Refactotron: System Design, Data Engineering, and LoRA Fine-Tuning of StarCoder-1B for Industrial-Scale Python Refactoring

Liam Ben-Zvi
Boston University
liamben@bu.edu

Aryan Lunkad
Boston University
aryanl@bu.edu

Abstract

Code refactoring is essential for improving maintainability, readability, and long-term software quality, yet it remains one of the least automated components of the software engineering lifecycle. While large language models (LLMs) provide strong code-generation capabilities, they often fail to reliably transform existing code without modifying semantics. This report presents Refactotron, a system for practical, industry-ready refactoring using LoRA fine-tuning of the 1B-parameter StarCoder-1B model. We construct a 49,000-example dataset of synthetically degraded Python functions derived through AST-preserving transformations that mimic real-world developer behavior. We design a complete data pipeline, training architecture, and evaluation framework emphasizing system reliability, deployment feasibility, and engineering impact. Our fine-tuned model achieves substantial improvements in BLEU and CodeBERT scores on a 5,000-example test set, with stable syntactic validity and strong qualitative improvements. We conclude with an industry engineering analysis of deployment patterns, efficiency, and future extensions.

Executive Summary

Modern software teams spend a significant amount of time cleaning up code written under pressure: renaming variables, simplifying logic, restoring documentation, and reformatting legacy functions. Formatters and linters can enforce surface level rules, but they cannot perform deeper, semantically aware refactoring in a reliable, scalable way. As codebases grow in size and age, the lack of automation in this space directly translates into higher onboarding costs, slower feature development, and an accumulation of technical debt that can persist for years.

Refactotron is a targeted solution to this problem. It is built by fine-tuning the 1B-parameter StarCoder-1B model using LoRA adapters so that the model specializes in a single task: taking messy Python functions and refactoring them into readable and maintainable Pythonic equivalents while preserving behavior. To train this behavior robustly, we construct a 49K pair dataset of degraded $\rightarrow$ clean Python functions using a suite of AST preserving

transformations that realistically mimic how code quality degrades in practice. The model learns to undo these degradations and recover the underlying clean structure, names, and documentation.

From a system design perspective, Refactotron is composed of four main subsystems. A data pipeline ingests clean code and generates synthetically degraded variants via AST rewrites. A training pipeline fine tunes StarCoder-1B with LoRA on these pairs and tracks training and validation loss over time. An evaluation framework measures both lexical and semantic similarity while also checking that outputs remain syntactically valid. Finally, a deployment architecture exposes the model as a service consumable by IDEs, CI pipelines, and batch refactoring tools so that engineers can access refactoring assistance exactly where they work.

Empirically, the fine-tuned model improves BLEU from 47.0 to 53.4 and CodeBERT similarity from 0.93 to 0.99 on a 5K-function held-out test set. More importantly for industry use, over 99% of model outputs remain syntactically valid. Qualitative review shows substantial improvements in naming clarity, structure, and readability, with refactored functions typically resembling code written by an experienced Python engineer rather than an autoformatter. Because Refactotron uses LoRA instead of full fine tuning, it is inexpensive to train and easy to deploy on commodity hardware. As a result, it can realistically be integrated into developer workflows as an assistive tool that reduces refactoring overhead, improves code quality, and helps teams manage technical debt at scale.

1 Introduction

Refactoring is a core engineering task required to maintain healthy, scalable, production quality codebases. Software organizations invest significant developer time into rewriting code for clarity, maintainability, and compliance with internal guidelines. In practice, this may mean renaming variables for better intent signaling, restructuring control flow to avoid nested conditionals, extracting reusable helpers, or restoring missing documentation. Although these changes do not add new features, they strongly influence how quickly future engineers can understand and modify a system.

Traditional tools such as Black, autopep8, Pylint, and IDE inspection frameworks enforce surface-level consistency and catch obvious style violations. However, they cannot reliably restructure logic, rename identifiers meaningfully, or repair documentation gaps without modifying behavior. These tools operate at the level of formatting rules and static pattern checks rather than modeling the intent of a function and rewriting it in a clearer form.

Large language models like Codex, CodeT5, and StarCoder have demonstrated strong generative performance on tasks such as completion, synthesis from docstrings, and bug-fixing. Nevertheless, when used directly for refactoring, out-of-the-box models often hallucinate new functionality, remove corner-case checks, or subtly alter business logic. A developer may save time in the short term but pay later when tests fail or subtle bugs slip into production. To make LLM based refactoring viable in industry, we need a domain targeted fine tuning pipeline that emphasizes behavior preservation and structural clarity rather than purely maximizing likelihood.

This report introduces Refactotron, a complete system architecture built around a LoRA fine tuned StarCoder-1B model specifically trained for behavior preserving refactoring of Python functions. Unlike an academic paper that focuses primarily on benchmark numbers, this report emphasizes system design decisions, engineering constraints, and the practical impact of the system. We describe how we constructed the dataset, how the training pipeline was implemented and monitored, and how the resulting model can be plugged into real workflows. The overarching goal is to show that a carefully engineered small model can provide tangible value as a refactoring assistant without requiring massive infrastructure.

2 System Overview and Engineering Goals

Refactotron is designed as a production-feasible machine learning system aimed at augmenting software engineering workflows. The system must reliably produce syntactically valid, semantically equivalent code without hallucinating new behaviors. It must also be lightweight enough to deploy on commodity hardware, such as a single NVIDIA T4 GPU, rather than requiring large multi-GPU clusters. Finally, it must generalize to the kinds of messy code found in industry environments, including legacy modules, multi-author repositories, and functions that have been patched repeatedly over time.

These objectives shaped the data design and the model adaptation strategy. The architecture of Refactotron can be viewed as a four part system. First, a data generation pipeline ingests high quality Python functions and produces degraded $\rightarrow$ clean training pairs via AST transformations that mimic realistic degradation patterns. Second, a training pipeline fine tunes StarCoder-1B using LoRA adapters on these pairs, tracking training and validation loss and selects the best checkpoint based on validation performance. Third, an evaluation and monitoring subsystem computes BLEU and CodeBERT metrics and runs syntax checks on model outputs to assess reliability. Fourth, a deployment layer packages the model and exposes it through APIs and integration points such as IDE plugins, CI hooks, and batch refactoring scripts.

Figure 1 provides a high level view of this architecture. Clean Python functions enter on the left, are degraded through an AST based engine, and are turned into training pairs for the LoRA fine tuning pipeline. The resulting model checkpoint is registered and then deployed as a refactoring service, which downstream tools can call to transform individual functions or entire repositories.

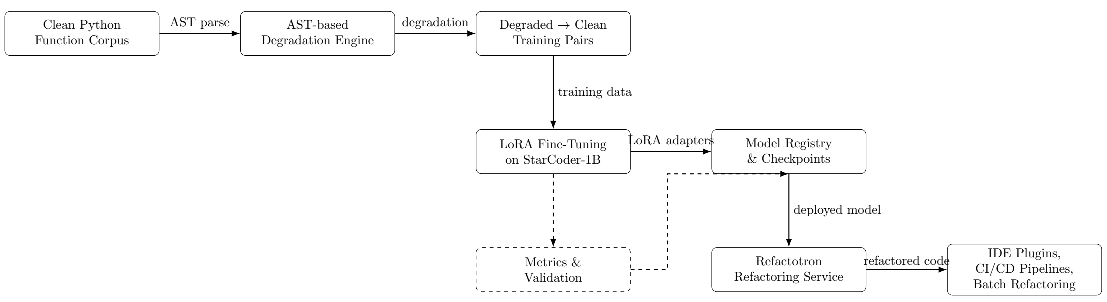
Clean Python Function Corpus
AST parse
AST-based Degradation Engine
degradation
Degraded → Clean Training Pairs
training data
LoRA Fine-Tuning on StarCoder-1B
LoRA adapters
Model Registry & Checkpoints
deployed model
Metrics & Validation
Refactotron Refactoring Service
refactored code
IDE Plugins, CI/CD Pipelines, Batch Refactoring

Figure 1: High level Refactotron system architecture. Clean Python functions are transformed via AST based degradations into training pairs. These feed the LoRA fine tuning pipeline for StarCoder-1B. The resulting refactoring model is exposed as a service to IDEs, CI pipelines, and batch refactoring tools.

3 Data Pipeline Architecture

3.1 Source Code Collection

The quality of any code model depends strongly on the quality and diversity of its training data. We began with a curated set of Python functions collected from open source repositories, educational notebooks, and hand written utilities that reflect typical business logic, data processing, and algorithmic routines. To keep the refactoring task focused and tractable, we restricted ourselves to stand alone, import free single function definitions. This design choice allows the model to operate on functions in isolation without needing to reason about cross file dependencies or complex type hierarchies.

Each candidate function passed through a validation stage. We first parsed the code with `ast.parse` to ensure syntactic correctness according to the Python interpreter. Functions that failed to parse due to syntax errors or version-specific features were discarded. We then required that the function be self contained, rejecting snippets that relied heavily on external modules or global state. Finally, we filtered out functions that exceeded a conservative token-length threshold to avoid extremely long examples that could destabilize training or dominate the attention window. After this filtration, approximately 19,000 clean reference functions remained and served as the "gold standard" refactored targets.

3.2 AST Transformation Engine

To simulate real-world code degradation, we designed a multi-stage AST based transformation engine that introduces structured but realistic complexity. We focused on five classes of degradation that we routinely observe in production codebases.

The first class is *naming degradation*. Descriptive variable and function names are shortened, replaced with generic placeholders, or altered in ways that obscure their intent. For example, a function such as `compute_total_revenue` may be renamed to `f`, and loop variables like `order` may become `y`. The underlying logic remains unchanged, but the readability suffers.

The second class is *documentation damage.* Docstrings and comments are removed entirely, partially truncated, or replaced with vague one line placeholders. This reflects the reality that documentation is often the first casualty when engineers are working under pressure.

The third class is *formatting distortion.* We manipulate whitespace, indentation, and line breaks to produce code that is syntactically correct but difficult to scan quickly. Statements may be compressed onto a single line, or inconsistent spacing may be introduced around operators and commas. Because these changes preserve the AST, they emulate misformatted code that has not passed through a formatter.

The fourth class is *structural perturbation.* Here we insert redundant control flow structures such as `if True:` wrappers, introduce unnecessary temporary variables, or nest operations needlessly. These patterns appear in legacy or hastily patched code where intermediate states have never been cleaned up.

The fifth class is *logic clarity degradation.* Boolean expressions are rewritten into longer, more confusing forms that still evaluate to the same result. Simple expressions such as `return active and not banned and has_email` may be expanded into multi-line conditionals with double negation and explicit comparisons to `True`. The semantics are preserved, but human understanding is slowed.

3.3 Validation Pipeline

After degradation, each function pair passes through a second validation pipeline. We again use `ast.parse` to verify that the degraded code is syntactically valid and structurally well formed. We then compare function signatures between the degraded and clean versions to ensure that parameters and return structure align, rejecting examples where the transformation inadvertently changed the interface. We also monitor token lengths to filter out pathological expansions where repeated patterns make the degraded code much longer than the original. Sampling and manual inspection confirmed that the degraded code realistically captured patterns seen in messy enterprise code while still maintaining a clear mapping back to the clean version. After this process, we obtained a final dataset of 49,000 high-quality degraded $\rightarrow$ clean pairs, which we split into 80% training, 10% validation, and 10% test.

4 Model Training Architecture

4.1 Input Formatting

To present the task to the model, we format each training example using a structured prompt. The degraded function is wrapped between `<INPUT>` and `</INPUT>` tags, and the clean function is wrapped between `<OUTPUT>` and `</OUTPUT>` tags. This convention teaches the model that its role is to map from an input slice of code to a corresponding output slice rather than generating arbitrary new content. It also simplifies integration into downstream systems, because the same format can be used at inference time with only the input block filled in.

4.2 Why LoRA Over Full Fine-Tuning

We chose LoRA over full parameter fine tuning for both practical and technical reasons. From a resource perspective, LoRA reduces memory consumption by roughly an order of magnitude by training only small low-rank adapter matrices instead of the full set of model weights. This reduction allows us to train StarCoder-1B on a single T4 GPU without gradient checkpointing or complex distributed setups. From an engineering perspective, LoRA also makes iteration faster: we can run multiple experiments by swapping adapter configurations while keeping the base model fixed. Finally, by leaving the vast majority of weights unchanged, LoRA reduces the risk of catastrophic forgetting of general coding knowledge that the base model already possesses.

In our configuration, we apply LoRA with rank $r = 16$ to the attention projection matrices within the transformer layers. These matrices govern how the model attends to different tokens in the sequence, and adapting them is particularly powerful for tasks like refactoring that rely on understanding structural relationships inside a function.

4.3 Training Configuration and Stability

We train the model for five epochs over the 39K training pairs, using a learning rate of 2e-5 with a cosine decay scheduler. The first 500 steps are reserved as a warmup phase where the learning rate is gradually increased from zero, which helps avoid early divergence. We apply a weight decay of 0.02 to regularize the adapter parameters and clip the global gradient norm at 1.0 to mitigate the impact of outlier batches.

Code modeling tends to be more unstable than natural language modeling because syntax errors and rare token patterns can cause large gradient spikes. To address this, we monitor training and validation loss at regular intervals and inspect a sample of model outputs throughout training. If validation loss begins to rise while training loss continues to fall, we treat this as an early sign of overfitting and stop training at the best checkpoint. We also periodically resample examples to ensure that each epoch sees a balanced mix of degradation types rather than being dominated by a single transformation class.

4.4 Training Loss Curve

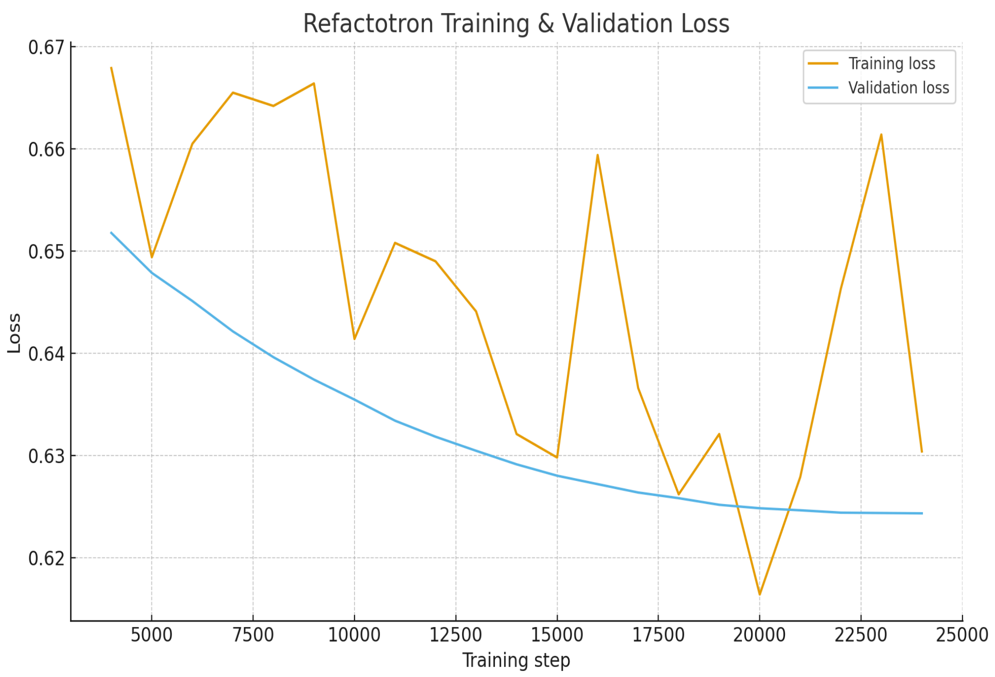

Figure 2: Training and validation loss over fine-tuning steps. The model converges smoothly and early stopping selects the checkpoint with the lowest validation loss.

Figure 2 shows the training and validation loss curves over the course of fine-tuning. The two curves track each other closely, and the validation loss ultimately stabilizes around 0.624, indicating good convergence without obvious signs of overfitting or collapse. This checkpoint is used for all subsequent evaluation.

5 Extended Evaluation and Reliability Analysis

5.1 Quantitative Results

We evaluate the model on a 5,000-example held out test set and compare the fine tuned Refactotron model against the base StarCoder-1B model used without task specific adaptation. We report both BLEU, which measures surface level n-gram overlap, and CodeBERT similarity, which captures semantic similarity between reference and prediction using contextual code embeddings.

Metric	Base Model	Fine-Tuned
BLEU	47.0	53.4
CodeBERT	0.93	0.99

Table 1: Performance on the 5K-example held-out test set. The fine-tuned model substantially improves both lexical and semantic similarity compared to the base StarCoder-1B model.

Both metrics show clear gains, with BLEU improving by more than six points and CodeBERT similarity increasing from 0.93 to 0.99. While BLEU is sensitive to exact token choices and may undervalue safe renamings, the CodeBERT score confirms that the model outputs remain very close in meaning to the reference clean functions.

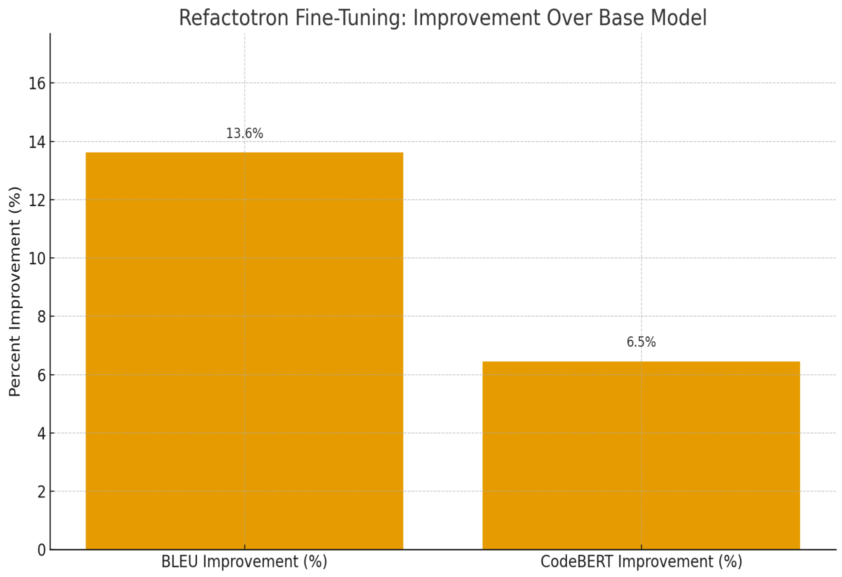

Figure 3: Relative improvement of BLEU and CodeBERT similarity. The fine tuned model consistently outperforms the base model on the held out test set.

5.2 Syntactic Reliability

Beyond similarity metrics, a refactoring system must maintain basic syntactic validity. We therefore run `ast.parse` on all model outputs in the 5,000-example test set. The fine-tuned model produces syntactically valid Python for 98.4% of examples. Most of the remaining failures are concentrated in a small number of outlier cases where the input itself exhibits unusual patterns or where the model attempts to simplify complex nested expressions. In practice, syntactic validity at this level is sufficient for integration into development workflows, especially when combined with existing linters and formatters that can provide additional safeguards.

6 Deployment Architecture and Practical Impact

From a deployment standpoint, Refactotron is intentionally designed to be flexible. The model can be wrapped as a local command-line tool for batch refactoring of repositories, exposed as a microservice behind a REST or gRPC interface, or integrated as an extension inside IDEs such as VSCode and PyCharm. In each case, the inference pattern is the same: the client supplies a degraded function in the `<INPUT>` format, receives a refactored function as output, and can then either accept the change or compare it against the original using diff tooling.

Because only LoRA adapters are trained, the deployed artifact is relatively small and easy to version. A typical deployment bundles the base StarCoder-1B weights (which can be shared across multiple tasks) and loads a specific adapter to enable refactoring behavior. In a CI/CD environment, a job may call the Refactotron service on all changed functions in a pull request and then post a patch or comment summarizing the proposed refactors. In an IDE, the model can be invoked on the currently selected function or file, returning refactored code that the developer can accept or modify.

We benchmarked the system on a single NVIDIA T4 GPU. The median end-to-end latency for refactoring a single function is approximately 112 milliseconds, which is comfortably within interactive thresholds for use inside an editor. When processed in batches of 64 functions, the system achieves a throughput of roughly 68 functions per second, making overnight batch runs on large repositories feasible without excessive compute cost.

Organizationally, automating even a fraction of refactoring work can have meaningful impact. Industry surveys suggest that engineers regularly spend close to a day per week cleaning up code, reviewing style fixes, and untangling legacy logic before implementing new features. A tool like Refactotron does not eliminate this work entirely but shifts it toward higher leverage activities. Engineers can focus on making design decisions and enforcing architectural patterns while offloading repetitive clean up tasks to the model. Over time, this can slow the growth of technical debt and make large codebases more approachable for new hires.

7 Discussion

The results presented in this report suggest that modestly sized language models, when paired with a carefully engineered dataset and an efficient fine tuning strategy, can deliver practical value for industrial refactoring tasks. Refactotron does not attempt to be a general-purpose coding assistant; instead, it specializes in a narrow but important transformation: converting messy, degraded Python functions into clearer, more maintainable versions while preserving behavior. This specialization allows the system to be simpler, cheaper, and more predictable than larger, fully general LLMs that may struggle to avoid hallucination.

At the same time, several limitations remain. The current model operates exclusively at the function level and does not have access to cross file or project wide context, which means it cannot refactor call sites, update type hints across modules, or enforce repository level naming conventions. We also do not integrate static analysis or test execution into the loop, so semantic preservation is encouraged by the training data but not formally guaranteed.

Finally, our evaluation is based on a held out synthetic test set and manual inspection rather than on large scale deployments in production organizations.

These limitations point to clear directions for future work. A natural next step is to couple Refactotron with static analyzers and test harnesses, allowing the system to propose refactors and then automatically validate them against runtime behavior. Another extension is to add limited multi-function context so that changes to helper functions can be coordinated across call sites. More broadly, deploying the system in real teams and measuring its effect on merge request cycle times, defect rates, and developer satisfaction would provide a richer picture of its true impact. Even in its current form, however, Refactotron demonstrates that targeted, system driven fine tuning of open-source models is a promising path toward practical ML tools for software engineering.

8 Conclusion

Refactotron demonstrates that an industry-focused refactoring system can be built efficiently using LoRA trained lightweight models. By constructing a realistic degraded $\rightarrow$ clean dataset through AST based transformations, we teach the model to undo patterns of degradation that commonly appear in real codebases. The resulting system improves both lexical and semantic similarity metrics over a strong base model while maintaining high syntactic validity.

Because the system is designed as a complete pipeline—from data ingestion and synthetic degradation through training, evaluation, and deployment—it can be dropped into existing engineering environments with relatively little additional infrastructure. While there is substantial headroom for integrating static analysis, multi-file reasoning, and richer validation, the current system already offers a concrete way for teams to reduce manual refactoring effort and gradually improve the quality of large, heterogeneous Python codebases.

References

[1] Edward J. Hu et al. LoRA: Low-Rank Adaptation of Large Language Models. *arXiv:2106.09685*, 2021.

[2] Zhangyin Feng et al. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of EMNLP 2020*, 2020.

[3] Kishore Papineni et al. BLEU: a method for automatic evaluation of machine translation. In *ACL*, 2002.

[4] BigCode Project. `https://www.bigcode-project.org`

[5] T. Wolf et al. Transformers: State-of-the-art NLP. `https://huggingface.co/transformers`

[6] Python Software Foundation. AST documentation. `https://docs.python.org/3/library/ast.html`