Universiteit Leiden

Can Repeated Sampling Help Smaller Language Models Outperform Larger Models under a Generation-Time Budget?

Jingbo Li (s3942376)

Supervisors:
Rob van Nieuwpoort & Elisa Chiarotto

Bachelor Thesis– Data Science and Artificial Intelligence

Leiden Institute of Advanced Computer Science (LIACS)
liacs.leidenuniv.nl

July 17, 2026

Abstract

This thesis studies whether the smaller code-oriented language model can compete with the single generation of a larger same-family model through repeated sampling under a calibrated generation-only runtime budget. This thesis advances the existing fixed-budget code generation research to CrossCodeEval's project-level cross-file code completion setting, and compares Qwen2.5-Coder-7B with repeated sampling and Qwen2.5-Coder-32B single generation on Python and C#. The experiment uses character n-gram cosine similarity as the primary quality metric, and uses generation-only runtime as the primary cost metric, with statistical comparison based on 10 random sample groups.

The results show that 7B repeated sampling provides strong evidence on Python and can exceed the 32B baseline at a lower runtime. C# results are more cautious, but still show comparable performance and positive average tendency. Further best-cosine upper bound analysis shows that there is quality space in the candidate set that is not fully utilized by logP-based selection. Therefore, this thesis provides cautious but positive empirical evidence for the smaller code model with repeated sampling, and points out that candidate selection is the key bottleneck for whether this direction can stably compete with the larger baseline.

Contents

1 Introduction **1**

2 Background **2**
- 2.1 Code Completion and Cross-File Context . . . 2
- 2.2 Generation Budget and Repeated Sampling . . . 3
- 2.3 Candidate Selection and Upper-Bound Evaluation . . . 3
- 2.4 Candidate Ranking and Selection . . . 4

3 Related Work **4**
- 3.1 Code Generation under Fixed Compute Budgets . . . 4
- 3.2 Cross-File Code Completion Benchmarks . . . 5
- 3.3 Candidate Selection and Reranking for Generated Code . . . 5
- 3.4 Evaluation of Code Completion . . . 7

4 Methodology **8**
- 4.1 Research Design . . . 8
- 4.2 Dataset and Sampling Protocol . . . 9
- 4.3 Models and Generation Setup . . . 10
- 4.4 Wall-Time Budget Calibration . . . 11
- 4.5 Candidate Selection . . . 11
- 4.6 Evaluation Metrics . . . 12
- 4.7 Statistical Analysis . . . 12
- 4.8 Sanity Validation . . . 12

5 Results **13**
- 5.1 Overview of Results . . . 13
- 5.2 Python Results . . . 13
- 5.3 C# Results . . . 14
- 5.4 Runtime Comparison . . . 15
- 5.5 Best-Cosine Upper-Bound Analysis . . . 16
- 5.6 Sanity Validation . . . 17

6 Discussion and Future Work **18**
- 6.1 Why Repeated Sampling Can Compete with a Larger Baseline . . . 18
- 6.2 Interpretation of the Language-Specific Difference . . . 19
- 6.3 Candidate Selection as a Bottleneck . . . 20
- 6.4 Limitations . . . 21
- 6.5 Future Work . . . 22

7 Conclusion **24**

References **26**

1 Introduction

Code completion is one of the important applications of code language models. Its goal is to predict missing or subsequent code fragments based on a given code context, so as to help developers complete programming tasks more efficiently. In recent years, large language models have made remarkable progress in code generation and code completion, and many code-oriented models have been able to generate high-quality code in tasks such as function generation, program synthesis, and repository-level completion. [3, 20, 14, 11, 6] At the same time, model scaling-related studies show that increasing model size, training data, and compute can usually improve the overall ability of language models. [12, 10] Therefore, larger models are often regarded as a direct path to improve code completion quality.

However, larger models also bring higher practical costs. Larger models usually require more memory, stronger hardware, and higher inference cost. [8] This is especially important for interactive code completion systems, because the system needs to generate suggestions quickly and frequently during the development process. In other words, raw single-generation capability is not the only factor to consider. In settings with limited runtime and hardware resources, how the generation budget is used will also affect the final system performance.

This study focuses on using a smaller model with repeated sampling. Unlike letting the larger model generate only once, the smaller model can generate multiple candidate completions under a similar or lower runtime budget, and select the final output through a selection method. This comparison is no longer just a comparison of model size, but a comparison of generation strategy. This study is not concerned with whether the single-generation capacity of the smaller model naturally exceeds that of the larger model, but whether, under the calibrated generation budget, the smaller model with repeated sampling can match or outperform larger model single generation on final output quality.

This study is based on existing fixed-budget code generation research. Existing work compares larger model single generation and smaller model multiple generations under a fixed compute or wall-time budget, and found that on execution-based code generation benchmarks such as HumanEval, MBPP, and APPS, smaller models can reach or even exceed larger models in some settings through multiple generations. [8] This shows that model size is not the only factor worth optimizing in fixed-budget settings, and the distribution method of the generation budget is also important. However, such benchmarks are mostly based on more standardized function-level or execution-based code generation, and in many cases, they can rely on unit tests to judge or select the correct output. For project-level code completion that is closer to an IDE-like use case, when unit tests are not available, the system must select a final completion from multiple candidates. Whether the advantage of repeated sampling can still be converted into final output quality still needs to be further verified.

Therefore, this study extends the fixed-budget small-vs-large model comparison to CrossCodeEval. CrossCodeEval is a multilingual cross-file code completion benchmark used to evaluate a model's ability to carry out line completion in the repository-level context. [4] Its data is organized by programming language and includes repository, file path, and cross-file context information. [4] This study uses the rg1_unixcoder setting with retrieved crossfile_context, so compared with isolated function-level generation, it is more suitable for checking whether repeated sampling is still valid in more complex project-level completion settings.

The research question of this thesis is:

Can a smaller language model with repeated sampling match or outperform a larger language model with single generation under the same generation-time budget?

This thesis investigates this question in the setting of cross-file code completion through a controlled same-family comparison between Qwen2.5-Coder-7B and Qwen2.5-Coder-32B. The 32B model is used as the larger single-generation baseline, while the 7B model is used as the smaller repeated-sampling model. The generation-time budget is calibrated by using the 32B single-generation runtime as the reference budget, and the 7B model is allowed to generate multiple candidates only within this budget. Importantly, the 32B model was distributed across four NVIDIA A100 GPUs, whereas the 7B repeated-sampling system used only one NVIDIA A100 GPU. Therefore, the 7B repeated-sampling system used only one-quarter of the allocated GPU count of the 32B baseline. The empirical comparison is conducted on Python and C#. Python is the main experimental language, and C# is used as a second-language robustness check. This thesis compares the quality and generation-only runtime of the 7B logP-selected result with the 32B single-generation baseline. It also reports the best-cosine upper bound to analyze whether the repeated-sampling candidate set contains higher-quality candidates that are not fully captured by the realistic logP-based selection route.

The experimental results show that Qwen2.5-Coder-7B with repeated sampling can form effective competition with Qwen2.5-Coder-32B single generation in the tested setting. Python results provide strong evidence, indicating that the smaller model with repeated sampling can exceed the larger same-family baseline under the calibrated runtime budget. C# results are more cautious, but still show comparable performance and positive average tendency. Further analysis shows that there is an upper-bound potential higher than the logP-selected result in the candidate set, indicating that candidate selection is an important bottleneck in the repeated sampling pipeline.

There are three main contributions to this study. First, this study promotes fixed-budget small-vs-large model comparison from simpler code generation benchmarks to CrossCodeEval's project-level cross-file code completion setting. Second, this study compares Qwen2.5-Coder-7B repeated sampling and Qwen2.5-Coder-32B single generation under calibrated generation-only runtime budget, so as to analyze the trade-off of model size and generation strategy. Third, this study distinguishes between realistic logP-selected result and best-cosine upper bound, and shows that the repeated sampling candidate set has underutilized quality space. It also points out that candidate selection is the key factor in whether smaller model repeated sampling can compete stably with the larger baseline.

The code, sampled benchmark subsets, aggregated results, and reproducibility documentation are publicly available at https://github.com/jingboli8/fixed-budget-repeated-sampling.

2 Background

2.1 Code Completion and Cross-File Context

The goal of code completion is to predict missing code fragments or subsequent code fragments based on a given code context. In the line completion setting used in this study, a task usually consists of the left context in the current file, the ground truth that needs to be predicted, and the right context after the ground truth. The model mainly relies on the context provided in the

prompt when generating the completion, while the evaluation compares the similarity between the completion generated by the model and the ground truth. Unlike complete function generation, line completion pays more attention to the completion of local code fragments, but it may still depend on longer context and the relationship between different code elements within the project. Cross-file code completion further expands the context range in this task. In actual software projects, the code in the current file often relies on classes, functions, types, configurations, or usage patterns defined by other files in the same repository. Therefore, using only the left context of the current file may not provide complete information. Cross-file context refers to relevant code snippets from other files in the same project, which can provide additional repository-level information for the current completion task.

The CrossCodeEval benchmark used in this study is built around this cross-file completion setting. [4] The benchmark organizes data according to programming language and records task id, repository with commit hash, file path, and cross-file context information in metadata. [4] The `rg1_unixcoder` setting used in this study contains retrieved cross-file context. Therefore, the input in this study's experiment is not an isolated code snippet, but a line completion task with retrieved repository context. This setting provides a basis for this study to evaluate the effect of repeated sampling in project-level code completion.

2.2 Generation Budget and Repeated Sampling

When comparing different model scales, direct comparison of single-generation output does not fully reflect the resource trade-off in actual use. [8] Larger models usually have a higher single-generation cost, while smaller models may be able to generate multiple outputs under the same or lower runtime budget. Therefore, in a fixed-budget or calibrated-budget setting, the focus of comparison is not only on the scale of the model parameters themselves, but also on how the same generation budget is used.

This thesis uses generation-only runtime to measure the time required for the model to complete the generation stage. This metric only focuses on the generation process itself, excluding model loading, preprocessing, postprocessing, or serving overhead. The purpose of this is to make the comparison more focused on the generation strategy itself. For the 32B single-generation baseline, the model only generates one completion for each task. For the 7B repeated-sampling system, the model generates multiple candidate completions under the calibrated runtime budget.

The core function of repeated sampling is to expand the candidate set. It does not mean that the single-generation ability of a smaller model must be stronger than that of a larger model, but that a smaller model can generate multiple possible completions on the same task. The final system performance depends on two factors. First, whether there is a high-quality candidate in the candidate set. Second, whether the selection method can select the appropriate final output from the candidate set. Therefore, repeated sampling converts part of the model-size comparison into a system-level comparison composed of generation strategy and selection strategy.

2.3 Candidate Selection and Upper-Bound Evaluation

After repeated sampling, the system must select a final completion from multiple candidates. This step is called candidate selection. The main result of this study uses the logP-selected result, that is, the final output is selected according to the log probability of the candidate completion given

by the model. This route does not rely on ground truth, so it can be understood as a realistic inference-time method. It is suitable for simulating how the system selects an output from multiple generated candidates without unit tests or reference answers.
However, logP-based selection does not guarantee that the selected candidate must be the closest to the ground truth. A candidate having a high probability under the model's own distribution does not necessarily mean that it is the most consistent result for the target completion in a specific repository context. Therefore, reporting the logP-selected result alone may not show all the potential in the repeated sampling candidate set. In order to analyze this point, this study also reports the best-cosine upper bound.
The best-cosine upper bound is an analysis-only result. It uses ground truth to select the candidate with the highest cosine similarity from the candidate set, so it cannot be used as a realistic inference-time method. Its function is not to show the performance of a deployable system, but to estimate the theoretically achievable best observed quality in the candidate set. By comparing the logP-selected result and the best-cosine upper bound, this study can distinguish between two problems. The first is whether repeated sampling generates higher-quality candidates. The second is whether the current selection route can stably identify these candidates. This distinction is important for understanding the results of this study, because the bottleneck of repeated sampling may not only be in generation itself, but also in candidate selection.

2.4 Candidate Ranking and Selection

The effect of repeated sampling depends not only on whether the model can generate diverse candidate completions, but also on whether the system can select the appropriate final output from these candidates. If a benchmark provides unit tests or execution feedback, candidate selection can use these external signals to identify candidates that are more likely to be correct. [8] However, in project-level cross-file code completion, each local completion usually does not have a unit test or execution feedback that can be used directly. Therefore, the system needs a selection rule that does not rely on ground truth or direct execution feedback.
In this thesis, logP-based selection is used as a realistic selection route, because it selects candidates according to the model's own probability estimate and does not require reference completion or unit tests. At the same time, this thesis reports the best-cosine upper bound as an analysis-only result. This upper bound uses ground truth to identify the candidate closest to the reference under the cosine-similarity metric, so it cannot be used for actual deployment. The comparison between the logP-selected result and the best-cosine upper bound helps distinguish candidate generation potential from candidate selection effectiveness. More detailed work on candidate selection, verifier-based selection, and reranking will be discussed in Section 3.

3 Related Work

3.1 Code Generation under Fixed Compute Budgets

Existing code generation studies usually regard larger models as an important path to improve performance. With the increase of model size, training data, and compute, language models can often gain stronger generation ability, so larger code models are also often used as strong

baselines. [12, 10, 20, 11] However, recent fixed-budget code generation research has put forward a different comparative perspective, that is, under the same compute or wall-time budget, whether multiple generations from smaller models can be more effective than a single generation from larger models. [8]

The closest work to this thesis is the study of budget reallocation. This work compares larger model single generation and smaller model multiple generations under fixed compute or wall-time budgets, and found that on execution-based code generation benchmarks such as HumanEval, MBPP, and APPS, smaller models can reach or even exceed larger models in some settings through multiple generations. [8] This discovery shows that under the constraints of fixed resources, model size is not the only optimization dimension, and the allocation of generation budget will also affect the final performance.

This thesis is directly related to this route, but the setting is different. Fixed-budget research is mainly carried out on more standardized code generation benchmarks, and many tasks can use unit tests to determine whether the candidate results are correct. This thesis pushes this problem to CrossCodeEval's project-level cross-file code completion setting, and focuses on whether logP-based selection can convert the candidate potential of repeated sampling into final output quality when no unit tests can be used for candidate verification. Therefore, this thesis does not simply repeat the existing fixed-budget comparison, but tests whether this trade-off is still valid in a more complex code completion setting.

3.2 Cross-File Code Completion Benchmarks

The design of code completion benchmarks will directly affect the evaluation method of model capability. [17] Early or more common code generation benchmarks often focus on function-level generation, program synthesis, or unit-test based evaluation, such as HumanEval, MBPP, and APPS. [3, 1, 9] These benchmarks are important for evaluating code generation ability, but they usually do not completely cover the completion scenario in the repository-level context.

Cross-file code completion benchmarks try to evaluate the ability of the model to use project-level context. CrossCodeEval is a multilingual cross-file code completion benchmark, covering Python, C#, Java, and TypeScript, and provides repository-level metadata and cross-file context information. [4] This setting is different from isolated function generation, because the model may need to use the definitions, types, imports, or usage patterns of other files in the same repository to complete the line completion in the current file.

This study chooses CrossCodeEval to retest fixed-budget small-vs-large model comparison in a setting closer to project-level completion. Compared with judging whether smaller model repeated sampling is effective only on function-level benchmarks, CrossCodeEval provides more complex contextual conditions, and is also more suitable for analyzing whether repeated sampling still has competitive potential in repository-level code completion. The final experiment of this study focuses on Python and C#, and takes Java and TypeScript as the natural extension direction of future work.

3.3 Candidate Selection and Reranking for Generated Code

Multiple-sample code generation naturally introduces the problem of candidate selection. When the model generates multiple candidate programs or completions for the same task, the final performance

of the system no longer depends only on whether the model can generate a high-quality candidate, but also on whether the system can select the appropriate final output from the candidate set. This is related to the common use of pass@k in code generation evaluation. Pass@k usually measures whether at least one sample can pass the evaluation after multiple samples are generated. [3, 13] However, pass@k itself is more of an evaluation setting than a deployment setting. In real use scenarios, the system usually cannot give all candidates directly to the user, but needs to return one or a few candidate results. Therefore, candidate ranking and candidate selection have become problems that cannot be ignored in multiple-sample generation systems.

Studies have proposed a variety of ways to select or rank the generated code candidates. In execution-based code generation benchmarks, candidate selection can rely on unit tests or execution feedback. That is to say, the system can generate multiple candidates, and then filter or rank the candidates according to test results, execution feedback, or other correctness signals [15]. The advantage of this method is that the execution results provide an external signal that is closer to functional correctness than the model's own likelihood. [3, 18] Therefore, in benchmarks with unit tests, the potential of multiple generations is easier to utilize, because the system can use the test results to identify correct or more reliable candidates.

Another type of research tries to reduce the dependence on manually written tests and improve candidate selection by generating tests, learning a verifier, or using reranking methods. For example, some methods generate additional tests to check candidate programs and select the final output based on execution agreement between candidates or whether candidates pass generated tests. [2] Other research trains verifiers to judge the reliability of generated programs and combines verifier scores with model generation probabilities for reranking. [18] Such research shows that candidate selection is not a simple post-processing step, but a core factor that can significantly affect the final performance of the multi-generation system.

However, these methods also reveal the differences between them and the setting of this study. Many candidate selection or reranking methods rely on execution, unit tests, generated tests, verifiers, or other external signals directly related to functional correctness. But in project-level cross-file code completion, these signals are usually not easy to obtain. A completion may be just a line or a local fragment in the repository context, and there may not be a unit test that can directly verify whether the completion is correct. [4, 8] Therefore, in the absence of ground truth, execution feedback, and a reliable external verifier, how the system can select the final completion from multiple candidates becomes a more difficult problem.

This thesis is related to the above-mentioned candidate selection and reranking research, but it studies a more restricted selection setting. The main result of this thesis uses logP-based selection, because it does not use ground truth and does not rely on unit tests, so it can be used as a realistic inference-time selection route for repeated sampling in cross-file code completion. At the same time, this thesis reports the best-cosine upper bound, which is used to estimate how much unused quality space remains in the candidate set if the candidate could be ideally selected under the cosine-similarity metric. Therefore, the gap between the logP-selected result and the best-cosine upper bound can be linked to a broader candidate selection problem. That is to say, repeated sampling may indeed generate useful candidate-level variation, but whether this variation can be converted into the final system performance depends on whether the selection strategy can stably identify high-quality candidates.

3.4 Evaluation of Code Completion

The evaluation method of code generation and code completion directly affects the interpretation of the model's ability. For function-level code generation or program synthesis tasks, common evaluation methods usually rely on unit tests or execution-based correctness. If the program generated by the model can pass the test, it can be regarded as successful in the task. The advantage of this evaluation method is that it is closer to functional correctness, because it not only compares text similarity, but also checks whether the generated code can produce correct behavior under a given test. At the same time, the reliability of execution-based evaluation still depends on the coverage and strength of the available tests. [16] Therefore, HumanEval, MBPP, APPS and other benchmarks are often used to evaluate the performance of models in executable code generation tasks. [3, 1, 9]

However, execution-based evaluation cannot naturally cover all code completion settings. Especially in repository-level or cross-file code completion, the target may be just a line in the current file, a local fragment, or a short completion that depends on the context of other files. Such tasks may not have a unit test for each completion, nor may they be able to judge whether the candidate is correct through direct execution. Therefore, in settings closer to IDE-like completion, evaluation often relies on non-execution metrics, such as token-level match, edit similarity, exact match, or similarity metrics based on n-grams. [4, 19] These metrics are not equal to functional correctness, but can provide a consistent comparison basis for different models and different generation strategies.

In addition to token-level similarity, edit similarity, and n-gram similarity, semantic-based or structure-based metrics can also be considered in code evaluation. [19, 5, 7] For example, semantic-based metrics can compare the semantic proximity between generated completions and reference completions with the help of an embedding model, while structure-based metrics can use syntax trees, parsers, or other code structure information to compare generated results. This kind of metric may theoretically be closer to code semantics or grammatical structure than pure text overlap, but it also introduces new experimental dependencies and interpretation risks. First of all, structure-based metrics often rely on parser and language version support, and may be affected by parser compatibility, grammar version, and partial code fragments in multilingual cross-file completion settings. Secondly, semantic-based metrics depend on additional embedding models or scoring models, so the evaluation results may be affected by the bias of the auxiliary model itself. For this thesis, the introduction of new semantic or structure-based metrics would change the evaluation basis of the experiment and make the results more difficult to compare with the current fixed-budget comparison. Therefore, this thesis does not use these metrics as formal evaluation metrics, but uses character n-gram cosine similarity as a simple, reproducible prototype quality metric that is applicable to both Python and C#. This choice sacrifices the ability to directly explain functional correctness, so this thesis is cautious about its interpretation in Methodology and Limitations.

The evaluation of cross-file code completion is also affected by the repository context. Whether a completion is appropriate depends not only on whether the local text is similar, but also on whether it correctly uses imports, types, functions, naming conventions, or usage patterns in the project. Therefore, pure text similarity metrics have limitations. They may underestimate completions that are semantically equivalent but textually different, and may also overestimate completions with similar text but incorrect functionality. On the other hand, in the absence of unit tests or executable correctness signals, this kind of metric can still be used as a reproducible proxy to

compare the relative performance of different systems on the same task set.
The evaluation setting in this study is related to the above work, but it also has its own limitations. This study uses CrossCodeEval's project-level cross-file code completion task, and focuses on the comparison between 7B repeated sampling and 32B single generation under a calibrated generation-only runtime budget. Due to the parser compatibility issue of the official metric in the current experimental environment, this study does not take official syntax-aware evaluation as the formal metric, but uses character n-gram cosine similarity as the primary quality metric. This choice allows the results of different models and different languages to be compared under the same prototype metric, but it also means that the results of this study should be interpreted as empirical evidence based on similarity, not a complete functional correctness evaluation. Therefore, this thesis reports cosine-based quality comparison in Results, and further discusses the limitations of this metric choice on the interpretation of conclusions in Limitations.

4 Methodology

4.1 Research Design

In order to answer the research question of this thesis, this experiment adopts a controlled empirical comparison. In this comparison, the smaller model generates multiple candidate completions for each task through repeated sampling, while the larger same-family baseline generates one completion for each task under a roughly matched generation-time budget. The focus of this design is not simply to compare model size, but to examine the relationship between output quality and generation cost under a limited generation budget.
Specifically, `Qwen/Qwen2.5-Coder-32B` serves as the larger same-family baseline. It generates only one completion for each task. `Qwen/Qwen2.5-Coder-7B` serves as the smaller repeated-sampling model and generates multiple candidate completions within the calibrated budget. Choosing two models from the same base family can reduce the additional impact caused by differences between unrelated model families, such as tokenizer, training method, or model design. This makes the comparison more focused on model scale, sampling strategy, and generation-time budget.
The key design choice of this experiment is to use generation-only wall-clock time as the cost constraint. Specifically, the single-generation runtime of the 32B model defines the reference budget, and the 7B model can only perform repeated sampling within this budget. Since the sample count must be an integer, the time budgets of the two models cannot be matched exactly second by second. Therefore, this experiment defines k as the maximum repeated sample count that does not exceed the 32B single-generation runtime. This design is conservative for the 7B model, because the 7B repeated-sampling runtime remains lower than the 32B baseline runtime.
For 7B repeated sampling, each task produces multiple candidate completions. The main comparison uses a realistic inference-time selection strategy. For each task, this thesis selects the candidate with the highest mean log probability and calculates the cosine similarity between the selected candidate and the ground truth as the 7B primary quality score. In addition, this study also reports the best-cosine upper-bound analysis, which selects the candidate with the highest cosine similarity to the ground truth from the candidate set. Since the best-cosine upper bound uses the ground truth, it does not represent a selection strategy available in real-world deployment. It is only used to estimate how much potential improvement remains in the candidate set generated by repeated

sampling but is not used by the current selection route. Subsequent subsections will further explain the dataset, model setup, calibration procedure, evaluation metrics, statistical analysis, and sanity validation.

To make the experimental design clearer, Table 1 summarizes the key settings used in this thesis. The table provides an overview of the benchmark, languages, models, runtime calibration, sampling protocol, candidate selection, metrics, and additional analyses. The following subsections explain these components in more detail.

Table 1: Summary of the experimental setup used in this thesis.

Component	**Setting**
Benchmark and task	CrossCodeEval project-level cross-file code completion
Languages	Python as the primary evaluation language, and C# as the second-language robustness check
Models	Qwen2.5-Coder-7B repeated sampling vs. Qwen2.5-Coder-32B single generation
Runtime calibration	32B single-generation runtime is used as the reference budget. Python uses $k = 7$, and C# uses $k = 6$ because $k = 7$ slightly exceeds the C# runtime budget.
Sampling protocol	10 random sample groups per language, with 150 tasks in each group
Main comparison	7B logP-selected mean cosine vs. 32B single-generation mean cosine
Quality metric	character n-gram cosine similarity
Cost metric	generation-only wall-clock runtime
Additional analysis	best-cosine upper bound, paired statistical comparison, runtime ratio, and sanity validation

4.2 Dataset and Sampling Protocol

This experiment uses CrossCodeEval as the code completion evaluation dataset. The task setting of this benchmark focuses on project-level cross-file code completion. In this setting, the model needs to generate a completion based on the current file context and the provided cross-file context. Since the local release does not provide a clear train, validation, or test split, this experiment uses the benchmark files of the corresponding language in the release as formal evaluation files. Small-scale preliminary subsets are only used for pipeline testing and time calibration, and are not included in the final reported results.

The local CrossCodeEval release contains task files for Python, Java, TypeScript, and C#. The corresponding files contain 2,665 Python tasks, 2,139 Java tasks, 3,356 TypeScript tasks, and 1,768 C# tasks. Python is used as the primary evaluation language because it is a common language in code generation evaluation and was also the starting point of the experimental pipeline in this experiment. In order to avoid limiting the analysis to a single programming language, this experiment adds C# as a second-language robustness check. C# is not chosen because it is more representative than Java or TypeScript, but as an additional non-Python setting to test whether the observed repeated-sampling pattern also appears outside Python. The final scope of this experiment is limited to two languages to ensure that each included language can complete the

full experimental procedure, including ten random sample groups, runtime calibration, candidate selection analysis, and statistical testing.

After identifying Python and C# as the evaluation languages, this experiment constructs 10 random sample groups for each language. Each group contains 150 tasks and is generated with fixed random seeds to ensure reproducibility. The specific seeds are 42, 123, 999, 1001, 1002, 1003, 1004, 1005, 1006, and 1007. Without-replacement sampling is used within each group to ensure that the same task does not appear repeatedly in the same group. Different seed groups are regarded as independent random samples, so overlap is allowed between different groups. Sampled task indices are sorted before writing the sample files to retain the task order in the original benchmark files.

After sampling is completed, this study conducts a basic integrity check on all sample files. Each sample group in Python and C# contains 150 rows and 150 unique task IDs, with no missing task IDs and no within-group duplicate task IDs. Therefore, each seed group provides a comparable evaluation subset for subsequent analysis. Subsequent statistical analyses use seed-level group averages, which enables this analysis to report variance, confidence intervals, and paired statistical tests between different random samples, instead of relying only on the overall average of a single random subset.

4.3 Models and Generation Setup

This experiment compares two checkpoints from the Qwen2.5-Coder family: `Qwen/Qwen2.5-Coder-7B` and `Qwen/Qwen2.5-Coder-32B`. The 32B checkpoint is used as the larger same-family baseline, while the 7B checkpoint is used for repeated sampling. Models from the same family are used because the main comparison should not mix the repeated-sampling effect with differences between unrelated model families. The comparison is still not a perfect parameter-size-only comparison, because the two checkpoints may differ in training dynamics and inference behaviour. However, keeping the family fixed makes the experiment more directly about model scale, sampling strategy, and generation-only runtime budget.

All formal runs use the same core generation configuration. The model type is set to `codelm_cfc`, which is the CrossCodeEval setting that includes cross-file context. The generation length is 10, the maximum sequence length is 2048, the cross-file context length is 512, and the batch size is 1. These values were not tuned separately for 7B and 32B. They were kept fixed across the two models so that the later comparison would not be affected by different prompt lengths, batch sizes, or generation lengths.

For the 7B runs, the timed candidate-saving script `eval_save_candidates_timed.py` was used. This script was needed because the 7B setting does not only produce one final completion: it stores the full candidate list for each task and records the mean log probability of each candidate. These saved candidates are later used to compute both the logP-selected result and the best-cosine upper bound. For the 32B baseline, the timed multi-GPU script `eval_32b_multigpu_timed.py` was used, which saves one prediction per task in `prediction.jsonl`. The 32B model was placed on four A100 GPUs with `device_map="auto"`, while the 7B runs used one A100 GPU. In both cases, the timed scripts record generation-only runtime. The final statistics are computed from the saved prediction files, candidate summaries, and runtime logs, rather than from manually copied terminal outputs.

4.4 Wall-Time Budget Calibration

This experiment uses generation-only wall-clock time as the primary cost constraint, rather than total runtime or token count. The reason is that this experiment focuses on the cost of generating the completion itself, rather than additional overhead such as model loading, environment initialization, or file I/O. Token count can be used as a descriptive indicator, but it cannot directly replace actual wall-clock runtime, because the same number of tokens under different model scales and hardware settings may correspond to different generation times.

For each language, the single-generation runtime of the 32B model is used as the reference budget. The repeated-sampling value k of the 7B model is defined as the maximum integer sample count that does not exceed this budget. Since k is a discrete value, the generation-only times of the two models cannot be exactly matched. If an additional round of sampling would exceed the 32B runtime, the current k value is kept.

In the final experiment, k was calibrated separately for each language. The goal was to choose the largest integer k for the 7B model that did not exceed the measured single-generation runtime of the 32B baseline. Python used $k = 7$, because the calibration run showed that the 7B repeated-sampling runtime with $k = 7$ remained below the 32B single-generation runtime, while the remaining time was insufficient for an eighth full sampling round.

For C#, the calibration result was borderline. The 7B runtime with $k = 7$ was 1.511960 seconds per task, compared with 1.510532 seconds per task for the 32B single-generation baseline. This difference is very small, but to keep the budget rule conservative and avoid exceeding the measured baseline runtime, this thesis adopted $k = 6$ for C#. This choice makes the C# comparison slightly more conservative for the 7B repeated-sampling system. Therefore, the C# results should be interpreted as performance under a strictly non-exceeding runtime budget, rather than as the most favourable repeated-sampling setting.

4.5 Candidate Selection

In 7B repeated sampling, each task produces multiple candidate completions. Therefore, a candidate selection rule is needed to choose a 7B output for the main comparison. In contrast, the 32B baseline only generates one completion per task, so the 32B output does not require a candidate selection step.

The main comparison uses mean log probability for candidate selection. As a selection criterion, it selects the candidate with the highest average likelihood assigned by the model, without using the ground truth. For each task, the candidate with the highest mean log probability is selected as the final 7B output. This process does not use the ground truth, so it can be regarded as a realistic inference-time selection method. After selecting the candidate, this thesis calculates the cosine similarity between it and the ground truth and uses it as the 7B primary quality score.

In addition, this analysis also calculates and records the best-cosine upper bound. The upper-bound result selects the candidate with the highest cosine similarity to the ground truth from the candidate set. Since this process uses the ground truth, it is not a deployable inference-time method, but an offline upper-bound analysis. It is used to evaluate the candidate set potential and the gap between logP selection and the best observed candidate under the cosine-similarity metric.

4.6 Evaluation Metrics

This experiment uses character n-gram cosine similarity as the primary quality metric. This metric represents the generated completion and the ground-truth completion as character n-gram vectors, and calculates the cosine similarity between them. The higher the score, the closer the generated result is to the reference completion at the character n-gram level. Since this metric measures character-level similarity, it should not be interpreted as a direct measure of code functional correctness.

The primary cost metric of this experiment is generation-only wall-clock time. It only records the actual time required for the model to generate the completion, excluding overhead such as model loading, environment initialization, or file I/O. This metric is used for the aforementioned time budget calibration and subsequent runtime comparison. Token count is only used as descriptive information, because the same number of tokens may correspond to different generation times under different model sizes and hardware conditions. CrossCodeEval's official metric depends on tree-sitter parsing. In the current experimental environment, this parser-based evaluation could not be applied because of a version mismatch between the installed tree-sitter Python binding and the compiled language grammar. The evaluation failed at `parser.set_language(language)` with the error

```
ValueError: Incompatible Language version 15. Must be between 13 and 14.
```

Therefore, this thesis does not use the official parser-based metric as a formal evaluation metric. Instead, this experiment uses character n-gram cosine similarity as the prototype quality metric, and discusses the impact of this choice in the limitations section.

4.7 Statistical Analysis

The statistical analysis of this study uses seed-level group averages. For each language, each seed group contains 150 tasks. For each group, this thesis calculates the mean cosine of the 32B single-generation model, the mean cosine of the 7B logP-selected result, the mean cosine of the 7B best-cosine upper bound, and the generation-only runtime of the two systems. This generates 10 group-level observations for each language. Using group-level averages can avoid mixing all task-level outputs directly into an overall average, so as to more clearly reflect the variability between different random samples.

The main quality difference is defined as 7B logP-selected mean cosine minus 32B mean cosine in each seed group. Since the two models are evaluated on the same seed group, these group-level differences constitute paired comparisons. This analysis reports the mean difference, standard deviation, standard error, 95% confidence interval, and paired t-test p-value based on 10 paired differences for each language. In addition, this analysis also reports group-level wins/losses/ties, best-cosine upper-bound gain, mean generation-only time, time saving, and 7B/32B time ratio. The wins/losses/ties here refer to group-level average comparison, not individual task-level comparison.

4.8 Sanity Validation

After generating final statistical tables, this analysis conducts additional sanity validation to confirm that the values in the summary table are consistent with the original experimental outputs. The

validation script rereads the output files corresponding to each language-seed group, and checks the 32B average cosine, 7B logP-selected average cosine, 7B best-cosine upper bound average cosine, and generation-only runtime. For cosine scores, validation uses the 32B prediction cosine file and the 7B candidate summary file respectively. For runtime, the output log is used first. If no log is saved in the earlier run, the timing recorded in the corresponding thesis note is used.
Validation covers 20 language-seed rows, corresponding to 10 Python groups and 10 C# groups. All rows passed the check. Only one timing value uses thesis-note fallback, and the rest of the timing values come from output logs. This step cannot replace formal statistical analysis, but it can reduce the risk of transcription errors or file correspondence errors when manually sorting results. Therefore, it is used as a completeness check before the final results summary.

5 Results

5.1 Overview of Results

This section reports the final experimental results on Python and C#. The main comparison object is the logP-selected result of the 32B single-generation baseline and the 7B repeated-sampling system. The best-cosine upper bound is used to measure the upper-bound performance that can be achieved when selecting based on ground truth in the candidate set.
Table 2 provides a compact summary of the main quality comparison and the 7B/32B generation-only runtime ratio. Table 3 reports the statistical comparison across random sample groups, and Figure 1 visualises the corresponding seed-level paired differences. Table 4 reports the detailed generation-only runtime comparison. The following sections analyse the Python and C# quality results, runtime differences, best-cosine upper-bound results, and sanity validation in more detail.

Table 2: Main quality and runtime-ratio summary across 10 random sample groups per language. The mean difference is computed as the 7B logP-selected mean cosine minus the 32B single-generation mean cosine.

Language	32B cosine	7B logP cosine	Mean diff.	7B/32B time ratio
Python	0.313586	0.357906	+0.044321	0.940059
C#	0.487399	0.495104	+0.007704	0.798098

5.2 Python Results

In the Python setting, the 7B logP-selected result is significantly higher than the 32B single-generation baseline. As shown in Table 2, the mean cosine of 32B is 0.313586, while the mean cosine of 7B logP-selected is 0.357906, and the corresponding mean difference is +0.044321. This difference remains stable in 10 random sample groups: 7B is higher than 32B in all 10 groups, and there are no losses or ties.
The statistical test further supports this result. The 95% confidence interval on Python is [0.033649, 0.054992], and the whole interval is higher than 0, indicating that the observed improvement is not caused only by a few sample groups pulling up the average. The p-value of the paired t-test is 0.000006, which also shows that the difference between 7B logP-selected and 32B single generation is statistically significant on the seed-level group average. Therefore, in the Python setting, 7B

repeated sampling not only achieves comparable performance under the main selection route, but also shows a stable quality advantage.

5.3 C# Results

In the C# setting, the average performance of the 7B logP-selected result is slightly higher than that of the 32B single-generation baseline, but this result is not as stable as Python. As shown in Table 2, the mean cosine of 32B is 0.487399, while the mean cosine of 7B logP-selected is 0.495104, and the corresponding mean difference is +0.007704. In 10 random sample groups, 7B won 7 groups and lost 3 groups without ties. This shows that 7B repeated sampling is not significantly inferior to the 32B baseline on C#, but its advantage is not stable.

The statistical results also support a more cautious explanation. The 95% confidence interval on C# is [-0.004436, 0.019845], which contains 0, indicating that the observed difference may not be stably separated from zero. The p-value of the paired t-test is 0.184941, so the C# result cannot be interpreted as a statistically significant improvement. The more appropriate conclusion is that in the C# setting, 7B repeated sampling achieves comparable performance and presents a small positive average difference, but the strength of evidence is weaker than Python.

Table 3: Statistical comparison across 10 random sample groups per language. The mean difference is computed as the 7B logP-selected mean cosine minus the 32B single-generation mean cosine. Wins, losses, and ties are based on group-level average comparison.

Language	95% CI	p-value	Wins/Losses/Ties	Interpretation
Python	[0.033649, 0.054992]	0.000006	10/0/0	Strong positive evidence
C#	[-0.004436, 0.019845]	0.184941	7/3/0	Weaker positive evidence

Figure 1 further visualises this contrast at the seed-group level. In Python, all 10 paired differences are above zero, whereas in C#, most paired differences are positive but three groups are below zero, including one near-zero negative difference.

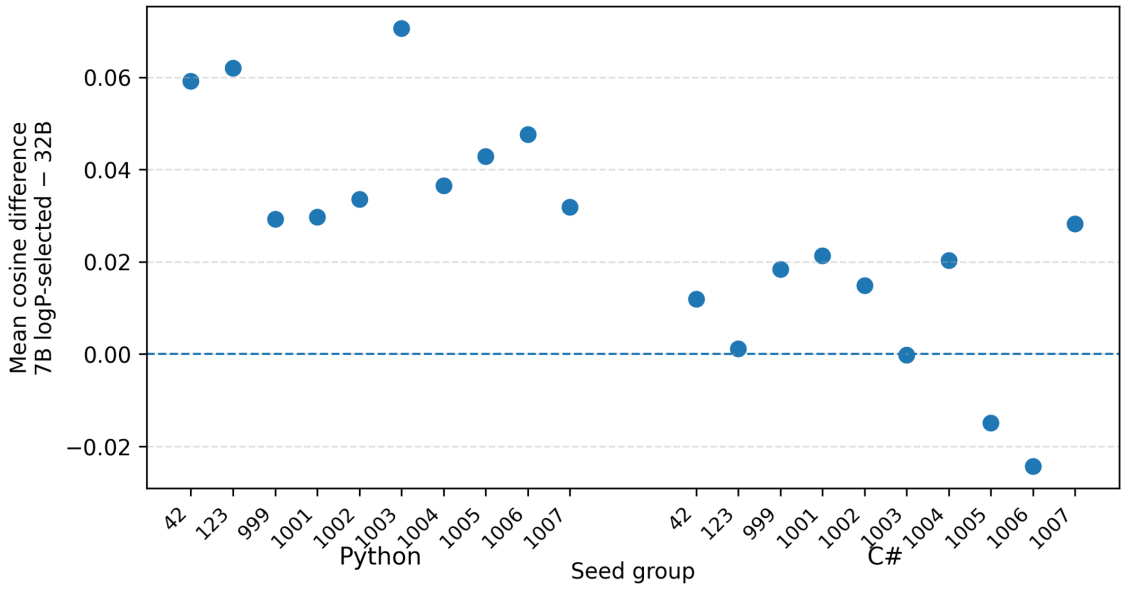

Figure 1: Seed-level paired differences computed as the 7B logP-selected mean cosine minus the 32B single-generation mean cosine across the 10 random sample groups. The dashed horizontal line marks zero difference. Positive values indicate that the 7B logP-selected result is higher than the 32B baseline for that seed group.

5.4 Runtime Comparison

In addition to quality results, runtime comparison is also a core part of this experiment, because this study focuses on whether the repeated sampling of small models under a calibrated generation-only wall-time budget can compete with larger same-family baselines. Table 4 and Figure 2 report the detailed generation-only runtime values. In Python, the average generation-only runtime of 7B repeated sampling is 206.30 seconds, lower than 219.46 seconds of 32B single generation, corresponding to a time ratio of 0.940059. In C#, the average runtime of 7B is 182.32 seconds, lower than 228.46 seconds of 32B, corresponding to a time ratio of 0.798098.

Table 4: Generation-only runtime comparison across 10 random sample groups per language. The time ratio is computed as the 7B repeated-sampling runtime divided by the 32B single-generation runtime.

Language	32B time (s)	7B time (s)	Time saving (s)	Time ratio
Python	219.46	206.30	13.16	0.940059
C#	228.46	182.32	46.14	0.798098

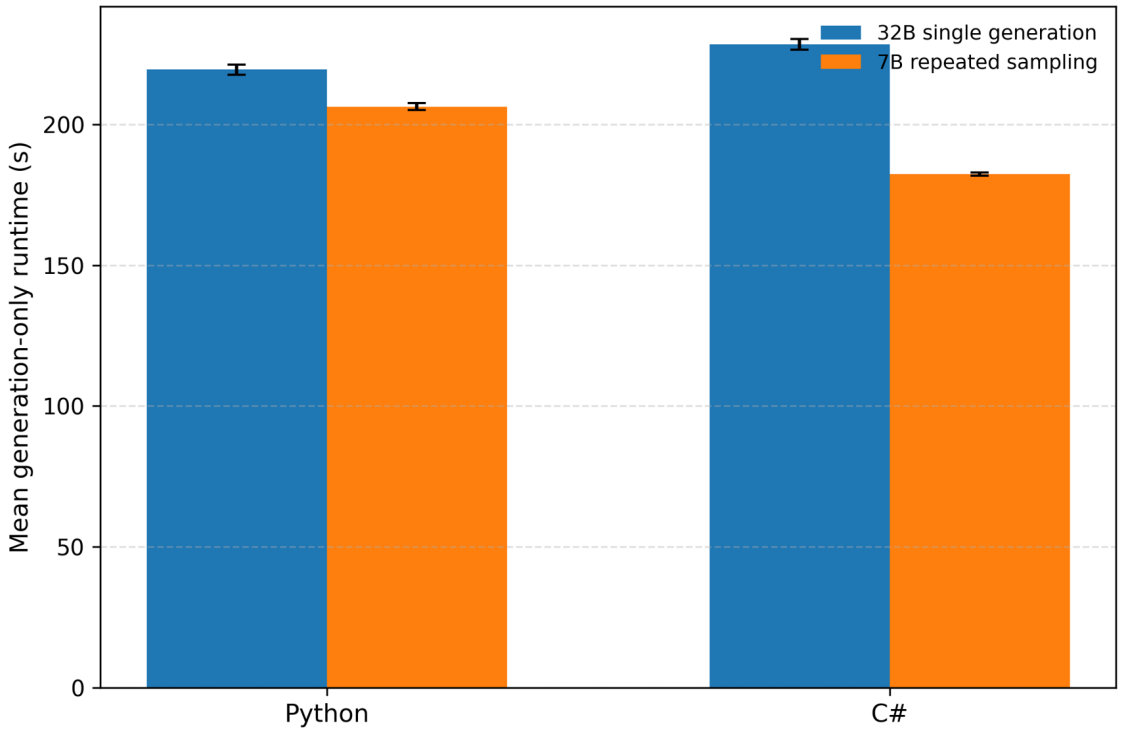

Figure 2: Mean generation-only runtime across 10 random sample groups for the 32B single-generation baseline and the 7B repeated-sampling system. Error bars indicate standard deviation across seed-level group averages.

These results show that the quality results of 7B repeated sampling were not obtained through a generation-time budget exceeding the 32B baseline. On the contrary, in both languages, 7B uses less generation-only runtime. In addition, the 7B experiments used one NVIDIA A100 GPU, whereas the 32B baseline used four NVIDIA A100 GPUs. Therefore, the Python results combine higher completion quality and lower generation-only runtime with only one-quarter of the allocated GPU count used by the 32B baseline. The C# results are more appropriately interpreted as achieving comparable quality with a small positive average difference and lower generation-only runtime, while using only one-quarter of the allocated GPU count of the 32B baseline.

5.5 Best-Cosine Upper-Bound Analysis

In addition to the logP-selected result, this analysis also reports the best-cosine upper bound, which is used to analyse whether the candidate set generated by repeated sampling contains higher-quality candidate results. This upper-bound result selects the candidate with the highest cosine similarity based on ground truth, so it cannot be used as a realistic inference-time method and cannot replace the logP-selected result in the main result. Figure 3 visualises this comparison across Python and C#.

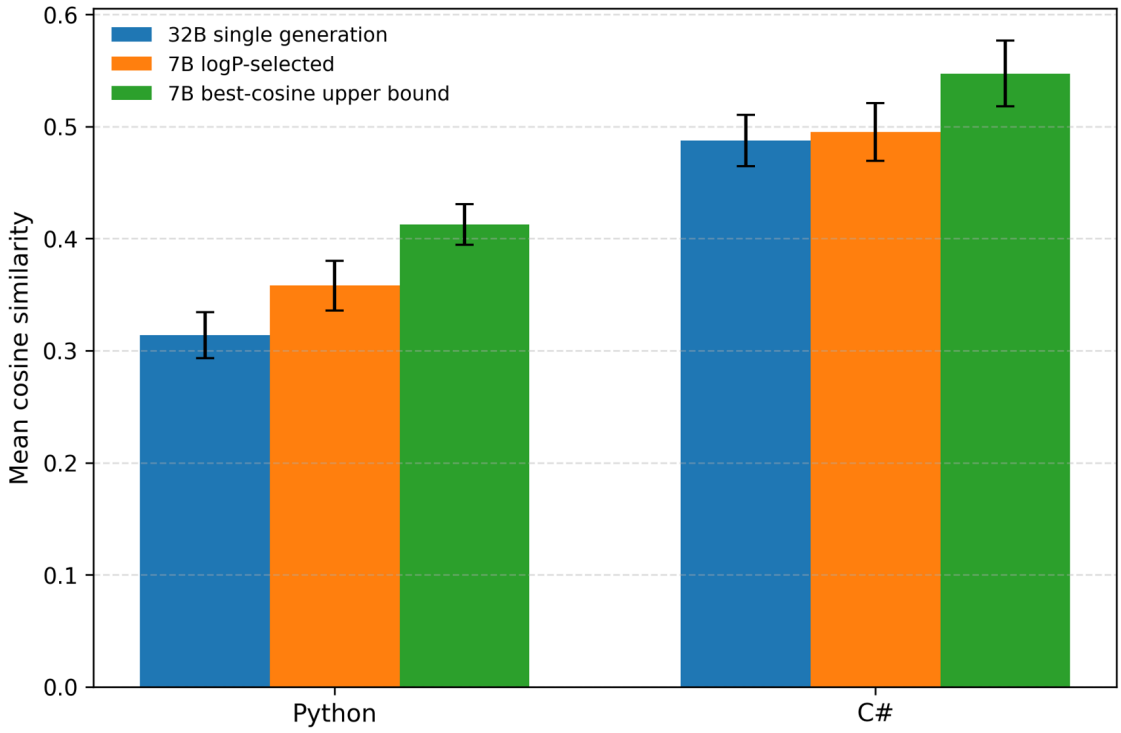

Figure 3: Mean cosine similarity across 10 random sample groups for the 32B single-generation baseline, the 7B logP-selected result, and the 7B best-cosine upper bound. Error bars indicate standard deviation across seed-level group averages.

The results show that the best-cosine upper bound is significantly higher than the logP-selected result in both languages. In Python, the mean cosine of 7B logP-selected is 0.357906, while the best-cosine upper bound is 0.412592, and the mean upper-bound gain is 0.054685. In C#, the mean cosine of 7B logP-selected is 0.495104, while the best-cosine upper bound is 0.547149, and the mean upper-bound gain is 0.052046. The similar gain in the two languages indicates that repeated sampling does produce additional candidate-level potential.
This result shows that the limitations of the current system come not only from the 7B model itself, but also from the candidate selection strategy. logP-based selection can select candidates without using ground truth, so it is suitable as the main experimental result, but it does not fully select the candidate in the candidate set that is closest to the ground truth. Therefore, the main meaning of best-cosine upper-bound analysis is to reveal the selection gap and provide direction for the subsequent improvement of candidate selection, rather than claiming that the current system has reached this upper bound.

5.6 Sanity Validation

In order to check that the final result table is not affected by manual spelling mistakes, wrong files, or errors in the statistical process, the results were rechecked from the actual output files and timing notes. This validation does not replace the previous statistical analysis, but only serves as a supplementary check for the reliability of the results.
This sanity validation checked a total of 20 lines of language-seed results, corresponding to 10

Python groups and 10 C# groups. The inspection content includes the average cosine of 32B, the average cosine of 7B logP-selected, the average cosine of 7B best-cosine upper bound, and the corresponding generation-only runtime. In the end, all 20 lines passed the inspection, and no mismatch was found.
The only thing that needs to be noted is that the 32B timing of Python seed 42 uses thesis-note fallback, because the corresponding run is not saved in this earlier output directory. However, the timing value has been recorded in the saved thesis note, so it can still be checked by the validation script. In general, this sanity validation supports that the final summary table is consistent with the actual experimental output and record files, but it does not constitute new statistical evidence in itself.

6 Discussion and Future Work

This section discusses the meaning of the above results beyond the direct numerical comparison. The focus of the discussion includes why repeated sampling can make the smaller model competitive under the calibrated generation-only runtime budget, why the effects observed in Python and C# are different, how candidate selection limits the current system, and the limitations and future work corresponding to these findings.

6.1 Why Repeated Sampling Can Compete with a Larger Baseline

The research question of this thesis is based on an important premise, that is, the larger model is usually expected to have a stronger single-generation baseline. Existing model scaling-related studies show that with the increase of model size, training data, and compute, language model performance tends to be improved. [12] Therefore, Qwen2.5-Coder-32B single generation is not an arbitrary control group in this study, but a reasonable larger same-family baseline. Because this baseline itself is highly competitive, Qwen2.5-Coder-7B with repeated sampling has clear research significance if it can reach or exceed it.
However, this study does not compare the raw model capabilities of 7B and 32B, but two generation systems formed under the runtime budget. The 32B single-generation baseline depends on the output quality of one larger model generation. The 7B repeated-sampling system generates multiple candidates under the calibrated generation-only runtime budget, and selects the final output through logP-based selection. Therefore, the core comparison of this study is not whether 7B single generation is stronger than 32B single generation, but whether 7B repeated sampling plus candidate selection can compete with 32B single generation in final output quality.
This changes the focus of the result interpretation. The function of repeated sampling is not to directly improve the inherent parameter ability of the 7B model, but to change the way the output space of the smaller model is used. Single generation only observes one completion, while repeated sampling can observe multiple possible completions under the same task. In this way, the system has the opportunity to choose a result closer to the reference from multiple candidates. In other words, repeated sampling transforms part of the model size gap into candidate generation and candidate selection problems. It does not deny the single-generation advantage of the larger model, but provides another way to use the limited generation budget.
The results of this study provide empirical support for this explanation. The main result shows that

the 7B logP-selected result can achieve competitive performance under the calibrated generation-only runtime budget, and does not obtain this result through a runtime budget exceeding the 32B baseline, as shown in Table 2. At the same time, the best-cosine upper bound is higher than the logP-selected result, as shown in Figure 3. This shows that the candidate set generated by repeated sampling does contain additional high-quality candidate results, rather than only producing multiple outputs of similar quality. Therefore, the reason why the 7B system can compete with the larger baseline is not only the number of generations, but also that repeated sampling produces candidate-level variation that can be used by the selection strategy.

This conclusion also shows that the results of this study should be understood as a system-level finding, not a simple model-size ranking. This study does not claim that the single-generation capacity of the 7B model comprehensively exceeds that of the 32B model, nor does it claim that the smaller model is superior to the larger model under all conditions. More precisely, this study shows that in the tested setting, the smaller model can get competitive final outputs through repeated sampling and candidate selection. This discovery supports a more practical research direction, that is, under limited generation-only runtime and lower model size, narrowing the performance gap with larger models by improving the generation strategy.

6.2 Interpretation of the Language-Specific Difference

The previous section interprets the results of this study as a system-level comparison, that is, 7B repeated sampling does not prove that the single-generation capacity of the 7B model comprehensively exceeds that of the 32B model, but compares a different generation strategy. This section further discusses the differences between Python and C# results. The meaning of this difference is not only that the scores on the two languages are different, but also that the benefits of repeated sampling will not be automatically converted from candidate generation to final output quality. It also depends on whether there are high-quality candidates in the candidate set and whether logP-based selection can stably select these candidates.

Python results show that this conversion process is relatively successful in this language setting. The 7B logP-selected result is stably higher than 32B single generation in multiple random sample groups, which shows that the candidate variation from repeated sampling can be transformed into the final quality advantage through logP-based selection. That is to say, the strong result on Python not only shows that 7B generates more candidates, but also shows that the useful variation in these candidates has been more stably used by the current selection route. Therefore, Python provides the clearest support for the research question of this study, indicating that the smaller model with repeated sampling can exceed the larger same-family baseline under the calibrated runtime budget.

The results of C# reveal the boundaries of this mechanism. The average 7B logP-selected result in C# is slightly higher and wins in most random sample groups, but this advantage does not achieve the same statistical stability as Python. Therefore, C# should not be understood as a counterexample of repeated sampling, but as a weaker conversion case. It shows that repeated sampling can generate competitive outputs, but logP-based selection may not always be able to stably convert the potential of the candidate set into a significant final performance gain.

Another factor is that the C# repeated-sampling setting is more conservative in runtime matching. Because repeated sampling can only use an integer number of samples, the calibrated setting may leave unused runtime budget. This effect is especially visible in C#. Since $k = 7$ slightly exceeded

the 32B runtime in the calibration run, $k = 6$ was used in the final C# experiment. As a result, the final C# 7B setup used only 79.8% of the 32B generation-only runtime on average. This makes the C# comparison more conservative than an exact time-matched comparison. Therefore, the C# result should not be interpreted as showing a statistically significant quality advantage for 7B. Rather, it suggests that the 7B repeated-sampling setup can achieve roughly comparable, slightly higher average quality while using substantially less generation-only time.

This can be further seen from the gap between the best-cosine upper bound and the logP-selected result. Figure 3 shows that the best-cosine upper bound in both languages is higher than the logP-selected result. This means that there are indeed candidates that are closer to the reference than the current selected output in the candidate set generated by repeated sampling. In other words, the weaker main result in C# cannot be simply explained by saying that the 7B model cannot generate high-quality candidates at all. A more reasonable explanation is that the potential in the candidate set is not captured equally stably by logP-based selection.

Therefore, the main reflection brought by the language-specific difference is that the effectiveness of repeated sampling depends not only on generating more candidates, but also on the cooperation between candidate generation and candidate selection. In Python, this combination produces a stable advantage. In C#, the candidate set still shows potential space, but the main selection route does not convert this space equally steadily into statistically significant improvement. This result makes the conclusion of this study more bounded: smaller model with repeated sampling is a potential direction, but its actual effect depends on the specific language setting and selection reliability.

This explanation still needs to be cautious. The results of this study can show that the conversion effect of repeated sampling on the two languages is different, and it can also show through the best-cosine upper bound that there is potential in the candidate set that is not fully utilized. However, this study does not specifically separate the influence of programming language properties, task distribution, training data exposure, or metric behavior. Therefore, this study cannot assert the only source of the difference between Python and C#. The more robust conclusion is that Python results show that repeated sampling can successfully enhance the smaller model, and C# results show that this enhancement process may be more unstable in different language settings.

6.3 Candidate Selection as a Bottleneck

The previous section explained that the return of repeated sampling depends on whether the potential in the candidate set can be steadily converted into final output quality. The results of this study further show that candidate selection is the key bottleneck of the current system. The best-cosine upper bound is higher than the logP-selected result in both languages, which indicates that the candidates generated by repeated sampling contain completions that are closer to the reference, but the current main selection route does not fully select these results. In other words, the limitation of the 7B system is not only whether it can generate high-quality candidates, but also whether it can identify these candidates without using ground truth.

This is very important to explain the value of repeated sampling. If repeated sampling only generates multiple outputs of similar quality, the significance of increasing the number of samples for the final output quality will be very limited. However, Figure 3 shows that there is a stable gap between the best-cosine upper bound and the logP-selected result. This means that the candidate set itself contains additional quality headroom. The problem is that this headroom will only be

truly transformed into deployable system performance when the selection strategy can effectively identify high-quality candidates.

The advantage of logP-based selection is that it does not rely on ground truth, so it can be used as a realistic inference-time selection method. In contrast, the best-cosine upper bound uses reference completion for selection, which can only be used for upper-bound analysis and cannot be used as an actual deployment method. However, the results of this study show that the candidate with the highest mean log probability is not always the candidate closest to the reference. This indicates that there is a gap between probability-based preference and task-level similarity. For code completion, a candidate being more probable under the model's own distribution does not necessarily mean that it is more in line with the target completion in the specific repository context.

Therefore, the results of this study shift the subsequent improvement direction of repeated sampling from simply increasing generation count to candidate selection. Adding more candidates may expand the candidate set, but if the selection method cannot reliably distinguish the actual quality of candidates, the benefits of additional sampling may still not be fully reflected. On the contrary, a more effective selection strategy may make better use of the high-quality outputs in the existing candidate set without significantly increasing runtime. Therefore, candidate selection is not only a post-processing step in the repeated sampling pipeline, but also the core link that determines whether the smaller model can compete stably with the larger baseline.

6.4 Limitations

The first limitation in this experiment comes from the evaluation metric. Due to the parser compatibility issue in the official CrossCodeEval metric in the current environment, this study does not directly use official syntax-aware evaluation, but uses character n-gram cosine similarity as the primary quality metric. This metric can provide consistent prototype-level comparison for the completion similarity of different systems, but it cannot completely replace the official metric. Especially in code completion tasks, character-level overlap does not necessarily fully reflect syntax correctness, semantic equivalence, or repository-level functionality. Therefore, the results of this study should be understood as empirical evidence based on cosine similarity, rather than a complete evaluation of functional correctness.

This limitation also affects the interpretation of the observed improvement. A higher cosine similarity indicates that the generated completion is closer to the reference at the character n-gram level, but it does not guarantee that the completion is executable, semantically equivalent, or correct in the repository context. Therefore, the comparison between 7B repeated sampling and 32B single generation should be understood as a comparison under the chosen similarity-based proxy metric. The positive results, especially on Python, show that repeated sampling improves this proxy quality score, but they should not be read as direct evidence that the generated code is functionally correct in all cases.

The second limitation is language scope. In the end, this experiment only completes the experimental process on Python and C#, including 10 random sample groups, runtime calibration, candidate selection analysis, and statistical testing. This design ensures that the experimental process of each included language is relatively complete, but it also limits the scope of generalization of the conclusion. Python results provide strong support, and C# results provide weaker but still positive robustness evidence. However, this experiment does not cover all available languages in CrossCodeEval, so it cannot be claimed that repeated sampling will have the same effect in all

programming languages.
The third limitation comes from the candidate selection setting. The main result of this study uses the logP-selected result, because it does not rely on ground truth and can be used as a realistic inference-time selection route. In contrast, the best-cosine upper bound uses reference completion for selection, so it can only be used for upper-bound analysis. This design helps to distinguish between deployable result and candidate set potential, but it also means that this study does not provide a practical selection method that can fully utilize the candidate set. In other words, this study shows the unutilized potential in repeated sampling, but does not solve the problem of how to stably capture this potential in a real inference setting.
The fourth limitation is related to runtime measurement. This experiment uses generation-only runtime as the primary cost metric, and selects the k of 7B repeated sampling through language-specific calibration. This design makes the comparison more focused on the generation stage, but it does not cover all the costs in the complete deployment pipeline, such as model loading, preprocessing, postprocessing, memory usage, or serving throughput. In addition, the hardware setup used by 7B and 32B is not exactly the same, so the runtime comparison in this experiment is more suitable to be interpreted as a controlled experimental cost comparison, not a complete production deployment benchmark.
Finally, the statistical analysis of this experiment uses seed-level group average as the unit of analysis, and each language contains 10 random sample groups. This design is more robust than a single sample split, and it can also reflect the variation under different sample groups. However, 10 groups is still a limited number, especially in the case of C# with a small effect size and a confidence interval containing 0. More random groups may help to estimate the stability of repeated sampling more accurately. Therefore, the conclusion of this study should be understood as the empirical finding obtained under the current sample protocol, not the final judgment of all possible task samples.

6.5 Future Work

The first direction for future work is still candidate selection, because this is the most direct limitation revealed by the results of this thesis. The comparison between the logP-selected result and the best-cosine upper bound shows that the repeated-sampling candidate set already contains outputs that are closer to the reference than the candidate selected by logP. This means that the problem is not only whether the 7B model can generate useful candidates, but also whether the system can identify them without using ground truth. Future work can therefore test selection methods that remain realistic at inference time, such as reranking based on model confidence, agreement between candidates, repository-level context, or an additional verifier. This would be especially useful for the CrossCodeEval setting, where direct unit-test feedback is usually unavailable for each local completion. A better selection method may reduce the gap between logP-selected performance and the best-cosine upper bound without necessarily increasing the number of generated candidates.
A second direction is to improve the evaluation setup. This thesis uses character n-gram cosine similarity because the official CrossCodeEval metric could not be used reliably in the current parser environment. This metric provides a consistent proxy for comparing the same systems across Python and C#, but it cannot fully represent functional correctness, syntax correctness, or semantic equivalence. Future work should rerun the experiment in an environment where the

official syntax-aware metric can be used correctly, and compare whether the trend observed with cosine similarity still holds. It would also be useful to test additional code evaluation metrics, such as syntax-aware, structure-aware, or embedding-based metrics, while carefully checking whether those metrics introduce new dependencies or biases. Such evaluation would make it clearer whether the observed advantage of repeated sampling is only a similarity-based effect or whether it also corresponds to stronger code completion quality under more code-aware metrics.

A third direction is to extend the language scope. The current experiment focuses on Python and C#, with Python used as the main language and C# used as a second-language robustness check. This already shows that the effect of repeated sampling is not identical across languages: Python gives strong positive evidence, while C# gives a weaker but still positive average result. However, CrossCodeEval also contains Java and TypeScript, and these languages were not included in the final experiment because of the time and scope limits of this thesis. Running the same calibrated-budget procedure on Java and TypeScript would help determine whether the observed pattern is specific to Python and C#, or whether smaller-model repeated sampling is also competitive in other programming languages. This would also make the conclusion less dependent on two selected languages.

Future work should also test the statistical robustness of the result with a larger number of random sample groups. This thesis uses 10 random sample groups per language, with 150 tasks in each group. This is enough to show the difference between the very stable Python result and the more mixed C# result, but more groups would give a clearer estimate of variance and confidence intervals. This is especially relevant for C#, where the average difference is positive but the confidence interval includes zero. Additional random groups may clarify whether the C# result remains a weak positive tendency, becomes more stable, or disappears under a larger sampling procedure.

Another useful extension is to test more model scales and repeated-sampling budgets. This thesis compares Qwen2.5-Coder-7B repeated sampling with Qwen2.5-Coder-32B single generation, because these two models provide a clear smaller-versus-larger same-family comparison. However, the trade-off between model size and repeated sampling may look different for intermediate models, such as 14B, or for larger baselines. Testing more model sizes would make it possible to study whether there is a more efficient point between single-generation large models and repeated-sampling small models. It would also help answer whether the advantage comes mainly from the 7B model being fast enough to sample multiple times, or from a more general property of repeated sampling under a fixed generation-time budget.

The runtime setting can also be made closer to real deployment. This thesis uses generation-only wall-clock time because it allows a controlled comparison focused on the generation process itself. However, an actual code completion system is affected by more than generation-only runtime. Model loading, memory usage, batching, inference backend, request scheduling, and serving throughput may all change the practical cost of the two systems. Future work can therefore repeat the comparison with an optimized inference backend, such as vLLM, or another deployment-oriented serving setup. This would test whether the repeated-sampling advantage remains visible when the system is evaluated under conditions closer to an interactive coding assistant.

Finally, future work can explore generation settings beyond the fixed completion length used in this thesis. The current experiment uses a generation length of 10, which keeps the comparison controlled and suitable for line-level completion, but longer completions may change both quality and runtime behavior. Longer generation lengths may give the smaller model more room to make mistakes, but they may also make repeated sampling more useful because candidate variation

becomes larger. Testing different generation lengths, sampling temperatures, or decoding settings would help clarify how sensitive the observed result is to the generation configuration. Together, these future directions would give a more complete picture of when repeated sampling is a practical alternative to larger-model single generation, and when its benefit is limited by selection, evaluation, language, model scale, or deployment constraints.

7 Conclusion

This thesis studies whether Qwen2.5-Coder-7B with repeated sampling can match or outperform Qwen2.5-Coder-32B single generation in the same model family under a calibrated generation-only runtime budget. Unlike simply comparing model size, this thesis focuses on how different generation strategies affect the final project-level code completion performance under a limited generation budget.

The experimental results show that the 7B repeated-sampling system can effectively compete with the 32B single-generation baseline in the tested setting in this thesis. Python results provide the strongest evidence. The logP-selected result of 7B is higher than the 32B baseline in all random sample groups, and uses a lower generation-only runtime. C# results are more cautious. 7B is slightly higher in average quality and wins in most groups, but statistical evidence is not enough to support the same strong conclusion. Importantly, the 7B repeated-sampling system achieved these results using one NVIDIA A100 GPU, while the 32B baseline used four NVIDIA A100 GPUs. Thus, the smaller-model system achieved higher or comparable quality with lower generation-only runtime while using only one-quarter of the allocated GPU count of the larger-model baseline. Therefore, this thesis cannot claim that repeated sampling is consistently superior to the larger model across all programming languages, but it shows that a smaller model with repeated sampling can offer clear competitive and resource-efficiency advantages in the evaluated setting.

The key to this conclusion is not that the single-generation capacity of the 7B model has comprehensively exceeded that of the 32B model, but that repeated sampling has changed the output usage of the smaller model. The best-cosine upper bound is higher than the logP-selected result, indicating that there is quality space in the candidate set that has not been fully utilized by the current selection route. Therefore, the results of this thesis support a system-level view, that is, final code completion performance is not only determined by model size, but also by the combined impact of sampling strategy, candidate selection, and runtime budget.

Overall, this thesis provides positive evidence that a smaller code model with repeated sampling can serve as a resource-efficient alternative to larger-model single generation. Larger models are still an important strong baseline, but in scenarios where runtime and deployment cost are limited, smaller models through repeated sampling and more effective candidate selection may also become valuable alternative directions. The conclusion of this thesis is still limited by metric choice, language scope, and experimental setup, but it shows that continuing to study the trade-off between model scale, generation strategy, and selection method is a meaningful direction.

References

[1] 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.

[2] Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. Codet: Code generation with generated tests. *arXiv preprint arXiv:2207.10397*, 2022.

[3] 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.

[4] Yangruibo Ding, Zijian Wang, Wasi Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, et al. Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion. *Advances in Neural Information Processing Systems*, 36:46701–46723, 2023.

[5] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. In *Findings of the association for computational linguistics: EMNLP 2020*, pages 1536–1547, 2020.

[6] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999*, 2022.

[7] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*, 2020.

[8] Michael Hassid, Tal Remez, Jonas Gehring, Roy Schwartz, and Yossi Adi. The larger the better? improved llm code-generation via budget reallocation. *arXiv preprint arXiv:2404.00725*, 2024.

[9] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, et al. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938*, 2021.

[10] Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, DDL Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, et al. Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556*, 10, 2022.

[11] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*, 2024.

[12] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.

[13] Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, and Percy S Liang. Spoc: Search-based pseudocode to code. *Advances in Neural Information Processing Systems*, 32, 2019.

[14] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*, 2023.

[15] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022.

[16] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in neural information processing systems*, 36:21558–21572, 2023.

[17] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*, 2021.

[18] Ansong Ni, Srini Iyer, Dragomir Radev, Ves Stoyanov, Wen tau Yih, Sida I. Wang, and Xi Victoria Lin. Lever: Learning to verify language-to-code generation with execution, 2023.

[19] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*, 2020.

[20] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.