each model's dominant confusion channel is. These outputs identify whether errors are diffuse or repeatedly concentrated in specific language boundaries, which is not visible from accuracy/F1 alone.

The qualitative/diagnostic component assigns sampled error cases to failure settings: partial_snippet, non_repo_context, and ambiguous_or_overlap. When the total number of errors is large, a fixed-size sampled subset is used for tagging to keep processing tractable; sampling configuration and seed are logged in analysis metadata for reproducibility. Resulting artifacts (CSV summaries, markdown reports, and comparison plots) are used to interpret failure mechanisms and guide future mitigation, but do not alter primary effectiveness/reliability/runtime benchmark scores reported in the main evaluation.

4. Results and Analysis

4.1. Evaluation Setup Recap

In order to ensure the same examples are used and uniform post-processing logic, every baseline is

scored based on results that are computed under a fixed, precomputed test split and a shared evaluation pipeline. This protocol minimizes methodological variance so that performance differences reflect model behavior rather than evaluation differences. In particular, each run utilizes the same test artifact, canonicalization map, error analysis, and summary/export path, allowing direct comparison between heuristic and application-specific deep learning baselines, GPT-OSS, and CodeBERT-PECAN.

We report five metric categories. Effectiveness is summarized by accuracy, macro-F1, and weighted-F1. Transparency is maintained through dual reporting of raw (strict-label) and canonical (alias-aware) scores. Reliability/coverage is captured through unknown rate, answered rate, and accuracy of answered samples. Computational behavior is captured through average and total inference times. Error analysis is reported as a post-hoc diagnostic using per-sample predictions, including top confusion pairs, per-language error rates, and failure-setting breakdowns.

4.2. Overall Performance

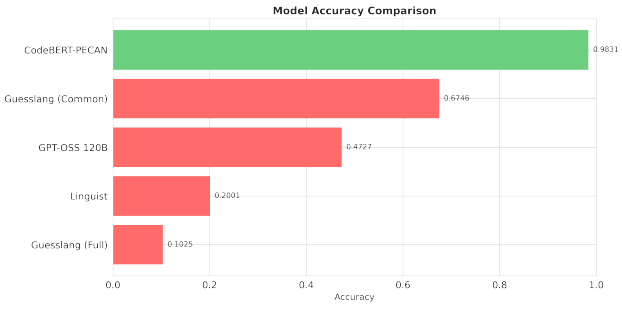

Figure 1. Effectiveness comparison under the shared metadata-free protocol. CodeBERT-PECAN achieves the highest accuracy, macro-F1, and weighted-F1 across evaluated baselines.

Figure 1 demonstrates the primary effectiveness metrics for each model (accuracy, macro-F1, weighted-F1) under the shared metadata-free evaluation pipeline. In this thesis setting, CodeBERT-PECAN achieves the highest overall performance across all three metrics, indicating a strong aggregate correctness along with robust class-balanced behavior. Compared with heuristic and application-specific deep learning baselines as well as GPT-OSS, there is a significant performance gap under strict snippet-only constraints, reinforcing that CodeBERT-PECAN (fine-tuned encoder) is highly effective for this task setting.

This result is particularly important under the evaluation conditions where there is no file-extension metadata, no repository context, and fixed-length fragment inputs. Within this setting, the model must rely on the structure and content of the given snippet with no external hints. This figure establishes the baseline performance comparison for all models. To interpret these results more clearly, we next examine how label normalization affects reported metrics.

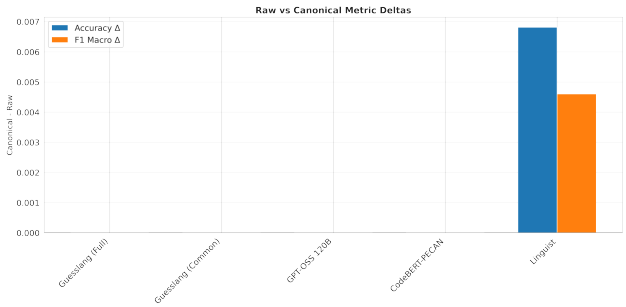

Figure 2. Canonical-minus-raw deltas for accuracy and macro-F1. Small deltas for most models indicate conclusions are not driven by alias normalization artifacts.

To ensure metric transparency, results are reported in both canonical and raw forms. Canonical scoring applies alias-aware normalization (for example, equivalent language names mapped to a shared canonical label), while raw scoring uses strict lowercase label matching between the raw prediction and original label. Reporting both metrics avoids hidden gains from relabeling while ensuring a reproducible interpretation among baselines with different output behavior.

In these runs, Figure 2 shows canonical vs raw deltas or differences are small for most models, and conclusions are largely unchanged. This indicates that overall ranking depends on model performance rather than their string labels. Where deltas are present, they are minimal and can be in-

Table 1. Reliability and coverage comparison across baselines along with performance metrics. Unknown rate, answered rate, and answered-only accuracy separate output compliance from classification quality.

Model	Acc.	F1	Inf. (ms)	Unk. Rate	Ans. Rate	Acc. (Ans.)
CodeBERT-PECAN	0.9831	0.9711	4.89	0.0000	1.0000	0.9831
Guesslang (Common)	0.6746	0.5696	2.54	0.0005	0.9995	0.6749
GPT-OSS 120B	0.4727	0.3733	2667.48	0.0004	0.9996	0.4729
Linguist	0.2001	0.1492	0.85	0.0227	0.9773	0.2048
Guesslang (Full)	0.1025	0.0346	2.54	0.0002	0.9998	0.1025

terpreted as naming-alignment effects rather than a reflection of changes in classification ability.

Reliability and coverage metrics are reported to separate prediction quality from model output validity. Specifically, unknown_rate captures abstentions from the model, answered_rate captures valid output coverage, and accuracy_answered_only captures the correctness of valid outputs produced by a model. This separation is important for constrained-output LLM baselines where formatting and output compliance can affect usable performance.

In this benchmark, GPT-OSS highlights how deployment controls can shape reliability metrics. GPT-OSS maintains a near-complete answered rate due to the enforced unknown retry policy with structured decoding, but its overall effectiveness remains substantially lower than CodeBERT-PECAN as seen through Table 1. As a result, its coverage reflects the behavior of the inference pipeline rather than intrinsic model performance, and should be interpreted carefully when comparing against non-retried baselines.

Runtime analysis is presented using average per-sample inference time. Figure 3 shows the accuracy-latency relationship among all baselines. It demonstrates that heuristic (Linguist) and application-specific deep learning (Guesslang) tools are often low-latency but degrade in accuracy in a strict metadata-free fragmented setting. GPT-OSS is substantially higher latency while still having significantly worse accuracy than CodeBERT-PECAN which provides strong

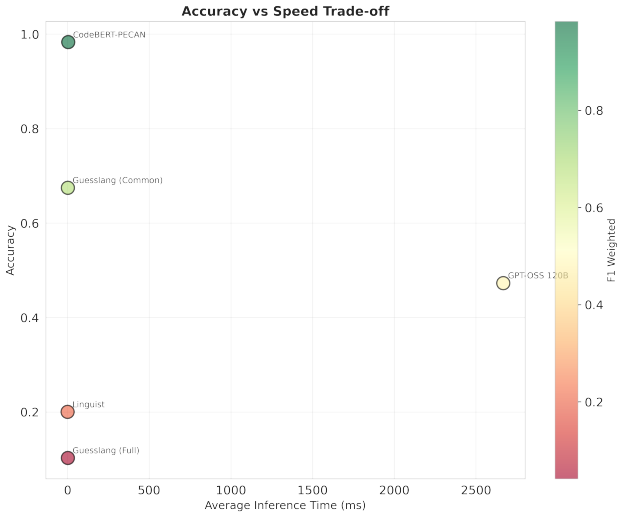

Figure 3. Accuracy–latency frontier. CodeBERT-PECAN occupies a favorable high-accuracy, low-latency region relative to other evaluated baselines.

effectiveness at a practical local inference cost.

From a deployment perspective, this suggests model choice should depend on deployment constraints rather than a single metric. If low-latency operation is necessary, application-specific deep learning tools such as Guesslang can be useful as a fallback. If strongest quality under snippet-only conditions is required while preserving local feasibility, CodeBERT-PECAN offers the most favorable tradeoff in this thesis scope.

4.3. Error Analysis and Limitations

Error analysis is treated as a post-hoc diagnostic stage built from per-sample prediction records. Quantitative outputs include top confusion pairs, top confusion pair counts, and per-language error

rates. Qualitative outputs include representative cases grouped by failure setting (partial_snippet, non_repo_context, ambiguous_or_overlap). This provides both aggregate trends and concrete examples of failure cases.

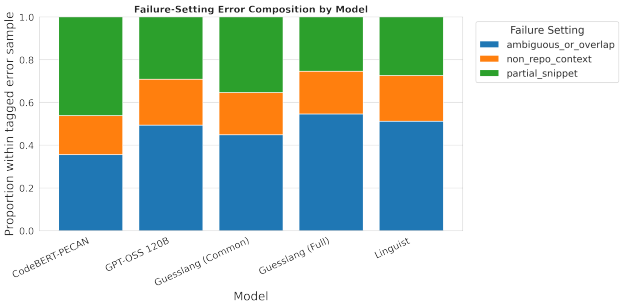

Figure 4. Failure-setting error composition by model. Error proportions differ across partial snippets, non-repository context, and ambiguous/overlap settings.

The cross-model confusion view in Figure 4 shows that weaker baselines tend to exhibit broader and higher-volume recurrent confusions, while CodeBERT-PECAN errors are narrower and more concentrated in overlap-like boundary cases. The failure-setting composition figure further indicates that error distribution differs by model family, which helps explain differences in aggregate metrics and where improvements are needed.

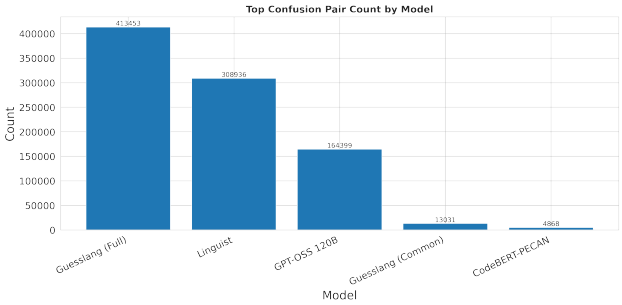

Figure 5. Top confusion pair count by model (count of each model's most frequent true→predicted error). Lower counts indicate less concentrated recurrent failure channels.

Top confusion pair count by model is especially useful here as it quantifies the strongest recurring failure channel for each model and provides a direct measure of brittleness concentration. High counts indicate repeated collapse on one language boundary, while lower counts suggest errors are more diverse or less structurally repetitive.

Several limitations should be considered. First, this thesis centers on one fully trained encoder family (CodeBERT) as the primary learned baseline; broader architecture-level claims are therefore intentionally limited. Second, runtime values are environment-sensitive and may vary with hardware, system load, and serving configuration.

Third, canonical scoring depends on alias-map definitions, which can influence edge-case values despite dual raw/canonical reporting. Finally, error-setting categorization is diagnostic by design and should be interpreted as structured empirical analysis rather than causal proof. These limitations do not undermine the findings, but they constrain external generalization and motivate follow-on work.

5. Conclusion

5.1. Key Findings

This thesis presents three main findings. First, CodeBERT-PECAN is the most effective with the best accuracy-latency tradeoff among evaluated baselines in this metadata-free, snippet-level PL-ID setting. Second, dual raw/canonical reporting shows that major conclusions are robust to label-normalization effects. Third, reliability and runtime reporting materially improve interpretation over accuracy-only comparisons, especially for constrained-output LLM baselines.

Together, these results suggest that PL-ID should be evaluated as a multi-objective problem balancing accuracy, reliability, and efficiency. Under that framing, CodeBERT-PECAN provides the strongest overall performance across evaluated metrics for the constraints targeted in this defense.

5.2. Practical Implications

For practical deployment, CodeBERT-PECAN is ideal when high snippet-level quality is required under metadata-poor conditions and local

inference is preferred. Heuristic and application-specific deep learning baselines remain useful in lightweight or fallback roles, but their degradation under strict fragment-only settings should be expected. GPT-OSS can be valuable where broader model flexibility is needed, but latency and output-control overhead must be explicitly budgeted.

These results are directly relevant to workflows in software indexing, static analysis pre-processing, digital forensics triage, and malware-analysis pipelines, where fragment-level inputs and missing context are common. The methodology also provides a reproducible template for evaluating future PL-ID systems under realistic constraints.

5.3. Future Work

Immediate next steps include extending full training/evaluation coverage to additional encoder families under the same protocol, enabling stronger architecture-level conclusions in the final paper. Additional work on tail-language calibration, confidence-aware abstention policy learning, and top-k diagnostic integration will improve both reliability and practical utility.

A second priority is closing the loop between diagnostics and mitigation: using confusion and failure-setting analyses to drive targeted model and post-processing improvements. This includes reducing dominant recurring confusion channels and stress-testing robustness under harder non-repository and partial-fragment distributions.

Bibliography

Agarwal, S., Ahmad, L., Ai, J., Altman, S., Applebaum, A., Arbus, E., Arora, R. K., Bai, Y., Baker, B., Bao, H., et al. gpt-oss-120b & gpt-oss-20b model card. *arXiv preprint arXiv:2508.10925*, 2025.

Allamanis, M., Barr, E. T., Devanbu, P., and Sutton, C. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 51(4):1–37, 2018.

Baltes, S., Dumani, L., Treude, C., and Diehl, S. Sotorrent: reconstructing and analyzing the evolution of stack overflow posts. In *Proceedings of the 15th international conference on mining software repositories*, pp. 319–330, 2018.

Bavota, G. Mining unstructured data in software repositories: Current and future trends. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 5, pp. 1–12. IEEE, 2016.

Dietrich, J., Luczak-Roesch, M., and Dalefield, E. Man vs machine–a study into language identification of stack overflow code snippets. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pp. 205–209. IEEE, 2019.

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., et al. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the association for computational linguistics: EMNLP 2020*, pp. 1536–1547, 2020.

Fratantonio, Y., Invernizzi, L., Farah, L., Thomas, K., Zhang, M., Albertini, A., Galilee, F., Metitieri, G., Cretin, J., Petit-Bianco, A., et al. Magika: Ai-powered content-type detection. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pp. 2638–2649. IEEE, 2025.

GitHub, Inc. Linguist. https://github.com/github-linguist/linguist, 2024. Version 9.5.0, Accessed: 2025.

Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., Liu, S., Zhou, L., Duan, N., Svyatkovskiy, A., Fu, S., et al. GraphCodeBERT: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*, 2020.

Guo, D., Lu, S., Duan, N., Wang, Y., Zhou, M., and Yin, J. UniXcoder: Unified cross-modal pre-training for code representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 7212–7225, 2022.

Kocetkov, D., Li, R., Allal, L. B., Li, J., Mou, C., Ferrandis, C. M., Jernite, Y., Mitchell, M., Hughes, S., Wolf, T., et al. The stack: 3 tb of permissively licensed source code. *arXiv preprint arXiv:2211.15533*, 2022.

Mittal, G., Korus, P., and Memon, N. Fifty: large-scale file fragment type identification using convolutional neural networks. *IEEE Transactions on Information Forensics and Security*, 16:28–41, 2020.

Niu, C., Li, C., Ng, V., Chen, D., Ge, J., and Luo, B. An empirical comparison of pre-trained models of source code. In *2023 IEEE/ACM 45th*

International Conference on Software Engineering (ICSE), pp. 2136–2148. IEEE, 2023.

Pimentel, J. F., Murta, L., Braganholo, V., and Freire, J. A large-scale study about quality and reproducibility of jupyter notebooks. In *2019 IEEE/ACM 16th international conference on mining software repositories (MSR)*, pp. 507–517. IEEE, 2019.

Rodriguez-Prieto, O., Pato, A., and Ortin, F. Plangrec: Deep-learning model to predict the programming language from a single line of code. *Future Generation Computer Systems*, 166:107640, 2025.

Roussev, V., Quates, C., and Martell, R. Real-time digital forensics and triage. *Digital Investigation*, 10(2):158–167, 2013.

Somda, Y. Guesslang: Detect the programming language of a source code. https://github.com/yoeo/guesslang, 2021. Version 2.2.1.

Tam, Z. R., Wu, C.-K., Tsai, Y.-L., Lin, C.-Y., Lee, H.-y., and Chen, Y.-N. Let me speak freely? a study on the impact of format restrictions on performance of large language models. *arXiv preprint arXiv:2408.02442*, 2024.

Willard, B. T. and Louf, R. Efficient guided generation for large language models. *arXiv preprint arXiv:2307.09702*, 2023.

Yang, G., Zhou, Y., Yu, C., and Chen, X. Deepscc: Source code classification based on fine-tuned roberta. *arXiv preprint arXiv:2110.00914*, 2021.