Exploring Opensource AI for LLM Code Translation Systems

Omotola Awofolu

December 2025

Abstract

AI code generation has unlocked a new wave of possibilities in recent years. Coding agents have evolved from simple autocompleters to full-fledged, collaborative coding agents, capable of reasoning, debugging, refactoring and other advanced development tasks. These breakthroughs have been inspired by innovations with transformer architectures, multi-agent systems and specialized code Large Language Models (LLM). However, significant challenges remain in several areas, including code understanding and code translation for legacy and niche languages. In particular, AI code translation for legacy languages is relatively less mature than other tasks, as it is more likely to be impacted by poor documentation, abstract code dependencies, tangled "spaghetti" code and fewer training datasets and benchmarks due to the scarcity of publicly available repositories. This creates a significant burden for enterprises, as it results in increased security risks, higher operational costs, reduced innovation and challenged talent retention. This project explores a practical approach for improving the efficiency of legacy code refactoring with open source LLMs. For demonstration purposes, the prototype involves the creation of an LLM-backed coding agent that automates the migration of Java Server Pages (JSPs) to a more modern NodeJS and React architecture. It leverages various methodologies such as continuous finetuning, synthetic data generation and spec-driven development, and presents an optimal, cost-effective approach for building the agentic system. Using predefined metrics as guidance, it builds a finetuned model that provides a measurable improvement in legacy code refactoring tasks over the baseline model. In particular, using DORA finetuning, the prototype yielded an 18%

median improvement in code relevant metrics over the baseline model, and outperformed LoRA finetuning by a factor of 2-2.5X in overall median performance.

Introduction

Overview of task

AI code generation is the use of artificial intelligence to perform coding tasks that have been traditionally owned by the software developer. Code generation covers a broad spectrum of tasks with varying degrees of autonomy. At the low end of the spectrum, code generation focuses on copilot assistant activities, such as autocomplete assistance and code styling. On the other end, code generation involves both partially and fully autonomous agents integrated with software development workflows, performing tasks like code translation, code refactoring, troubleshooting and code writing. AI code generation is one of the most rapidly evolving areas of advancement in Generative AI till date [1].

Prior to the current era, code generation had been an active area of interest for AI researchers and computational linguists for decades. The earliest attempts involved methodologies that were similar to natural language generation approaches, using static rule-based systems, template pattern matching, and statistical methods such as Hidden Markov Models and Conditional Random Parsing [2]. Later approaches began to explore neural networks for improved semantic understanding and larger context handling without the need for extensive manual feature engineering [3]. However, the state of the art involves transformer model architectures [4]. The self-attention mechanism of transformers allows them to capture long-range dependencies with larger data sizes, while the associated capacity for parallel processing significantly streamlines the efficiency of such processing. Transformers provide a practical approach for learning from millions of code snippets and hundreds of gigabytes of code-to-text pairs without overtraining. Modern AI code generation also benefits from the advent of reasoning models, more sophisticated frontier LLMs, and training datasets that have been growing by orders of magnitude over time.

Illustrative Example

The following is an example of a prompt for a Python code generation problem that can be handled by state-of-the-art large language models:

```
# Example of average benchmark performance
Write a generic, fullstack Guestbook web application using Python
3.10 as the backend language. The application must follow the
requirements and validation rules specified here:
<spec>
Include unit tests (ensure at least 70% code coverage), a README file
with deployment instructions, an API specifications document, and all
library dependencies.
# Example of below-average benchmark performance
After building an initial prototype, refactor the code by identifying
potential performance bottlenecks and rewriting a more efficient
version of the code.
```

The prompt above displays the following capabilities of modern AI code generation:

1. ### Complete Feature Generation
   As shown, modern code generation is able to translate high-level requirements into mostly production ready code.

2. ### Fullstack Code Generation
   Modern coding agents can implement features from end to end. This includes advanced coding tasks such as end-to-end feature development, API design, and automated error handling.

3. Intelligent Test Generation
   Modern coding agents have the ability to autogenerate complete test suites based on the specified requirements.

As shown above, modern code generation also has its gaps. For example, while coding agents have the ability to refactor codebases, their understanding of certain kinds of context surrounding the codebases is limited in areas like performance improvement. As discussed earlier, coding agents are also less performant when it comes to generating code in legacy and/or niche languages.

Problem Definition

This project focuses on one of the current gaps in modern code generation - legacy code refactoring - and uses a real-world prototype to demonstrate an approach for improving its efficiency with open source LLMs. The prototype showcases the implementation of an coding agent that will automate the translation of Java Server Pages (JSPs) to a more modern NodeJs and React architecture using an LLM that has been finetuned from a published baseline. The objective of the project is to use predefined metrics to showcase a measurable percentage improvement in legacy code refactoring tasks over the baseline model. This will accomplish the following:

1. Continuous Learning
   By presenting a measurable performance improvement, the prototype demonstrates the ability to improve the performance of legacy code refactoring for specific use cases using continuous learning.

2. Opensource-First AI
   By using open source LLMs, the prototype demonstrates the ability to achieve performance improvement in a self-managed, controlled and cost effective manner. It removes the need to be a part of a frontier lab, and mitigates the legal, budgetary and/or flexibility bottlenecks that can be associated with using proprietary models.

The following is an outline of the motivation behind this project, as well as its potential significance:

1. Acceleration of time-to-value for enterprises
   Many enterprises are dealing with large amounts of legacy code that are costly and time-consuming to maintain, modify or operationalize in an agile manner. They would like to refactor these codebases, but struggle with knowing where to begin. Introducing AI-enhanced automation helps to short-circuit the refactoring process for these enterprises.

2. Reduction of technical debt
   Legacy codebases are often fragile, difficult to understand, and could even pose security risks. Using AI code translation, it becomes easier to tackle this kind of technical debt.

3. Future research opportunities
   Although the project's 13-week duration is a relatively short timeline, findings from this project could be leveraged for more in-depth research in the future.

Literature Review

Many research papers have been reviewed, including more than the limited selection identified here. However, this section provides some of the most important papers that were relevant to the project implementation:

1. Source Code Summarization in the Era of Large Language Models [5]
   This paper introduces the "Code-to-Summary-to-Code" approach that was used to finetune the candidate models in this project. Specifically, it captures the bottom-up, hierarchical approach that was employed for the prototype. Unlike a top-down approach, which assumes prior understanding of the entire codebase and can miss low-level details, a bottom-up approach to code summarization is ideal for capturing as many low-level details of the code as possible, as well as composability. Additionally, a hierarchical representation of the codebase was constructed by constructing a dependency graph of the code. A hierarchical representation was chosen over a basic sequential representation in order to capture the uniquely graphical nature of code in general.

2. Textbooks are All You Need [6]
   This explores the methodology that was used to train phi-1, the precursor to the Phi family of small language models. It provides background information that will be used to identify smaller, low-compute model options during the model selection process.

3. DORA: Weight-Decomposed Low-Rank Adaptation [7]
   This paper introduces a category of Parameter Efficient Finetuning (PEFT) which draws from prior work in weight reparameterization. DORA is one of the finetuning methodologies that will be used to build candidate models for this project. It has gained recent popularity over other similar methods due to its ability to provide better accuracy results even with similar training and inference costs.

4. Role-Player for Code Summarization Evaluation (CODERPE) [8]
   This paper illustrates an LLM-centric approach to evaluating code summaries. It attempts to provide a more salient code evaluation approach. It has been shown to be more scalable

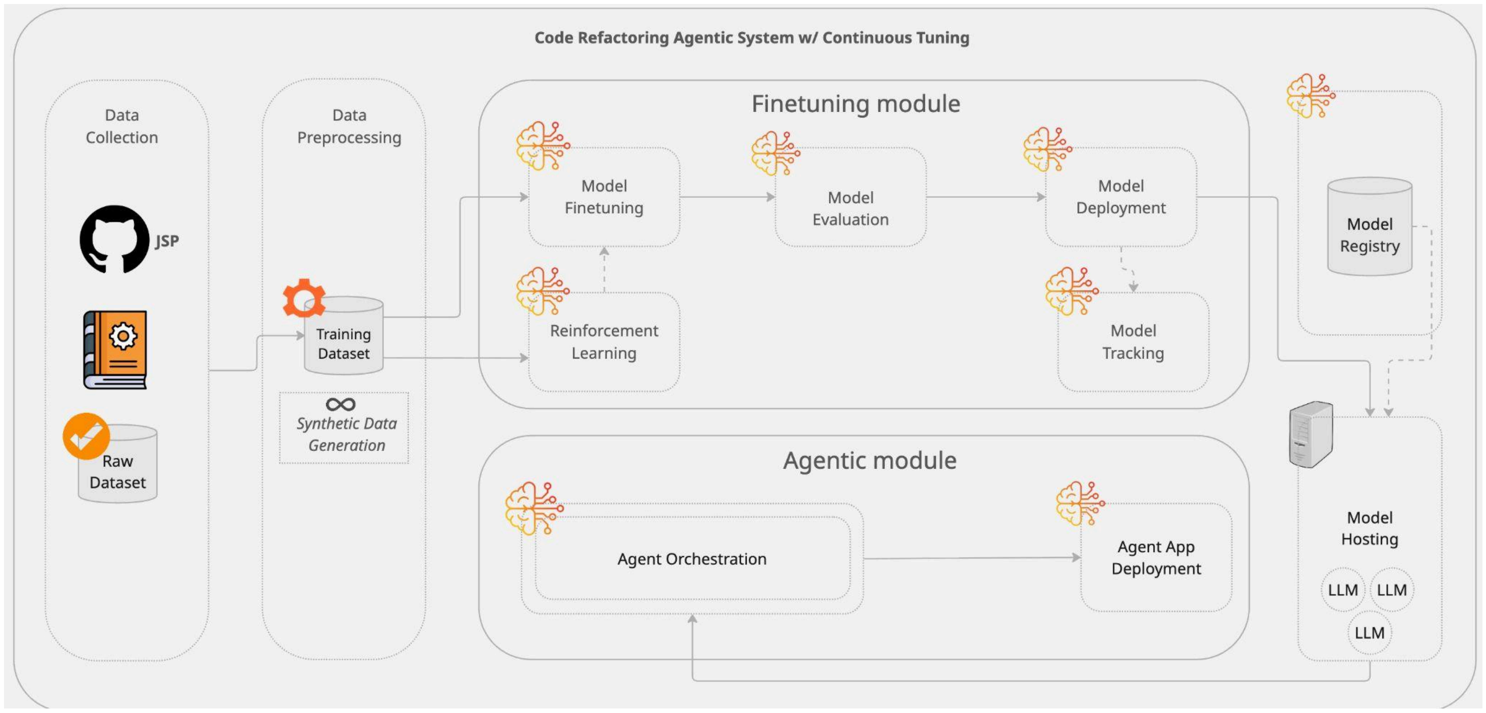
and less laborious than human evaluation, while simultaneously providing better correlation to human judgement than traditional NLP metrics such as ROUGE-L, BLEU-4 and BERTScore.

Experimental Design

High-Level Overview

The following diagram presents a high-level overview of the prototype:

Diagram 1: High Level Technical Overview of Code Translation System



The prototype will consist of two modules:

1. Finetuning module

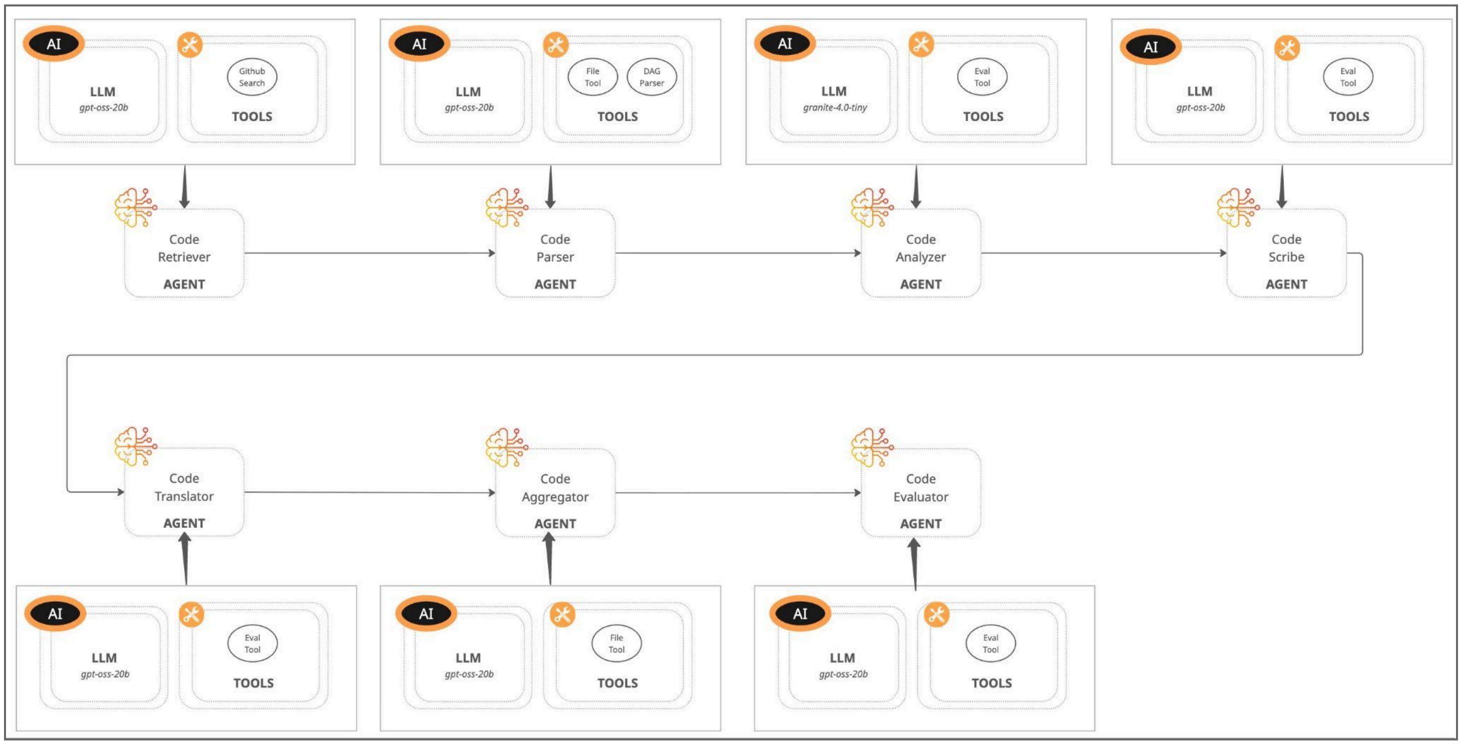
   The **finetuning module** is responsible for the continuous tuning of the JSP code summarization model, which is the legacy code targeted for this prototype. The JSP code summarization model is one of the agent-backing LLMs that was used by the multi-agent system in the agentic module. This module leveraged a selection of parameter-efficient finetuning (PEFT) pipelines to finetune the candidate models for the agentic module, including DORA (Weight-Decomposed Low-Rank Adaptation) and LoRA (Low-Rank Adaptation).

2. Agentic Module

The **agentic module** is the primary module responsible for the code translation workflow. It was implemented using the CrewAI framework.The agentic module consists of the following agents and tools:

- **Code Retriever**: Retrieves the legacy codebase (JSP) from a specified list of github repositories.
- **Code Parser**: Using a simple lexical parser, transforms the codebase into a hierarchical graph structure based on their dependencies, then clusters and ranks the processing order of the code based on these dependencies.
- **Code Analyzer**: Generates a mapping of code-to-text pairs for the low-level components of the code.
- **Code Scribe**: Generates software specification documents for the code using the output from the Code Analyzer.
- **Code Translator**: Using spec-driven development, generates a translated version of the code into the target language/framework (NodeJS/React) based on the specs from the Code Scribe.
- **Code Aggregator**: Uses a bottom-up approach to aggregate the translated code and generated specs into one codebase.
- **Code Evaluator**: Evaluates the generated code and specs using both custom and classic metrics.

Diagram 2: High Level Technical Overview of Agentic Module



Data

There were 3 categories of datasets that were used: Raw Datasets, Data Augmented Datasets and Synthetically Generated Datasets.

Raw Datasets

These are the original, real-world datasets that were manually curated from public data sources. They were used as the source of truth for synthetic data generation which was utilized for this task [9]. To curate this, a total of **5** public GitHub repositories were selected; Github's code search feature was used to find viable, publicly available JSP repositories that were at least 10 years old. Additionally, the textbook "Core Servlets and JSP, Volume 1"[12] by Marty Hall was obtained from an online source and downloaded as a JSP file.

Data Augmented Datasets

These are real-world datasets that were completely curated from the raw datasets and transformed using feature engineering into the input formats required for model training. These datasets mostly support classic data pre-processing techniques, such as feature encoding, class imbalance handling and missing value handling. Specifically, repositories with missing relevant source files were deleted, and textbook pages without code sections were also deleted from the corpus. These pre-processing techniques were applied during the Exploratory Data Analysis phase [9].

Synthetically Generated Datasets

These are datasets that were either completely or partially sourced from artificial data produced by LLMs and other generative AI models. In addition to classic techniques, pre-processing for synthetic datasets requires the use of LLM evaluation metrics such as relevance, faithfulness and correctness. Specifically, coherence and relevance metrics from the **G-Eval** framework were used to evaluate the synthetic datasets used for this project.

Optimal filtering mechanisms for the synthetic dataset were discovered during the Exploratory Data Analysis process [9]. For example, a threshold score of 0.6 for relevance was used to filter out invalid synthetic data records, and manual spot checking was also applied for the dataset evaluation. The evaluation results themselves were captured as a HuggingFace dataset ("Dataset Evaluation Results" in the table 1 below).

Of these datasets, the **Code-to-Text** dataset could be seen as the primary dataset for this project, as it was the core training dataset used for the finetuning module. The other datasets were source-of-truth datasets that were ultimately used to

a) derive the Code-To-Text dataset; or

b) complement the Code-To-Text during the model training and evaluation process.

List of Datasets

The following table is a summarized outline of all of the datasets that were generated for the code translation task.

Table 1: summarized outline of datasets

Dataset	**Purpose**	**Type**	**Link**
Github Code Repositories	Source for input features in the evaluation dataset	Raw dataset (evaluation)	See Public GitHub Repository Links
"Core Servlets and	Unstructured source	Raw dataset	Github Link

JSP” Textbook (PDF)	for input features in the finetuning dataset	(training)	
“Core Servlets and JSP” Converted Textbook (Markdown)	Structured / enriched source for input features in the training dataset	Data augmented dataset	GitHub Link
Code-to-text pairs	Training dataset	Synthetically generated dataset from textbook, LLM generation	HuggingFace Repo
Dataset Evaluation Results	Dataset Evaluation Results	Synthetically generated dataset	GitHub Link
Model Evaluation Results	LLM Evaluation Results	Synthetically generated dataset	HuggingFace Repo
Prompt Templates	Used by LLMs for text generation	Data augmented dataset	Jupyter Notebooks
Reference Documents	Used by judge LLM for LLM-as-judge evaluations	Data augmented dataset	GitHub Link

Analysis of Finetuning Dataset: Characteristics, Size, Samples

The primary dataset used for the finetuning was the **Code-to-Text** training dataset. The code-to-text pairs were generated using synthetic data generation, based on a bottom-up approach to hierarchical code summarization as described in the paper “Hierarchical repository-level code summarization for business applications” [2]. The resulting dataset was published to HuggingFace (Detailed Explanation of Datasets).

The following code-to-text pairs were generated:

- **Code-to-markdown:** Code snippets were paired with their source markdown sections.
- **Code-to-summary:** Code snippets were paired with the autogenerated code summary.
- **Code-to-functional-requirements:** Code snippets were paired with autogenerated functional requirements.
- **Code-to-business-requirements:** Code snippets were paired with autogenerated business requirements.
- **Code-to-topics:** Code snippets were paired with autogenerated topics.
- **Code-to-components (JavaBeans, Controllers, Views, Custom Tags):** Code snippets were paired with JSP-relevant components in the code.
- **Code-to-domain-model:** Code snippets were paired with an autogenerated code domain model based on inspecting the code.

The code-to-text pairs were serialized as JSON features and published to HuggingFace (List of Datasets).

The original dataset had **346** total records in the initial code-completion dataset, representing about **28.9%** of the original corpus that was extracted for the code translation task. The data extraction process involved filtering the corpus down to chunks with code snippets. After data preprocessing was applied, the final dataset retained **260** records, which were serialized as a **json file** and published to HuggingFace. An **80-20** training-test split was applied to the data [9].

Evaluation Metrics

Evaluation metrics were computed for both the finetuning dataset and the candidate models. In both cases, Human-in-the-Loop (HITL) and LLM-as-judge metrics were used. This approach was used to compensate for the relative lack of benchmarks for JSPs and other legacy languages and frameworks. Specifically, popular coding benchmarks such as HumanEval and MBPP do not provide support for JSPs. Hence, LLM-as-judge metrics were evaluated with customized prompts as a workaround. Details are provided in the following sections.

Dataset Evaluation

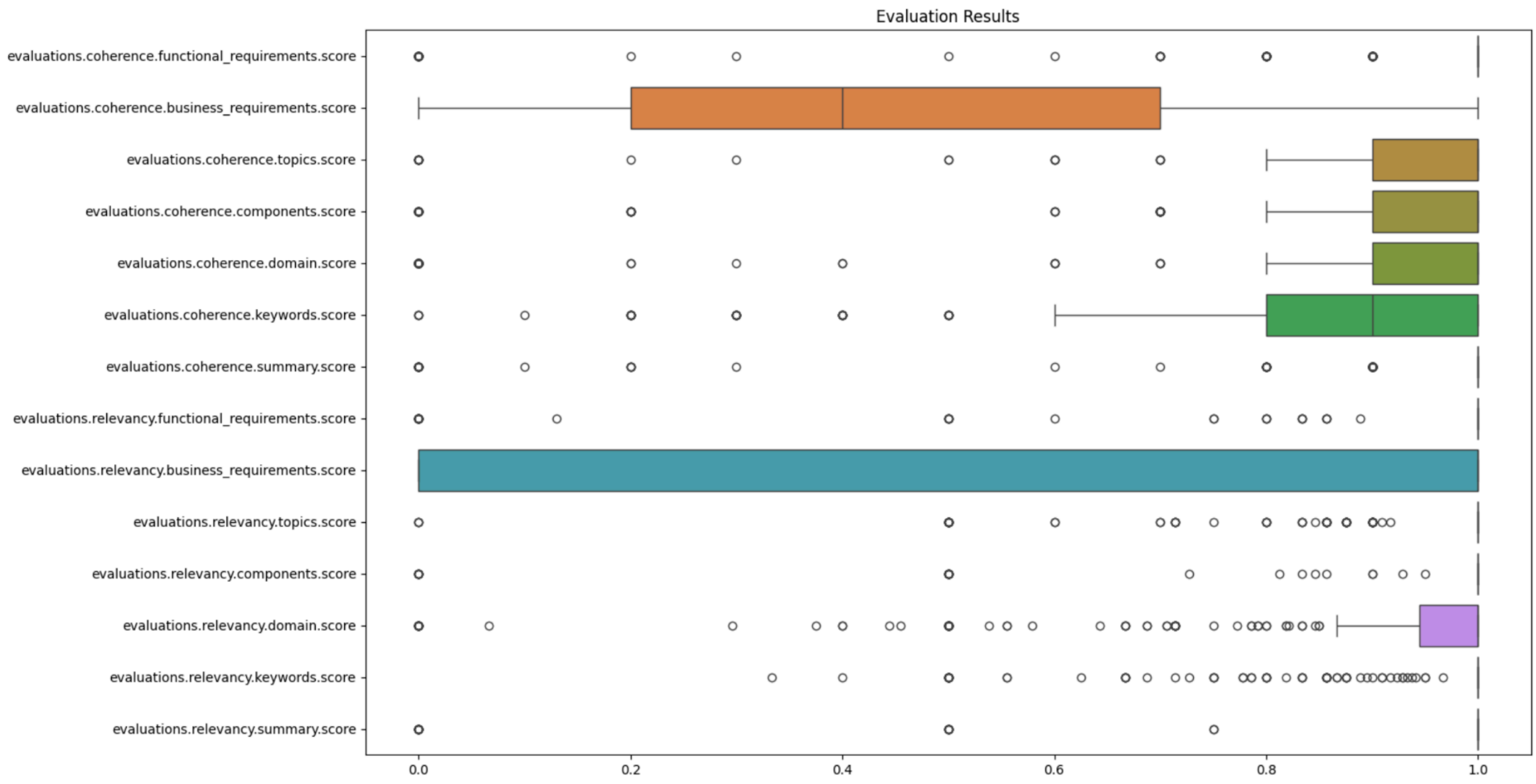
The finetuning dataset was evaluated and filtered using the following metrics:

- **Answer Relevance:** An LLM-as-judge metric for addressing whether the answer addresses the provided prompt.
- **Coherence:** An LLM-as-judge metric for addressing whether the answer flows logically from the provided prompt.

The main challenge with the dataset evaluation metrics is that LLM-as-judge metrics are non-deterministic, and can underestimate or overestimate results based on the quality of the underlying prompts. To mitigate this, some work was done to optimize the prompts used for these metrics. The full set of optimized prompts used for this project are outlined in the List of Datasets, and a sample of the prompts used has been provided in Appendix 1.

The following is a box-and-whiskers plot of the evaluation features in the dataset prior to pre-processing:

Diagram 3: Evaluation Results



When analyzing the evaluation scores, it was observed that they were mostly **left skewed**. This implies that **most of the population scores well on most of the evaluation metrics**, which is the preferred behavior for a quality dataset. The main exception to the left skewedness are the

business requirements metric scores. Data points that did not exhibit a left-skewed distribution were found to have performed poorly on most of the evaluation metrics, and were mostly filtered out in the Published Baseline.

Model Evaluation

The candidate models were evaluated using both **pointwise** and **pairwise** metrics. Pointwise metrics evaluate each data point independently by assigning an absolute score, while pairwise metrics compare two data points and assign a ranking score.

Specifically, the metrics that were evaluated include the following:

- **BLEU-4:** The BLEU metric is a precision metric that measures the overlap between 4-grams in the generated text and the reference text. It is best suited for accuracy-sensitive tasks such as certain kinds of machine translation and code generation. For this task, BLEU is a somewhat limited metric by itself as it does not capture semantic meaning.
- **ROUGE-L:** ROUGE-L is a subset of the ROUGE metric family. It is a recall-focused metric which measures how much of the reference text exists in the generated text. It is best suited for tasks like extractive text summarization, although it is also limited in its ability to capture semantic meaning.
- **METEOR:** METEOR extends the exact keyword approach of BLEU by incorporating some semantic features such as synonymy, stemming and paraphrasing. It also accounts for both precision and recall by generating a score that incorporates the harmonic mean of the two values, similar to the F1 score.
- **BERTScore:** BERTScore uses pre-trained contextual embeddings to provide deeper semantic similarity scores. Overall, its performance compared to human evaluators is better than the aforementioned traditional metrics, which mostly provide similarity scores based on lexical overlap.

- **Summarization by G-Eval:** This is an LLM-as-judge metric which is used to score the quality of a summarization generated text compared to a reference text.
- **Arena by G-Eval:** This is also an LLM-as-judge metric that does pairwise comparison of two challenger texts and assigns a winning text based on some background information. It is similar to A/B testing with human evaluation.

Simple Baseline

Model Selection

The selected baseline model was the IBM Granite 4 Tiny model [10]. It was selected for its lightweight size and compute footprint, open source licensing, tool calling capabilities and long-context support (128,000 max context length), making it suitable for budget-sensitive code generation tasks.

Finetuning Implementation

For this project, the baseline model was trained using the LoRA finetuning method. LoRA is a parameter efficient finetuning (PEFT) method that works by freezing the baseline model's original weights, and adding trainable, low-rank matrices as a separate adapter. It is a popular approach for compute efficient finetuning compared to full finetuning.

The ibm-granite/granite-4.0-h.tiny-d model was finetuned using the HuggingFace trl and Trainer libraries, which are finetuning libraries that abstract away some of the boilerplate code involved in finetuning Pytorch models.

- The **Optuna** hyperparameter tuning framework was used to select optimal hyperparameters for several of the hyperparameters used for training. A few of the automated hyperparameters were the **learning rate**, **dropout rate** and **batch size**.

- **Early Stopping** was used as an automated approach to stop the training process early. With early stopping, the trainer aborts the process prior to the indicated number of epochs if the validation loss indicates overfitting, which is indicated when the training loss is improving, but the validation loss is stagnant or gets worse.
- The **load_best_model_at_end=True** setting allows the Trainer to store a checkpoint for the model at specified intervals (at the end of each epoch, in this case). Using this, it loads the best checkpoint into the model after the training process is complete. In this sense, the Trainer takes care of selecting the best model checkpoint generated by the training process.

See Appendix 3a for the loss curve associated with the LoRA candidate model.

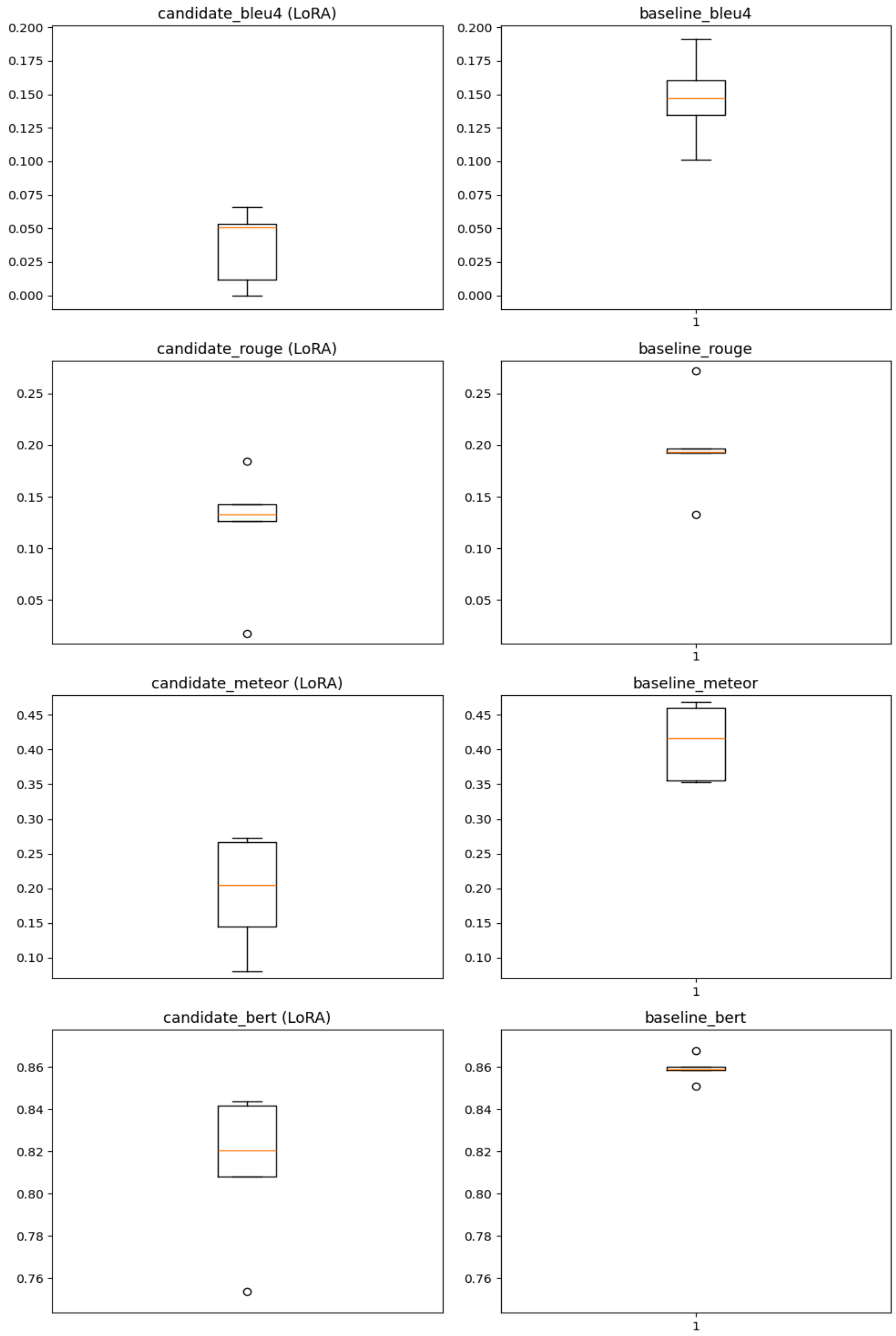
The loss curve for the LoRA candidate model showed a normal downward trending, elbow-shape, indicating a healthy training process with no signs of overfitting or underfitting.

A snippet of the central code that was used to finetune the model is provided in the Appendix, and the full code is available on Github [11].

Evaluation Results

The LoRA candidate model was evaluated using the **Wilcoxon signed-rank test**. The Wilcoxon signed-rank test is a nonparametric test that uses the rank scores of a pair of observations to evaluate whether the difference between them is statistically significant. It is often used for before-and-after comparative analysis, including the comparative analysis of two models, and is preferred for non-normally distributed data or small sample sizes. This makes the Wilcoxon signed-rank test an appropriate metric for the baseline evaluation, as its experiments were based on a small sample size of 5 git repositories. To compute the rankings, the metrics outlined in Model Evaluation were used to compare the performance of the baseline model against OpenAI’s gpt-oss-20b model, a larger, frontier model which was used as the reference model.

The following is a side-by-side box plot showing the evaluation results of the LoRA candidate model compared to the baseline model:


candidate_bleu4 (LoRA)
baseline_bleu4
candidate_rouge (LoRA)
baseline_rouge
candidate_meteor (LoRA)
baseline_meteor
candidate_bert (LoRA)
baseline_bert

Similarly, the following table shows the percent improvement of the LoRA candidate model over the baseline model:

Metric	**Percent Improvement over Baseline Model (LORA)**
BLEU4	-66.99
ROUGE-L	-27.30
METEOR	-43.17
BERT	-3.55

The results show that the overall performance of the LoRA model **decreased** compared to the baseline. This would suggest that the implementation of LoRA used in the project may not be the ideal finetuning approach for legacy code refactoring.

It is possible that the **rank** used for the LoRA finetuning process was too small to handle the complexity of code generation tasks. **Rank** is a hyperparameter that controls the size of the inner dimension of the two low-rank matrices that contain the trainable parameter: a lower rank means lower inner dimension and less trainable parameters, and vice versa. LoRA is known to struggle with tasks that require higher rank.

Experimental Results

Published Baseline

The baseline was finetuned using the DoRA finetuning strategy. DORA is a parameter efficient finetuning (PEFT) method that works by decomposing the trainable parameters into magnitude and direction components, updating the directional component using low-rank updates (similar to LoRA), and training the magnitude component separately. It has gained recent popularity over other similar methods due to its ability to provide better accuracy results even with similar training and inference costs.

Finetuning Implementation

The ibm-granite/granite-4.0-h.tiny-d model was finetuned using the HuggingFace trl and Trainer libraries, similar to LoRA.

A snippet of the central code that was used to finetune the model is provided in the Appendix 1.

Models

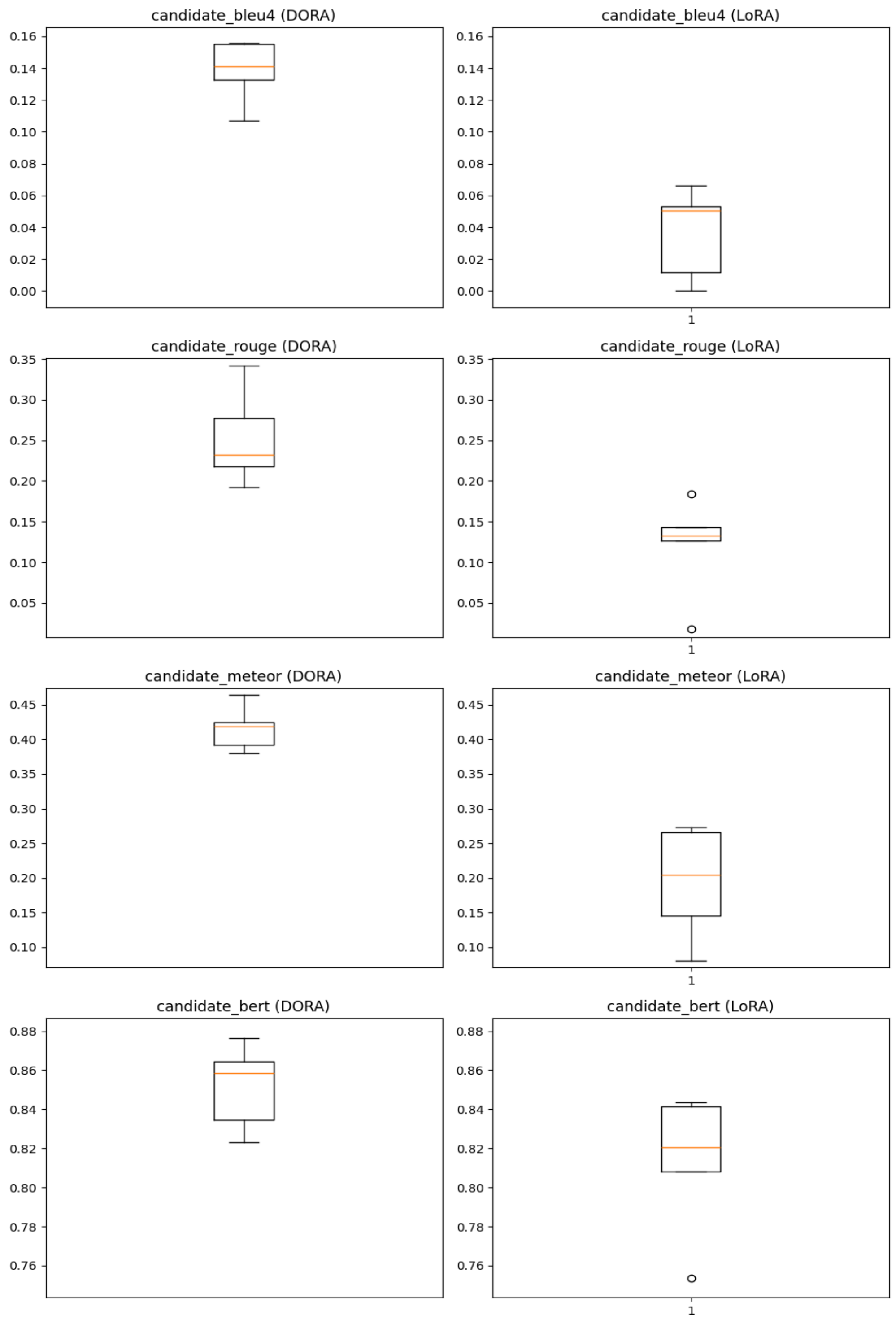
See Appendix 3b for loss curves associated with the DoRA model finetuning.

Similarly to the LoRA model, the finetuning process showed no signs of overfitting or underfitting.

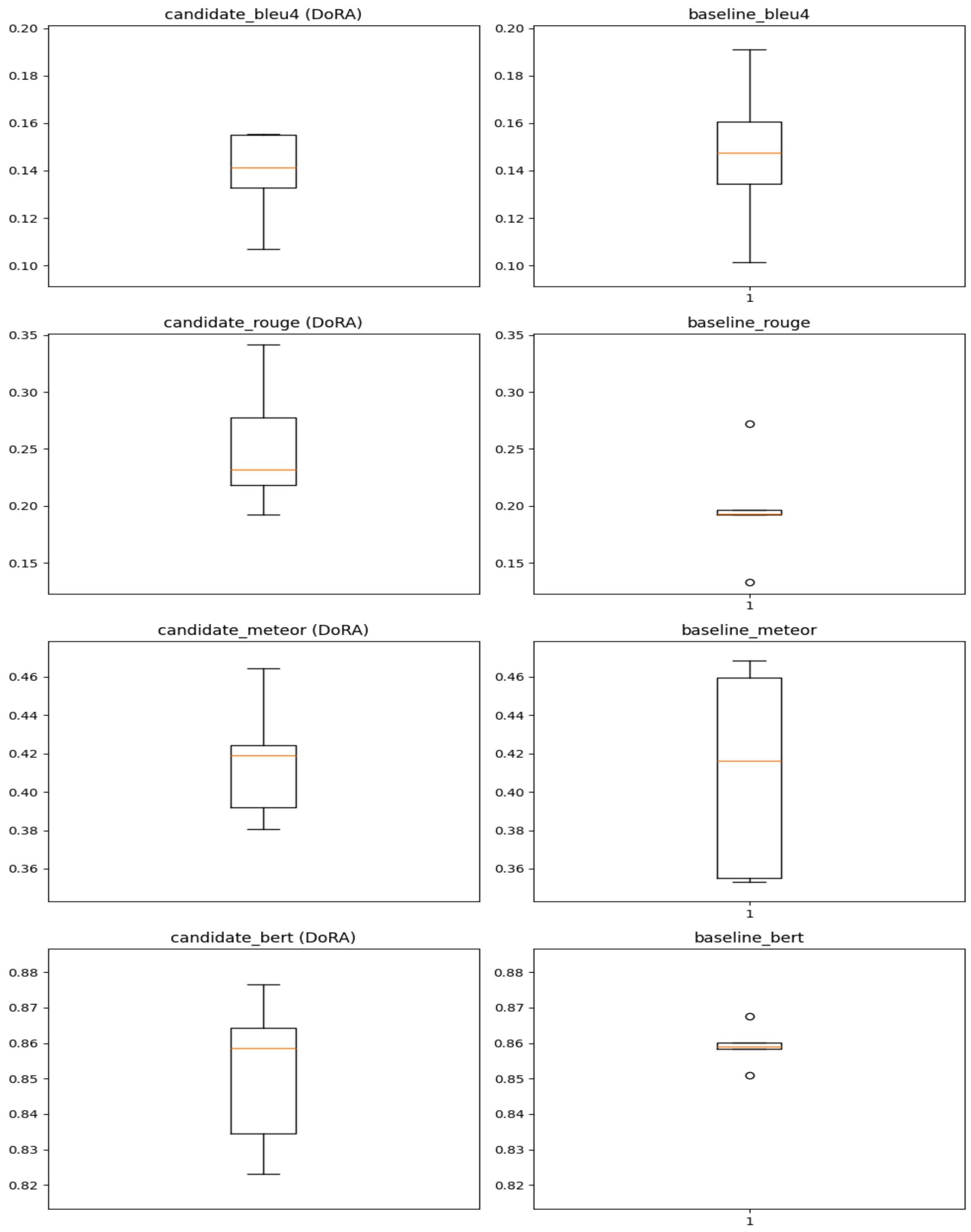
Evaluation Results

Similar to the LoRA model, the DORA candidate model was evaluated for statistical significance using the **Wilcoxon signed-rank test**, and OpenAI's gpt-oss-20b model was used as the reference model. More details about the statistical test selection are available in the Simple Baseline Evaluation Results section.

The following is a side-by-side box plot showing the evaluation results of the DoRA candidate model compared to the LoRA model:



Similarly, the following is a side-by-side box plot showing the evaluation results of the DoRA candidate model compared to the baseline model:



The results of the baseline-to-DORA evaluations show the following:

- The median performance improvement for the DoRA candidate model with the ROUGE-L metric was **18.2%**. There was a relatively low performance improvement with the other scores, and a performance degradation with BLEU4.
- Significantly, the ROUGE-L metric is the most focused on syntax-based summarization compared to the other metrics. This means that the ROUGE-L metric is an important metric for this task, as syntax-based summarization is likely to be more relevant in code summarization cases than semantic-based summarization. Consequently, it can be inferred that this project was able to demonstrate up to 18.2% performance improvement in code-relevant metrics with DORA finetuning.
- The Wilconox signed rank scores do not show statistical significance when the DoRA model is compared to the baseline model. It is likely that the small sample size was not enough to establish statistical significance.

The results of the LORA-to-DORA evaluations show the following:

- The median performance improvement for METEOR, BLEU4 and ROUGE was significant. The range of the percent improvement was from **85.5%** to **166.2%**. This translates to an approximately **2-2.5X** fold median performance improvement with the DoRA candidate model.
- There was a relatively low performance improvement with the BERT Score compared to the other scores. However, it is notable that the other metrics are more syntactic-based than semantic-based, and the low scores may be related to the fact that the BERTScore

has been tailored for natural language and not coding samples. (CodeBERT has been trained with coding samples, but does not provide official support for JSPs).

- Also, the performance improvement was **statistically significant.** The Wilconox signed-rank scores showed that the performance improvement was statistically significant for **p-value=0.10**, showing strong statistical significance.

The following table shows the Wilconox signed-rank scores for the LoRA-to-DoRA evaluation results:

Metric	**p_value**	**Is_significant (alpha=0.10)**
BLEU4	0.0625	True
ROUGE-L	0.0625	True
METEOR	0.0625	True
BERT	0.0625	True

More results are shown under Appendix 4: Published Baseline Evaluation Results.

Extensions

The following extensions were applied:

1. Prompt tuning
   The original prompts used to create the reference dataset were generating noisy datasets that included a lot of irrelevant code segments. Several iterations were required to generate comprehensive reference sets for the 5 curated git repositories used in this project.

2. Merging PEFT weight adapters
   Due to a known issue with the Granite 4 Tiny model, as well as the lack of DoRA support that was the case during the scope of this project, the LoRA and DoRA models were

merged with their weight adapters in order to support their deployment in a timely manner.

Error Analysis

The following error analysis was performed:

- 25% of records scored 0.6 or less on all metrics except business requirements and coherence metrics. Hence, all records that scored **less than 0.6** were truncated.
- 16 records showed **THIS IS NOT VALID CODE** as their generated outputs, indicating that the LLM-as-judge evaluator assessed that their associated code snippets were invalid. These records were also truncated.
- The final dataset retained **260** records, which were serialized as a **jsonl file** and published to HuggingFace. An **80-20** training-test split was applied to the data.

Future Research

- Model and dataset evaluations for this project were carried out without any formal benchmarks due to the lack of standardized benchmark sets for JSPs (and other, similar legacy programming languages and frameworks). In particular, coming up with a comprehensive ground truth / reference dataset for the evaluations was a laborious exercise that involved several rounds of prompt tuning. Benchmarks provide a more standardized and objective approach to evaluation, and will be necessary for broader adoption of GenAI-based legacy code refactoring. It would be worth exploring a suitable approach for designing benchmarks for legacy frameworks like JSPs.
- For more comprehensive results, a larger sample size of legacy JSP repositories should be explored. The search for publicly available legacy JSP repositories was a challenge

for this project, and expanding the sample set would require a more significant time investment.

- Model and dataset evaluations for this project were carried out without any formal benchmarks due to the lack of standardized benchmark sets for JSPs (and other, similar legacy programming languages and frameworks). As a result, coming up with a comprehensive ground truth / reference dataset for evaluations was a laborious exercise that involved several rounds of prompt tuning. Benchmarks provide a more standardized and objective approach to evaluation, and will be necessary for broader adoption of GenAI-based legacy code refactoring. It would be worth exploring a suitable approach for designing benchmarks for legacy frameworks like JSPs. For instance, standardized tests of questions and answers similar to CodeCriticBench and StackEval could be explored.

Conclusions

This project was able to demonstrate that using DoRA for finetuning open source models can result in up to **18%** median percent improvement in code-relevant evaluation metrics for code legacy transformation. It also showed that it can outperform LoRA finetuning by a factor of 2-2.5X in median performance across all metrics. It suggests that DoRA should be considered as a preferred finetuning approach for iterative performance improvement in legacy code transformation with JSPs.

In conclusion, this project provides a framework that could be used for improving the performance of open source legacy code translation models. It shows that there is demonstrable potential for the use of AI-enhanced legacy code refactoring for users that might be budget conscious, timeline constrained, or burdened with large amounts of technical debt. By evaluating specific problem areas and using finetuning to improve performance in those areas, it is possible to achieve continuous improvement of the candidate model’s performance using open source models and cost-efficient finetuning.

References

1. G2 Research. "AI Code Generation Drives Developer Interest." G2 Insights, 2024, https://research.g2.com/insights/ai-code-generation-drives-developer-interest. Accessed: November 19, 2025.
2. Wang, Minlue, et al. "A Probabilistic Address Parser Using Conditional Random Fields and Stochastic Regular Grammar," 2016 IEEE 16th International Conference on Data Mining Workshops (ICDMW), Barcelona, Spain, 2016, pp. 225-232, doi: 10.1109/ICDMW.2016.0039.
3. "Evolution of Neural Networks to Large Language Models." Labellerr, www.labellerr.com/blog/evolution-of-neural-networks-to-large-language-models. Accessed: November 17, 2025.
4. "A Comparative Study on Code Generation with Transformers." arXiv, 2024, arxiv.org/html/2412.05749v1.
5. Sun, Weisong, et al. "Source Code Summarization in the Era of Large Language Models." arXiv, 2024, arXiv:2407.07959.
6. Gunasekar, Suriya, Yi Zhang, Jyoti Aneja, Caio César Teodoro Mendes, Allie Del Giorno, Sivakanth Gopi, Mojan Javaheripi, Piero Kauffmann, Gustavo de Rosa, Olli Saarikivi, Adil Salim, Shital Shah, Harkirat Singh Behl, Xin Wang, Sébastien Bubeck, Ronen Eldan, Adam

Tauman Kalai, Yin Tat Lee, and Yuanzhi Li. "Textbooks Are All You Need." arXiv, 2023, arXiv:2306.11644.

7. Liu, Shih-Yang, Chien-Yi Wang, Hongxu Yin, Pavlo Molchanov, Yu-Chiang Frank Wang, Wan-Ling Chao, and Jan Kautz. "DoRA: Weight-Decomposed Low-Rank Adaptation." arXiv, 2024, arXiv:2402.09353.
8. Wu, Yang, et al. "Can Large Language Models Serve as Evaluators for Code Summarization?" arXiv, 2024, arXiv:2412.01333.
9. Awofolu, Omotola. "Code Generation EDA." GitHub, 2025, https://github.com/agapebondservant/code-generation-capstone/blob/main/eda/code_generation_eda.ipynb.
10. Granite Team, IBM. "Granite-4.0-H-Tiny." *Hugging Face*, 2 Oct. 2025, huggingface.co/ibm-granite/granite-4.0-h-tiny.
11. Awofolu, Omotola. "Code LLM Finetuning." GitHub, 2025, https://github.com/agapebondservant/code-generation-capstone/blob/main/eda/code_llm_finetuning_dora.ipynb.
12. Hall, Marty. "Core Servlets and JavaServer Pages, Volume 1: Core Technologies." 2nd ed., Prentice Hall, 2003.

Appendix

Appendix 1: Sample Prompts

Purpose	**Text**

Code-to-Summary	`Here is some code:`
	`{inputs}`

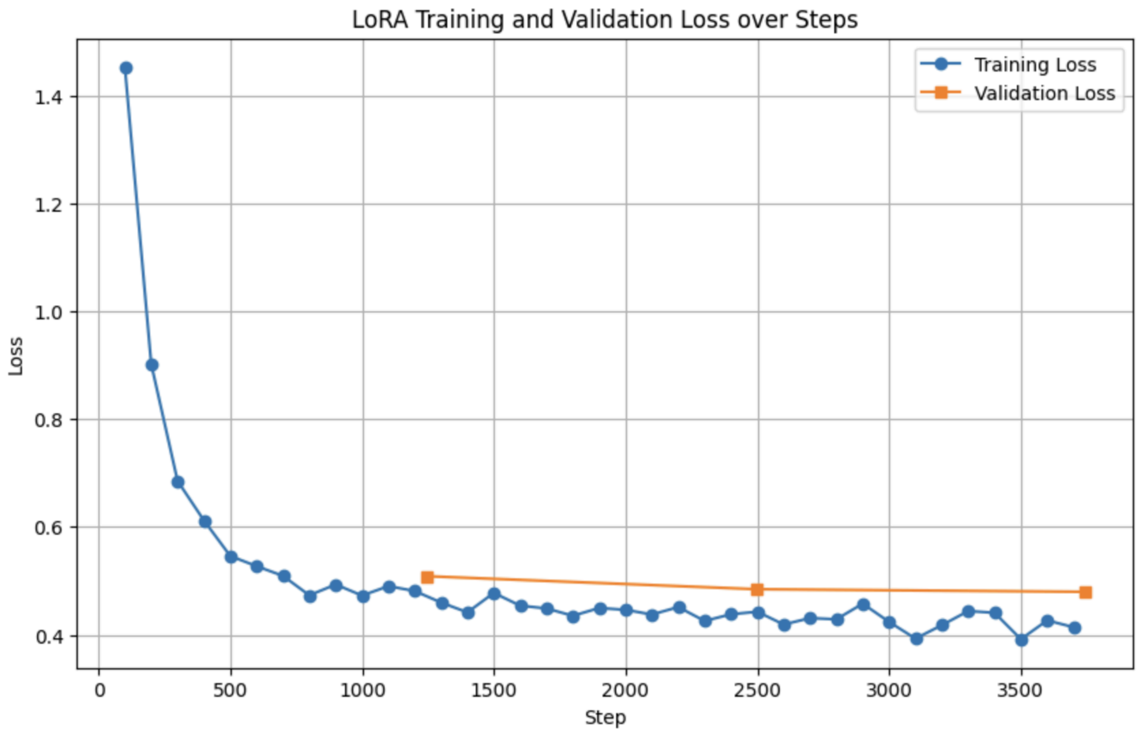
	{summary} Your task is to analyze this code and provide a concise explanation of the purpose of the code. If the provided snippet does not appear to be valid code, indicate that this is not valid code. Stop after you have finished writing the 3 sections described below. Your analysis must include the following sections: **Summary:** Provide a clear and concise explanation of the purpose of the code without getting into too many specific technical details. **Components:** Provide a concise outline of all the important JSP relevant components you can find. **Domain:** Generate a concise outline of the domain model associated with this code, including the current state of the domain objects based on information extracted from the code.
Summary-to-Spec	Here is a brief code summary: {inputs} Using the summary above, generate a language-agnostic Software Design Document (SDD). The document should be structured, professional, and suitable for both technical and non-technical stakeholders. Also exclude any details that link the requirements to JSP, Java or any other specific programming language or framework. Instead, it should focus on universal concepts and architecture that can be implemented in any programming language. The SDD must include the following sections: **1. System Architecture** * **Overall Design:** Describe the main architectural patterns used and how the different components interact. * **Key Components:** Detail the primary modules, classes, domain model and services and their responsibilities. **2. Functional Requirements** * **Input Handling:** How does the system accept inputs? * **Data Processing:** Detail the main logic, algorithms, and data transformations. * **Output Generation:** How are results produced and presented?

	3. Business Requirements * **Business Rules:** Specific rules that govern how the business operates, which the software must enforce. * **Success Criteria:** Measurable criteria to determine if the project is successful, also known as acceptance criteria. Only use the context provided in the summary above. Do not stray from the context provided in the summary. Include named components and objects based on the context where it makes sense.
Spec-to-Code	Here is a software specification: {spec} Using this specification, develop a complete NodeJS and React application based on the provided document. The application should include both backend and frontend components with comprehensive unit testing using mock objects. Backend Requirements (NodeJS): 1. Implement a RESTful API using Express.js or your preferred NodeJS framework. 2. Follow the services, data models, and business logic defined in the spec. Do NOT just develop a generic application. Frontend Requirements (React): 1. Build responsive UI components in React. 2. Develop the components according to the design specifications in the spec. Do NOT just develop a generic application. Unit Testing Requirements: Backend Tests: 1. Write unit tests for all services, controllers, and utility functions. 2. Use a mock testing framework like Jest (or Mocha/Chai) as the testing framework. Mock database calls, external API requests, and dependencies. 3. Aim for high code coverage (70%+ minimum). Frontend Tests: 1. Write unit tests for the React components. 2. Use a mock testing framework like Jest and React Testing Library as the testing framework. 3. Aim for high code coverage (70+% minimum).

Appendix 2: Parameter Efficient Finetuning (PEFT) Code Snippet

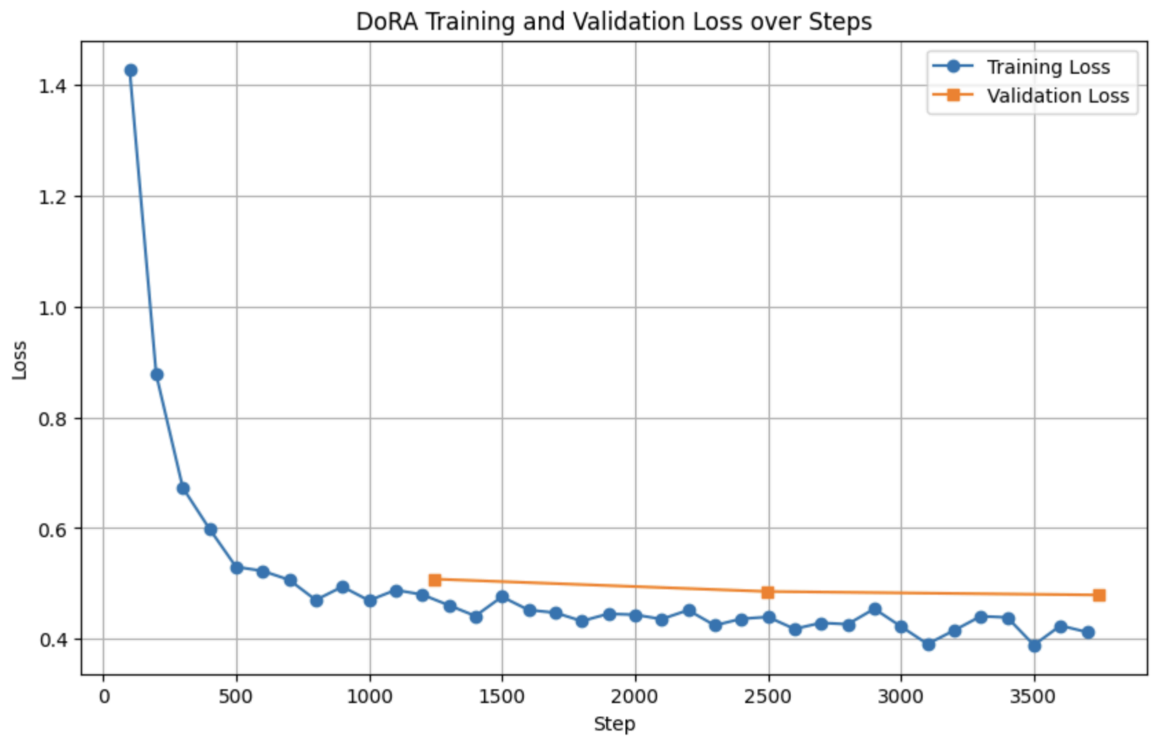
```
training_args = SFTConfig(

                    output_dir=f"{MODEL_DIR}/{base_model_dir}/experiment",

                    learning_rate=learning_rate,

                    per_device_train_batch_size=per_device_train_batch_size,

                    per_device_eval_batch_size=per_device_train_batch_size,

                    num_train_epochs=num_train_epochs,

                    logging_steps=100,

                    fp16=True,

                    report_to="none",

                    eval_strategy="epoch",

                    save_strategy="epoch",

                    load_best_model_at_end=True,

                    metric_for_best_model="eval_loss",

                    greater_is_better=False,

                    max_length=8192,

                    packing=False,

                    seed=42,
                )

 ##############################################################################
 # Supervised Finetuning Trainer
 ##############################################################################

 trainer = SFTTrainer(

     model=get_peft_model(

         model,

         lora_config
     ),

     args=training_args,

     train_dataset=train_dataset,

     eval_dataset=test_dataset,

     peft_config = lora_config,
     formatting_func = code_text_formatter,

     data_collator = collator,

     callbacks=[early_stopping_callback],
 )

 trainer.train()
```

Appendix 3a: Loss Curve Associated with LoRA Model



Appendix 3b: Loss Curve Associated with DORA Model

Diagram 4: DoRA Training and Validation Loss over Steps



Appendix 4: Published Baseline Evaluation Results

The following table shows the percent improvement of the LoRA candidate model over the DoRA model:

Percent improvement of the LoRA candidate model over DoRA candidate model	
Metric	**Percent Improvement over Baseline Model (LORA)**
BLEU4	166.24
ROUGE-L	85.55
METEOR	107.71
BERT	3.89
Percent improvement of the LoRA candidate model over baseline model	
Metric	**Percent Improvement over Baseline Model (LORA)**
BLEU4	-12.12

ROUGE-L	18.29
METEOR	0.66
BERT	-0.05