CPP-UT-Bench: Can LLMs Write Complex Unit Tests in C++?

Vaishnavi Bhargava[1*], **Rajat Ghosh**[2], **Debojyoti Dutta**[2]
[1]University of Wisconsin-Madison, [2]Nutanix
vbhargava3@wisc.edu, {rajat.ghosh, debojyoti.dutta}@nutanix.com

Abstract

We introduce CPP-UT-Bench, a benchmark dataset to measure C++ unit test generation capability of a large language model (LLM). CPP-UT-Bench aims to reflect a broad and diverse set of C++ codebases found in the real world. The dataset includes 2,653 {code, unit test} pairs drawn from 14 different opensource C++ codebases spanned across nine diverse domains including machine learning, software testing, parsing, standard input-output, data engineering, logging, complete expression evaluation, key value storage, and server protocols. We demonstrated the effectiveness of CPP-UT-Bench as a benchmark dataset through extensive experiments in in-context learning, parameter-efficient fine-tuning (PEFT), and full-parameter fine-tuning. We also discussed the challenges of the dataset compilation and insights we learned from in-context learning and fine-tuning experiments. Besides the CPP-UT-Bench dataset and data compilation code, we are also offering the fine-tuned model weights for further research. For nine out of ten experiments, our fine-tuned LLMs outperformed the corresponding base models by an average of more than 70%.

1 Introduction

Large Language Models (LLMs) [29] have demonstrated impressive performance on a number of recently proposed coding benchmarks such as HumanEval [28], MBPP, [25], and MultiPL-E [27]. Nonetheless, existing benchmarks, in general, have reached saturation [32, 36] and lack representation from real-world software engineering tasks [37]. Evaluating coding performance on short and self-contained algorithmic tasks, existing coding benchmarks such as MBPP are far from the real-world software engineering tasks such as unit test writing. Moreover, the existing coding benchmarks mostly cover high-level languages such as Python. Lower-level languages (e.g., C, C++) have higher Kolmogorov complexity [33] and cyclomatic complexity [34] due to its verbosity, advanced features (e.g., templates, macros), and manual memory management. Therefore, a C++ codebase is harder to maintain and stands to benefit considerably from unit test generation automation. However, there is hardly any benchmark dataset for C++ unittest generation representative of the real world software engineering.

Inspired by this challenge of the lack of C++ unit test generation benchmark dataset, we introduce CPP-UT-Bench from diverse domains. We evaluate multiple state-of-the-art LLMs on CPP-UT-Bench and study their performances for few-shot in-context learning, parameter-efficient fine-tuning (PEFT), and full-parameter fine-tuning.

[*]Work done during internship at Nutanix

38th Conference on Neural Information Processing Systems (NeurIPS 2024).

2 CPP-UT-Bench

CPP-UT-Bench is a benchmark featuring 2,653 {code, unittest} pairs from from 14 popular open-source C++ repositories with permissible licenses. CPP-UT-Bench is organized in the following schema:[2] [3]

- **ID**: A unique identifier for each entry in the dataset. [Example: "0"]
- **Language**: The programming language of the file. [Example: "cpp"]
- **Repository Name**: The name of the GitHub repository, formatted as organisation/repository. [Example: "google/googletest"]
- **File Name**: The base name of the file (without extension) where the code or test is located. [Example: "sample1"]
- **File Path in Repository**: The relative path to the file within the GitHub repository. [Example: "googletest/samples/sample1.cc"]
- **File Path for Unit Test**: The relative path to the unit test file, if applicable. [Example: "googletest/samples/sample1_unittest.cc"]
- **Code**: The code content of the file, excluding any documentation or comments.
- **Unit Test (Ground Truth)**: The content of the unit test file that tests the code.

We collected this data from GitHub. Although GitHub is a rich data source for software engineering, not all codebases have sufficient unit test coverage. Also, the relationship between code and unit test is often noisy, ad-hoc, and poorly documented. Our data curation pipeline is designed to be generic and adaptable, making it applicable to diverse C++ codebases. To compile a high-quality C++ unit test generation benchmark at scale, we use the following two-step pipeline, as shown in Figure 1. [4]

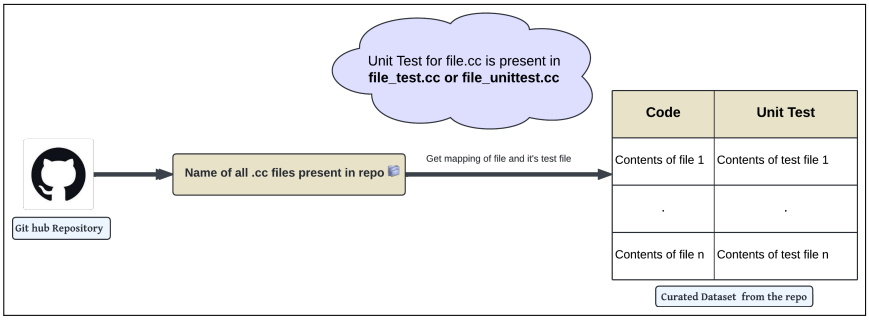

Figure 1: Data extraction pipeline for CPP-UT-Bench. It uses GitHub repos as upstream sources and then processes the code, unittest pairs extracted to create the benchmark dataset.

- **File Extraction and Grouping:** The initial phase involves extracting relevant files from the codebase. We concentrate on C++ source files (with extensions `.cc` and `.h`) and unit test files (with extensions `_test.cc` and `_unittest.cc`). A recursive directory search ensures comprehensive identification of these files. Once extracted, files are grouped by their base names, derived by stripping file extensions. For example, files named `Foo.cc` and `Foo.h` are grouped under the base name `Foo`, linking implementation files with their corresponding declarations. Similarly, test files are associated with their source files based on these shared base names.
- **Mapping Source Files to Test Files and Documentation:** Following the extraction and grouping of C++ source files and unit test files, we map each source file to its respective test files. When both `_test.cc` and `_unittest.cc` files are present, we prioritize `_test.cc`. This structured

[2] Huggingface link to CPP-UT-Bench dataset: `https://huggingface.co/datasets/Nutanix/CPP-UNITTEST-BENCH`

[3] Huggingface link to fine-tuning dataset: `https://huggingface.co/datasets/Nutanix/cpp_train_dataset_chat_format_less_than_8k`

[4] Python Script to create CPP-UT-Bench dataset: `https://huggingface.co/datasets/Nutanix/CPP-UNITTEST-BENCH/blob/main/data_scrape.py`

mapping is crucial for analyzing code coverage and evaluating the effectiveness of unit tests. The final stage of the process involves documenting the extracted data. For each base name, we compile detailed records of the repository name, source code content, and test code content into an Excel spreadsheet. This organized documentation enables comprehensive analysis, providing valuable insights into code coverage and the adequacy of unit tests.

2.1 Task Formulation

We evaluate CPP-UT-Bench for different tasks, as follows:

Few-Shot In-Context Learning: Few-shot in-context learning (FS-ICL) in this work refers to the setting where the model is given a few demonstrations of the task at inference time as conditioning [26], but no weight updates are allowed. As shown in Equation 1, FS-ICL takes an query, x_{test} at inference time and uses a fixed-parameter model, f_θ, along with k demonstrations, $(x_i, y_i)\}_{i=1}^{k}$, to produce a response, y_{test}. The response quality depends on the concerned LLM f_θ and the demonstration set.

$$y_{\text{test}} = f_\theta \left(\{(x_i, y_i)\}_{i=1}^{k}, x_{\text{test}} \right) \tag{1}$$

Parameter-Efficient Fine-Tuning: Parameter-efficient fine-tuning (PEFT) involves updating some subsets of weights of a pre-trained model, f_θ by training on a supervised dataset specific to a desired task. In general, at least a few thousands of labeled examples are used. While fine-tuning improves task-specific performances, it needs a large demonstration dataset. For PEFT, low-rank adaptation (LoRA) [30] is one of the most prevalent techniques, as shown in Equation 2.

$$\max_{\Theta} \sum_{(x,y)\in\mathbb{Z}} \sum_{t=1}^{|y|} \log \left(p_{\Phi_0 + \Delta\Phi(\Theta)} \left(y_t \mid x, y_{<t} \right) \right) \tag{2}$$

Full-Parameter Fine-Tuning: Full-parameter fine-tuning [35] involves updating the all weights of a pre-trained model, f_θ by training on a supervised dataset specific to a desired task. Because it is updating all the weights, it comes with much higher computational cost than PEFT/LoRA.

$$\max_{\Phi} \sum_{(x,y)\in\mathbb{Z}} \sum_{t=1}^{|y|} \log \left(P_\Phi \left(y_t \mid x, y_{<t} \right) \right) \tag{3}$$

Both PEFT and full-parameter fine-tuning are inter-related. A pre-trained LLM, $P_\Phi(y \mid x)$ is parameterized by Φ. A downstream task is represented by a training dataset of context-target pairs: $Z = \{(x_i, y_i)\}_{i=1,\ldots,N}$ where both x_i and y_i are sequences of tokens. During full fine-tuning, the model is initialized to the base weights Φ_0 and updated to $\Phi_0 + \Delta\Phi$ by repeatedly following the gradient to maximize the conditional language modeling objective as shown in Equation 3. Fine-tuning the entire pre-trained weight space could be prohibitively expensive. That is where PEFT brings value. PEFT adopts a more parameter-efficient approach, where the task-specific parameter increment $\Delta\Phi = \Delta\Phi(\Theta)$ is further encoded by a much smaller-sized set of parameters Θ with $|\Theta| \ll |\Phi_0|$. The task of finding $\Delta\Phi$ thus becomes optimizing over Θ, as shown in Equation 2.

2.2 Features of CPP-UT-Bench

Traditional code benchmarks such as MBPP typically involve only short and standalone input and output sequences. In contrast, CPP-UT-Bench represents real-world software engineering in C++. CPP-UT-Bench consists of widely popular open-source code bases such as TensorFlow. Figure 2 shows the distribution of CPP-UT-Bench in terms of source repositories. It shows the dataset has imbalanced representation from different projects with the dominant being Tensorflow. Overall, it has 2,653 pairs from 14 open-source projects with permissible licenses covering nine different domains.

The domain diversity of CPP-UT-Bench is shown in Table 2.2. It covers a wide gamut of real world software applications including machine learning, data engineering, software testing, telecommunications, key-value storage, server protocol, geolocation, concurrency, and application logging.

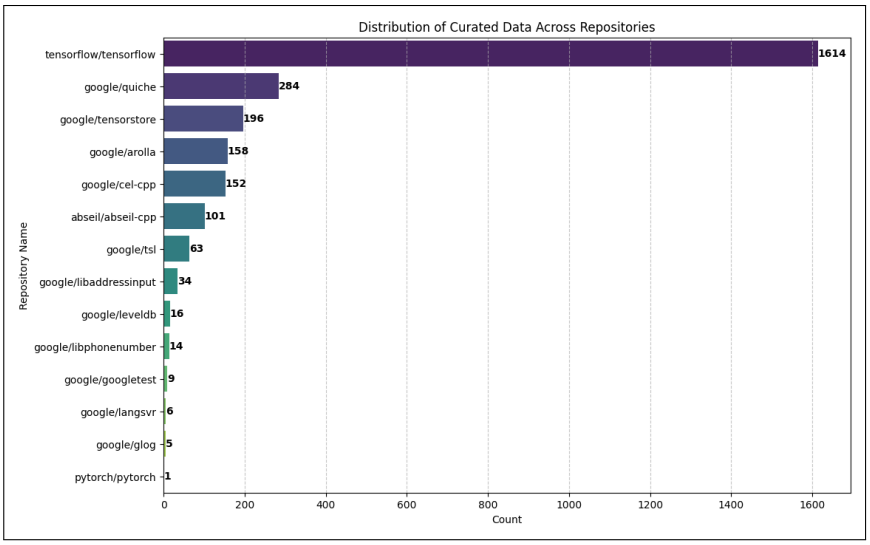

Figure 2: Data distribution of CPP-UT-Bench from 14 different GitHub Repositories. The dominant contribution (greater than 60%) comes from Tensorflow and the least from PyTorch.

Domain	**Repository**
Machine learning	Pytorch [19], TensorFlow [20]
Storage and data engineering	Tensorstore [21]
Software testing	Google Test [8], Abseil [1],
Telecommunications	Libphonenumber [12]
Key-value storage	LevelDB [10]
Server protocol	Langsvr [9], Cel-cpp [5]
Geolocation	Libaddressinput [11]
Concurrency and multi-threading	tsl [23]
Application logging	glob [7]

Table 1: Domain diversity of CPP-UT-Bench.

The distribution of lengths for {code, unit test} pairs in CPP-UT-Bench grouped by different repositories is shown in Figure 3. This shows all repositories have average line lengths greater than 100 with considerable variance and outliers, which is representative of the real-world code bases.

2.3 LLM-as-a-Judge

To evaluate LLM performances in few-shot in-context learning and fine-tuning, we have adopted LLM-as-a-Judge paradigm [38] with GPT-4o-mini as the oracle model. This choice is to avoid the shortcomings of conventional NLP metrics such as BLEU and ROUGE which fail to effectively capture the semantic similarity required for evaluating generated code [28]. Equation 4 formally describes the standardized evaluation model, $\mathcal{E}$, we follow. It evaluates a triplet, (r_A, r_B, g). r_A is the response from LLM-A. r_B is the response from LLM-B. g is the ground truth. The oracle LLM judges between $\{r_A, r_B\}$ which response is more closely aligned to g. The alignment judgement function, J and the relative comparison between two alignments is executed by the Oracle LLM itself in a zero-shot manner with the prompt template shown in Figure 4. The evaluation prompt was carefully designed to capture subtle differences between the outputs of the models and their alignment with the ground truth. To mitigate potential biases, such as GPT's preference for longer responses or positional bias, we further tuned the prompt.

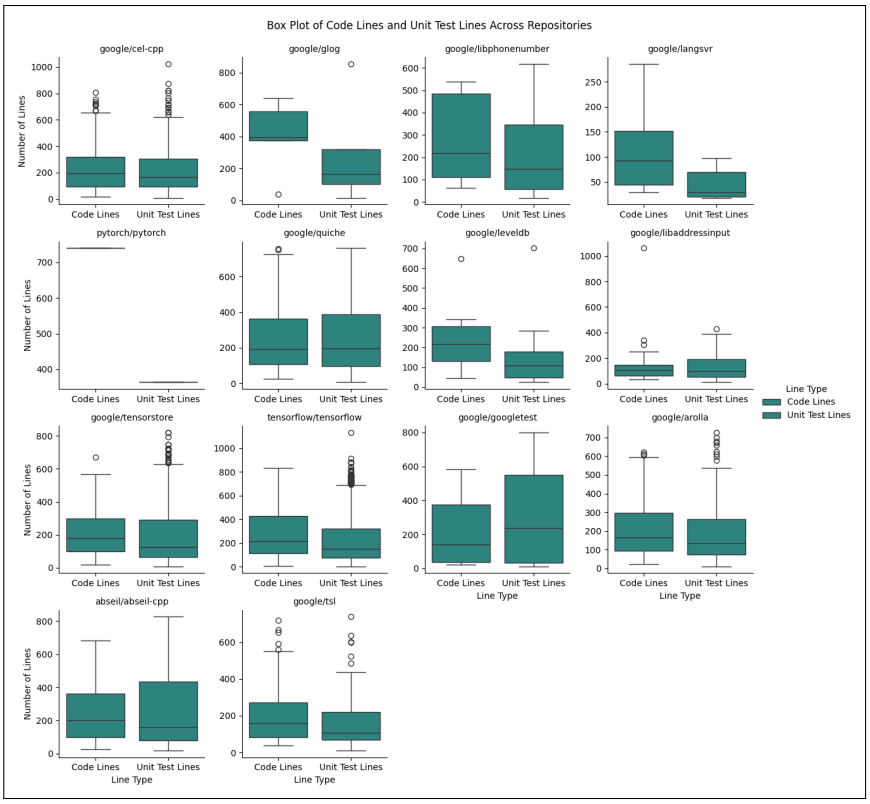

Figure 3: The diversity in {code, unit test} pairs in terms of line lengths across 14 different opensource repositories in CPP-UT-Bench.

In this pairwise approach, the win rate for an evaluation set is defined as the percentage of instances where first model's output is judged to be more closely aligned with the ground truth compared to the competing second model's output. This method is supported by numerous studies demonstrating that GPT-based evaluation closely mimics human judgment while being less expensive and time-consuming [38].

.

$$\mathcal{E}(r_A, r_B, g) = \begin{cases} r_A & \text{if } J(r_A, g) > J(r_B, g) \\ r_B & \text{if } J(r_A, g) < J(r_B, g) \\ \text{Tie} & \text{if } J(r_A, g) = J(r_B, g) \end{cases} \tag{4}$$

2.4 Framework for C++ Unit Tests Generation

We use the following three-step workflow to generate the unit test given a source file. This is an important design consideration for our work given that many of the real world C++ code-base often exceed the context length for an LLM.

1. **Code Chunker**: In scenarios where a C++ class file exceeds 200 lines, it becomes suboptimal to prompt the LLM to generate unit tests for the entire file in one go. To address this, we implemented a method that processes the code file by generating multiple smaller chunks. We leveraged the code chunker introduced by SweepAI which employ Concrete Syntax Tree (CST) based strategies [2]. Equation 5 describes CST based chunking formally with $T(r)$ is the CST for the code r and C_i is the i^{th} code chuck. It's designed to handle extremely large files by breaking them down into manageable sections that preserve the code's structure and context. This ensures each code chunk remains coherent and contextually relevant, thereby improving the accuracy and reliability of the generated unit tests.

System Prompt:

Please act as an impartial judge and evaluate the quality of the responses provided by two AI assistants to the user question displayed below. Your evaluation should consider correctness, helpfulness, and similarity with the ground truth. You will be given a reference answer, assistant A's answer, and assistant B's answer. Your job is to evaluate which assistant's answer is more aligned with ground truth. Begin your evaluation by comparing both assistants' answers with the reference answer. Avoid any position biases and ensure that the order in which the responses were presented does not influence your decision. Do not allow the length of the responses to influence your evaluation. Do not favor certain names of the assistants. Be as objective as possible. Additionally, compare the results with the ground truth and determine which model's results are more aligned with it. After providing your explanation, output your final verdict by strictly following this format:
"[[A]]" if assistant A is better, "[[B]]" if assistant B is better, and "[[C]]" for a tie.

User Prompt:

{question}
[The Start of Reference Answer]
{answer_ref}
[The End of Reference Answer]
[The Start of Assistant A's Answer]
{answer_a}
[The End of Assistant A's Answer]
[The Start of Assistant B's Answer]
{answer_b}
[The End of Assistant B's Answer]

Figure 4: Prompt for pairwise evaluation of two LLM generated responses (Assistant A and Assistant B) w.r.t. the ground truth.

$$T(r) = \{C_1, C_2, \ldots, C_n\} \tag{5}$$

2. **Unit Test Generation for a chunk**: For each chunk i, we prompt the LLM to generate the corresponding unit test, $ICL(C_i)$. Through extensive prompt engineering, we have refined the prompts to achieve optimal results. The prompt template is shown in Appendix, Figure 13.

$$UT(C_i) = \{\text{ICL}(C_i)\} \tag{6}$$

3. **Compilation of unit test chunks**: Finally, we take the generated unit tests for the chunks and simply append them to give the final unit test file. This can further be enhanced by having an LLM prompt for combining the unit tests.

$$UT(T(r)) = \sum_{i=1}^{n} UT(C_i) \tag{7}$$

3 Experiment Design

The key value of a benchmark dataset such as CPP-UT-Bench comes from its value as a test data for few-shot in-context and a demonstration dataset for PEFT and full-parameter fine-tuning.

Research Question-1 (RQ-1): Can CPP-UT-Bench replicate known results from well-known benchmarks in few-shot in-context learning? To answer this question, we compare the two-shot in-context learning performances for the following pairs: {Phi-3-medium [17] vs Phi-3-Small [18]}, {Mistral-7B-Instruct-v0.2 [16] vs Mistral-7B-Instruct-v0.1 [15] }, and {Llama-3-70B-instruct-awq [13] vs Llama-3-8B-instruct-awq [14] }.

To evaluate the two-shot performance across these models, we employed the pipeline described in Section 2.4, to generate unit tests for various code files. For inference, we configured the sampling parameters uniformly across both the original and fine-tuned models, setting a temperature of 0.1, a maximum token limit of 4,096, a frequency penalty of 0.3, and a top-p value of 0.7. We conducted preliminary experiments with various parameter values to determine these optimal settings.

We generated unit tests for each model using 200 samples from the evaluation dataset, and then applied the methodology from Section 2.3 to assess model performance. The evaluation was conducted using GPT-4o-mini as the judge, and the comparison was quantified through win rate. Figure 5 shows the distribution of evaluation data used for the few-shot in-context learning experiments. The

repository choice has been somewhat random. In future, we will perform more analysis for other data distributions.

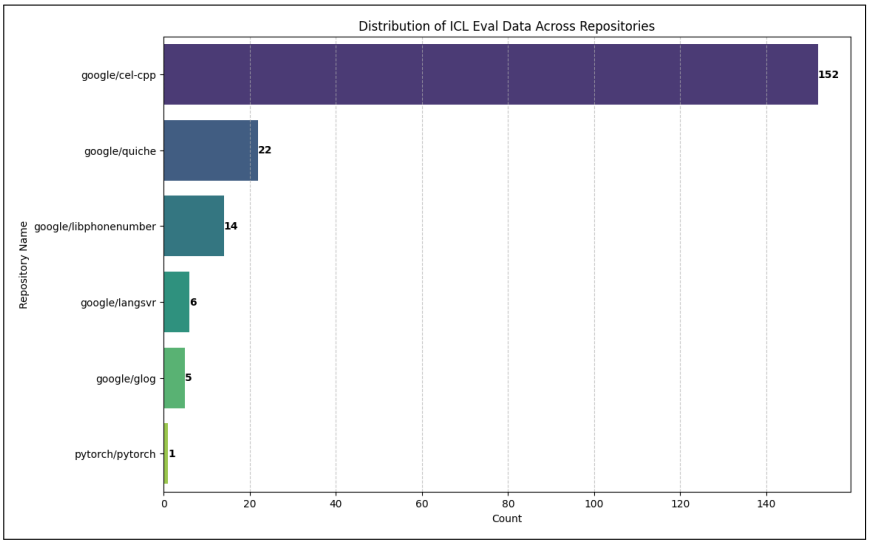

Figure 5: Distribution of evaluation dataset for the few-shot in-context learning.

Research Question-2 (RQ-2): Does a full-parameter fine-tuned LLM with CPP-UT-Bench dataset performs better than its PEFT counterpart relative to a base LLM? To answer this question, we have compared both PEFT and full-parameter fine-tuned versions w.r.t. the corresponding base versions for five LLMs, including Mistral-7B-Instruct-v0.2 [16], TinyLlama-1.1B-Chat-v1.0 [22], CodeLlama-7B-Instruct [6], Llama-3-8B-Instruct [3], and Llama-3.1-8B-Instruct [4].

- **PEFT Finetuning:** For our fine-tuning experiments, we used the Low-Rank Adaptation (LoRA) technique. Through a grid search, we optimized the LoRA parameters and found that a rank of 8 and an alpha of 16 yielded the best results. The fine-tuning was performed over two epochs on our curated dataset, with a learning rate of 5×10^{-5}. We observed that using a smaller learning rate led to more stable training. LoRA was applied to the dense layers, including the gate_proj, down_proj, and up_proj layers of the MLP block, as well as the q_proj, v_proj, k_proj, and o_proj layers in the Attention block. These layers provided the most effective results during training. The detailed hyper-parameter choices for the fine-tuning experiments are shown in Appedix (Table A).
- **Full-Parameter Finetuning:** For the full fine-tuning or domain adaptation approach, we fine-tuned all the parameters of the model. We trained for two epochs on our dataset, using a learning rate of 5×10^{-5}.

To evaluate the performance of the fine-tuned models against their original counterparts, we used the process mentioned in RQ-1. We employed the same pipeline (Section 2.4) and sampling parameters for inference, and the results were evaluated using the methodology in Section 2.3, with GPT-4o-mini as the judge, quantifying performance via win rate. Figure 6 shows the distribution of evaluation dataset for the fine-tuning experiments. The repository choice has been somewhat random. In future, we will perform a thorough ablation study.

4 Results

This section is divided into two sub-sections each for two research questions.

4.1 Results for Few-Shot In-Context Learning (RQ-1)

In few-shot in-context learning, we accessed the performance of three LLM families: Llama-3, Phi-3, and Mistral-7B-v0.2, as shown in Figure 7.

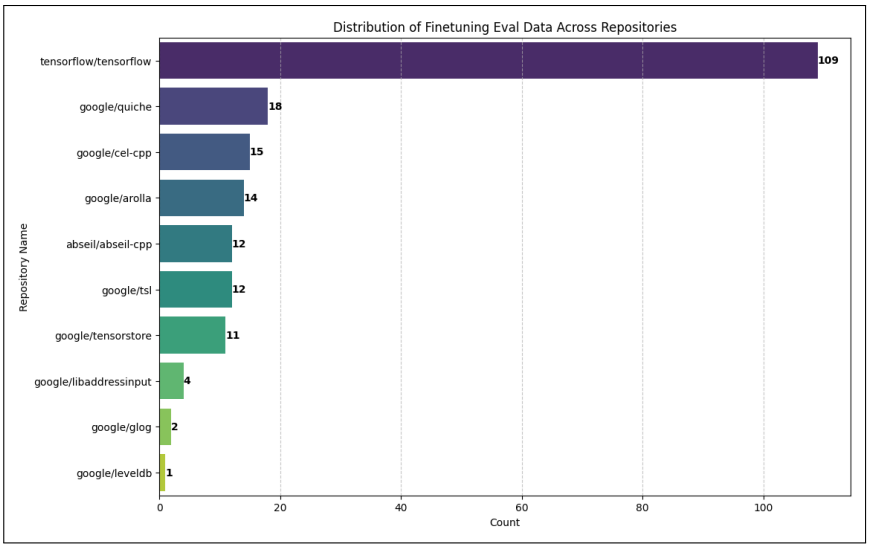

Figure 6: Distribution of evaluation dataset for the fine-tuning experiments.

Llama-3 Family: From Figure 7 (top), we see Llama-3-128K-70B is winning over Llama-3-8B 76.3% times. This can be attributed to higher context length and longer context length of Llama-3-128K-70B. This also corroborates the existing benchmarks [29].

Phi-3 Family: From Figure 7 (mid), we see Phi-3-medium is winning over Phi-3-small 58.9% times. Phi-3-small of 7B parameters and Phi-3-medium of 14B parameters are both trained for 4.8T tokens. They perform respectively 75%, 78% on MMLU, and 8.7, 8.9 on MT-bench [24]. Following similar trends, our result also show slightly superior performance for Phi-3-medium.

Mistral-7B Family: From Figure 7 (bottom), we see Mistral-7B-Instruct-v0.2 is winning over Mistral-7B-Instruct-v0.1 a whopping 91.9% times. Although both models have same parameter counts, Mistral-7B-Instruct-v0.2 several important characteristics that have possibly contributed to its superiority in C++ unit test generation. First, one of the most significant upgrades in v0.2 [15, 16] is the increase in the context window from 8k to 32k tokens. This allows the model to handle and generate longer sequences more efficiently, improving its ability to maintain context in larger inputs, especially for complex C++ unit test generation tasks. Second, the positional encoding mechanism was fine-tuned in v0.2, with the Rope-theta parameter adjusted to 10^6. This optimization allows better handling of longer token sequences in C++ unit test generation. Finally, v0.2 drops the use of sliding window attention, a mechanism used in v0.1, which limits the model's ability to capture long-range dependencies. By eliminating this feature, v0.2 improves its understanding of full input sequences, possibly contributing to enhanced unit test generation in C++.

4.2 Results for Fine-Tuning (RQ-2)

This section discusses the fine-tuning results for five different LLMs families. We hypothesize a PEFT model tuned on a task-specific demonstration data performs better than the corresponding base model for the task. Along the same line, we hypothesize a full-parameter fine-tuned model produces superior results than the corresponding PEFT counterpart.

4.2.1 Mistral-7B-Instruct-v0.2

Figure 8 shows the win-rates for Lora-PEFT Mistral-7B-Instruct-v0.2 vs Mistral-7B-Instruct-v0.2 and full-parameter fine-tuned Mistral-7B-Instruct-v0.2 vs Mistral-7B-Instruct-v0.2. It shows PEFT is working better than the base. But, quite surprisingly, the full-parameter model is performing poorly compared to the base model. This confounding observation can be explained by the MoE architecture [39].

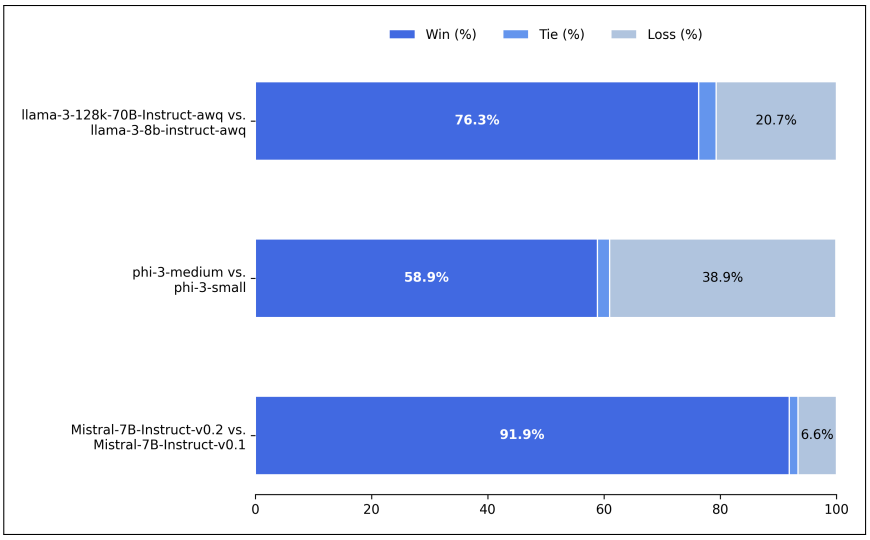

Figure 7: Few-Shot in-context learning performance assessment for three LLM pairs: {Llama-3-70B-instruct-awq [13] vs Llama-3-8B-instruct-awq [14] }, {Phi-3-medium [17] vs Phi-3-Small [18]}, and {Mistral-7B-Instruct-v0.2 [16] vs Mistral-7B-Instruct-v0.1 [15] }. The results corroborate with other general coding benchmarks [24, 16, 29, 31].

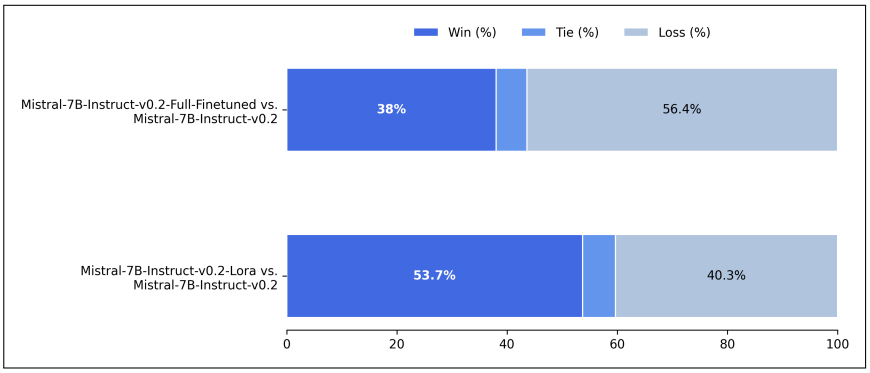

Figure 8: Fine-tuning results for Mistral-7B-Instruct-v0.2 [16]. The results corroborate with other general coding benchmarks.

4.2.2 TinyLlama

Figure 9 shows the win-rates for Lora-PEFT TinyLlama-1.1B-Chat-v1.0 vs TinyLlama-1.1B-Chat-v1.0 and full-parameter fine-tuned TinyLlama-1.1B-Chat-v1.0 vs TinyLlama-1.1B-Chat-v1.0. It shows PEFT is working better than the base, winning 77.8% times. With full-parameter fine-tuning the model performance improves further to 84.7% w.r.t. the base.

4.2.3 CodeLlama

Figure 10 shows the win-rates for Lora-PEFT CodeLlama-7B-Instruct-hf vs TinyLlama-1.1B-Chat-v1.0 and full-parameter fine-tuned CodeLlama-7B-Instruct-hf vs CodeLlama-7B-Instruct-hf. It shows both PEFT and full-parameter finetuning are performing on par with each other. This can be attributed to the relative strength of CodeLlama as a coding model, our hyper-parameter choice, $rank = 8$, and relatively small data-size.

4.2.4 Llama-3-8B

Figure 11 shows the win-rates for Lora-PEFT Meta-Llama-3-8B-Instruct vs Meta-Llama-3-8B-Instruct and full-parameter fine-tuned Meta-Llama-3-8B-Instruct vs Meta-Llama-3-8B-Instruct. It

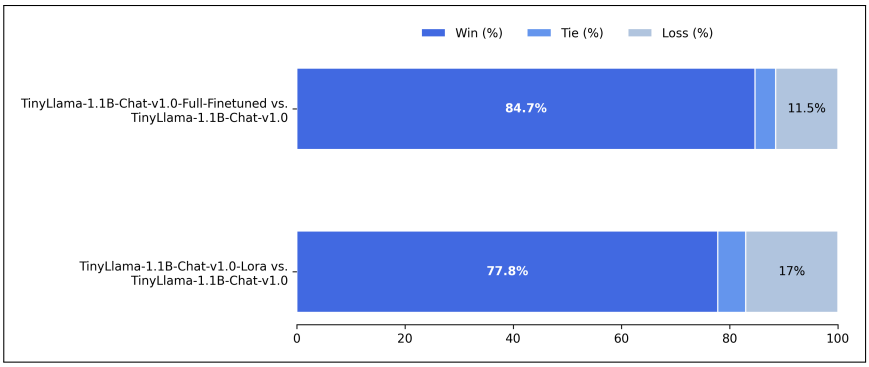

Figure 9: Fine-tuning results for TinyLlama [22]. The results corroborate with other general coding benchmarks.

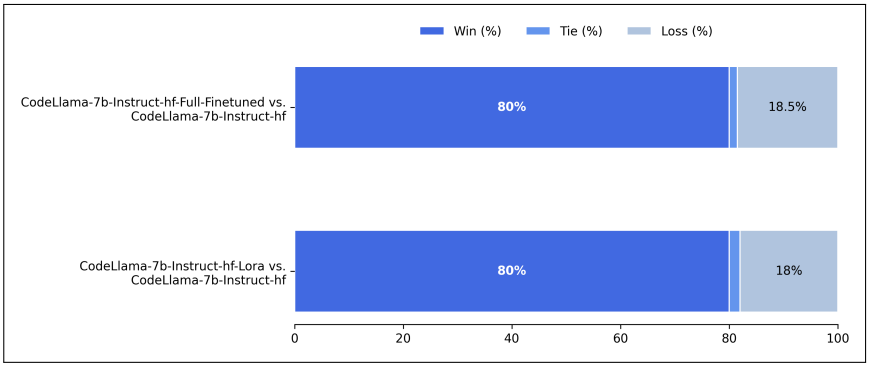

Figure 10: Fine-tuning results for CodeLlama-7B [6] . The results corroborate with other general coding benchmarks.

shows PEFT is working better than the base, winning 67% times. With full-parameter fine-tuning the model performance improves further to 75.5% w.r.t. the base.

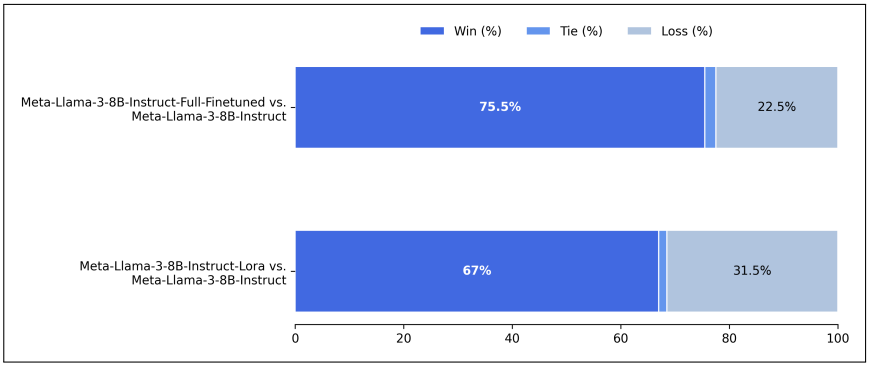

Figure 11: Fine-tuning results for Meta-Llama-3-8B-Instruct [3] . The results corroborate with other general coding benchmarks.

4.2.5 Llama-3.1-8B

Figure 12 shows the win-rates for Lora-PEFT Meta-Llama-3.1-8B-Instruct vs Meta-Llama-3.1-8B-Instruct and full-parameter fine-tuned Meta-Llama-3.1-8B-Instruct vs Meta-Llama-3.1-8B-Instruct. It shows PEFT is working better than the base, winning 52.2% times. With full-parameter fine-tuning the model performance improves further to 62.5% w.r.t. the base. The relative improvement for Llama-3.1 is lower than Llama-3 can be explained by the superiority of former in the coding benchmarks [29].

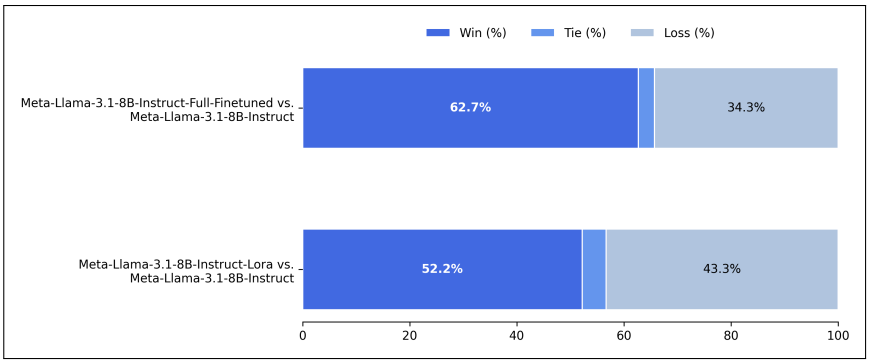

Figure 12: Fine-tuning results for Meta-Llama-3-8B-Instruct [4] . The results corroborate with other general coding benchmarks.

5 Conclusion

In this work, we offer a C++ unit test benchmark, CPP-UT-Bench. We presented its scale and diversity across domains and features. We examined the effectiveness of CPP-UT-Bench for three different task scenarios: few-shot in-context learning, parameter-efficient fine-tuning (PEFT), and full-parameter fine-tuning for different LLM families. The patterns we discovered from our examinations corroborate with existing benchmarking standards. The resulting fine-tuned LLMs with CPP-UT-Bench show significant accuracy improvement compared to the base model. Therefore, we can claim the usability of CPP-UT-Bench as a benchmark dataset in C++ unit test generation with in-context learning and fine-tuning. For reproducibility, we will release our code. The future work will extend the scope to include alignment as well.

References

[1] https://github.com/abseil/abseil-cpp.

[2] https://docs.sweep.dev/blogs/chunking-improvements.

[3] https://huggingface.co/meta-llama/Meta-Llama-3-8B, .

[4] https://huggingface.co/meta-llama/Meta-Llama-3.1-8B, .

[5] https://github.com/google/cel-cpp.

[6] https://huggingface.co/codellama/CodeLlama-7b-hf.

[7] https://github.com/google/glog?tab=readme-ov-file.

[8] https://github.com/google/googletest.

[9] https://github.com/google/langsvr.

[10] https://github.com/google/leveldb.

[11] https://github.com/google/libaddressinput, .

[12] https://github.com/google/libphonenumber, .

[13] https://huggingface.co/casperhansen/llama-3-70b-instruct-awq, .

[14] https://huggingface.co/casperhansen/llama-3-8b-instruct-awq, .

[15] https://huggingface.co/mistralai/Mistral-7B-v0.1, .

[16] https://huggingface.co/mistralai/Mistral-7B-v0.2, .

[17] https://huggingface.co/microsoft/Phi-3-medium-128k-instruct, .

[18] https://huggingface.co/microsoft/Phi-3-small-8k-instruct, .

[19] https://github.com/pytorch/pytorch.

[20] https://github.com/tensorflow/tensorflow, .

[21] https://github.com/google/tensorstore, .

[22] https://huggingface.co/TinyLlama/TinyLlama-1.1B-Chat-v1.0.

[23] https://github.com/google/tsl.

[24] Marah Abdin, Sam Ade Jacobs, Ammar Ahmad Awan, Jyoti Aneja, Ahmed Awadallah, Hany Awadalla, Nguyen Bach, Amit Bahree, Arash Bakhtiari, Harkirat Behl, et al. Phi-3 technical report: A highly capable language model locally on your phone. *arXiv preprint arXiv:2404.14219*, 2024.

[25] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

[26] Tom B Brown. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

[27] Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, et al. Multipl-e: a scalable and polyglot approach to benchmarking neural code generation. *IEEE Transactions on Software Engineering*, 49(7):3675–3691, 2023.

[28] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[29] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

[30] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021.

[31] Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. Mistral 7b. *arXiv preprint arXiv:2310.06825*, 2023.

[32] Douwe Kiela, Max Bartolo, Yixin Nie, Divyansh Kaushik, Atticus Geiger, Zhengxuan Wu, Bertie Vidgen, Grusha Prasad, Amanpreet Singh, Pratik Ringshia, et al. Dynabench: Rethinking benchmarking in nlp. *arXiv preprint arXiv:2104.14337*, 2021.

[33] M Li. An introduction to kolmogorov complexity and its applications, 2008.

[34] Mateus Lopes and Andre Hora. How and why we end up with complex methods: a multi-language study. *Empirical Software Engineering*, 27(5):115, 2022.

[35] Kai Lv, Yuqing Yang, Tengxiao Liu, Qinghui Gao, Qipeng Guo, and Xipeng Qiu. Full parameter fine-tuning for large language models with limited resources, 2024. URL `https://arxiv.org/abs/2306.09782`.

[36] Simon Ott, Adriano Barbosa-Silva, Kathrin Blagec, Jan Brauner, and Matthias Samwald. Mapping global dynamics of benchmark creation and saturation in artificial intelligence. *Nature Communications*, 13(1):6793, 2022.

[37] Aarohi Srivastava, Abhinav Rastogi, Abhishek Rao, Abu Awal Md Shoeb, Abubakar Abid, Adam Fisch, Adam R Brown, Adam Santoro, Aditya Gupta, Adrià Garriga-Alonso, et al. Beyond the imitation game: Quantifying and extrapolating the capabilities of language models. *arXiv preprint arXiv:2206.04615*, 2022.

[38] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in Neural Information Processing Systems*, 36:46595–46623, 2023.

[39] Barret Zoph, Irwan Bello, Sameer Kumar, Nan Du, Yanping Huang, Jeff Dean, Noam Shazeer, and William Fedus. St-moe: Designing stable and transferable sparse expert models. *arXiv preprint arXiv:2202.08906*, 2022.

A Appendix / supplemental material

Model Name	PEFT Technique	Rank	Alpha	Layers Targeted
Mistral-7B-Instruct-v0.2	LoRA	8	16	up_proj, o_proj, gate_proj, v_proj, k_proj, down_proj, q_proj
Llama-3-8B-Instruct	LoRA	8	16	up_proj, o_proj, gate_proj, v_proj, k_proj, down_proj, q_proj
Llama-3.1-8B-Instruct	LoRA	8	16	q_proj, v_proj, k_proj
CodeLlama-7b-hf	LoRA	8	16	up_proj, o_proj, gate_proj, v_proj, k_proj, down_proj, q_proj
TinyLlama-1.1B-32k-Instruct	LoRA	16	16	up_proj, o_proj, gate_proj, v_proj, k_proj, down_proj, q_proj

Table 2: Configurations for LoRA Fine-tuning of Different Models

B Experimental Result Reproducibility

To support the reproducibility of our experimental results, we provide links to the LoRA adapter weights and the fully finetuned model weights for each model used in our experiments. These resources allow other researchers to replicate the training procedures and fine-tuning outcomes presented in this paper.

The following table summarizes the models along with their corresponding weights:

Model Name	Links: LoRA Adapter Weights, Full Finetuned Model Weights
Mistral-7B-Instruct-v0.2	https://huggingface.co/Nutanix/Mistral-7B-Instruct-v0.2_cppunittest_lora_8_alpha_16 https://huggingface.co/Nutanix/Mistral-7B-Instruct-v0.2_cpp_unit_tests_full_finetuning_class_level
Llama-3-8B-Instruct	https://huggingface.co/Nutanix/Meta-Llama-3-8B-Instruct_cppunittest_lora_8_alpha_16 https://huggingface.co/Nutanix/Meta-Llama-3-8B-Instruct_cppunittest_full_finetuning
Llama-3.1-8B-Instruct	https://huggingface.co/Nutanix/Meta-Llama-3.1-8B-Instruct_cppunittest_lora_8_alpha_16 https://huggingface.co/Nutanix/Meta-Llama-3.1-8B-Instruct_cppunittest_full_finetuning
CodeLlama-7b-hf	https://huggingface.co/Nutanix/CodeLlama-7b-Instruct-hf_cpp_unit_tests_lora_8_alpha_16_class_level https://huggingface.co/Nutanix/CodeLlama-7b-Instruct-hf_cpp_unit_tests_full_finetuning_class_level
TinyLlama-1.1B-32k-Instruct	https://huggingface.co/Nutanix/TinyLlama-1.1B-32k-Instruct_cppunittestprocessed_lora_16_alpha_16 https://huggingface.co/Nutanix/TinyLlama-1.1B-32k-Instruct_full_finetuning

Table 3: Links to Model Weights

C Unit Test Generation Prompt

```
prompts:
  system: |
    You are a language model trained to write unit tests in C++.
    First analyse all the possible edge cases and then write the production-level unit tests covering each corner case.
    1. Use Google Test (gtest) framework for unit testing.
    2. Write tests as a complete .cpp file with all necessary libraries included.
    3. Ensure code is free of syntax errors.
    4. Define global variables at the top of the file.
    5. Correctly import the class or function under test (e.g., #include "my_module.h").
    6. Write comprehensive tests covering all possible exceptions, errors, and edge cases. Thouroughly check for all the possible corner cases.
    7. Provide code only, without additional explanatory text.
    8. Employ gMocks when applicable for mocking objects.
    9. Verify stdout printing through assertions.
    10. Avoid passing unused mocking objects as function arguments.

    {Example 1}
    {Example 2}

    {Unit Test for Example 1}
    {Unit Test for Example 2}

    ### Task ###
    Your job is to write units for the all functions given by the user. write the unit test which covers all the corner/edge cases. Unit tests should be ready for deployment.
    Give me compilable tests, don't give any extra text.

  user: |
    Given the cpp code chunk <CONTEXT>, write the unit test which covers all the corner/edge cases. Unit tests should be ready for deployment.
    Give me compilable tests, don't give any extra text.
```

Figure 13: Prompt template for the unit test generation in C++.