National College of Ireland

Comprehensive Study of Parameter Efficient Fine tuning techniques on Small Language Models for Gherkin Scenario Generation

MSc Research Project
MSCAI

Shruthi Chandra Babu
Student id:23248556

School of Computing
National College of Ireland

Supervisor: Abdul Shahid

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	Shruthi Chandra Babu
Student ID:	X23248556
Programme:	MSCAI
Year:	2025
Module:	Practicuum
Supervisor:	Abdul Shahid
Submission Due Date:	01/09/2025
Project Title:	Comprehensive Study of Parameter Efficient Fine tuning techniques on Small Language Models for Gherkin Scenario Generation
Word Count:	8174
Page Count:	21

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature:	Shruthi Chandra Babu
Date:	26th August 2025

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

Attach a completed copy of this sheet to each project (including multiple copies).	☐
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	☐
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	☐

Assignments that are submitted to the Programme Coordinator office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Comprehensive Study of Parameter Efficient Fine tuning techniques on Small Language Models for Gherkin Scenario Generation

Shruthi Chandra Babu
Student ID: X23248556

August 26, 2025

Abstract

Validating software against requirements is one of the critical phases in software development. In recent times,Test cases which aid in this validation, are written as gherkin scenarios, as they represent the test cases in a more structured way. This paper will be evaluating the potential of small language models, such as StarCoder, DeepSeekCoder , Phi and Tinyllama , to accelerate the gherkin scenario generation. In addition , this paper will also be evaluating the resource efficient methods of fine tuning such as DoRA and Hybrid LoRA, which are extensions of LoRA fine tuning . The primary motivation behind choosing small language models and fine tuning such as LoRA , is to study resource efficent models and fine tuning methods, which would make them a practical choice for real world deployments. Further, The quality of the generated gherkin scenarios will be evaluated on the basis of metrics such as BLEU, ROUGE score to analyse similarity and BERT-score to evaluate the semantic similarity.

Keywords: Small Language Models, LoRA Fine tuning , Dora Fine tuning, IA3 fine tuning, Gherkin Scenario Generation, Chain of Thought prompting technique.

1 Introduction

1.1 Background

The process of software testing has evolved a lot over a period of time. In the traditional method of software testing, any gaps, in terms of understanding the requirements, between the developer or tester, would come into light only when the tester raises a defect, during the test execution phase. To address this process issue and to facilitate the transparency in understanding between the key stakeholders: developers, testers and business analyst, shift left methodologies were introduced, which emphasized in finding the defects at earlier stages of software development.

Behavior Driven Development is a type of shift left methodology, which encourages the collaboration between developer, tester and business analyst , from the start of software development lifecycle. The test cases which were constructed in the form of gherkin scenarios are shared with the 3amigos – developer, tester and business analyst, right

from the requirement gathering phase. Owing to the simplicity and structured format, gherkin scenarios are a popular choice for discussing test cases. A simple example of gherkin scenario is as follows:

Requirement: User must be able to login to the system successfully.
Gherkin Scenario:
Given User is in the login page
When User enter the valid username and password
Then The user should be successfully logged in
And The user is taken to the Home Page

1.2 Importance

In traditional methods, creating test cases from requirements often relied on manual interpretation from the tester, which lead to omission of test scenarios or leading to inconsistencies, thereby causing insufficiency in test coverage. Hence, automating the generation of gherkin scenarios from requirements, helps the tester in overcoming these issues and providing a time efficient way to produce high quality test cases. For instance , in a software development lifecycle, a typical test case generation task would involve multiple manual tasks spanning from requirement gathering, requirement brainstorming and peer reviews.It would roughly take around 1-2 days(depending on the complexity of the requirement) to understand and come up with gherkin scenarios , to validate those requirements. The automation could bring the time down to few hours, thereby reducing manpower as well as the time taken for gherkin scenario creation.

As the capacity of Large language models proceed to advance, their capability in understanding text and generating text are improving at a rapid scale. These advancements in turn open up new possibilities of automating software development workflows, which were earlier done manually. While LLMs have demonstrated remarkable capacity in natural language understanding and generation, the major drawback with LLMs are that they require substantial GPU resources and also memory for the same, making them expensive choice for resource constrained environments . Further, many task do not require the full capability of the LLMs, yet they have to bear the full computational costs of the LLMs.

Hence, to overcome these issues, the potential of small language models are being studied in this research. Since, small language models are trained for specific tasks and are trained with mostly lesser than 3 billion parameters, they would serve as an ideal choice for performing gherkin scenario generation from requirements, in computationally challenged environments. Notably, they also have an upper hand in inference speed and execution time, as cited in [1].

In spite of the inherent capabilities of the small language models , of being computationally efficient , it should be noted that small language models are often trained for a wide range of generic task, which might not completely align with the gherkin scenario format. Hence, to bridge this gap, it would be essential to fine tune the models to convert the natural language requirements to gherkin scenarios. But , traditional fine tuning techniques such as standard fine tuning or sequential fine tuning , posed a challenge of being computationally extensive, yet again. For instance , [2] cites that performing standard fine tuning for vision-language models, requires a batch size of 32,768 and 18 days of training on 592 V100 GPUs. Hence, parameter efficient fine tuning has been observed to be effective method of finetuning , where rather than fine tuning all the parameters

for a specific task from the scratch, only parameters specific to the task are adapted for the task.

1.3 Research Question

How effective are small language models in automatically generating Gherkin scenarios from natural language requirements, and what are the optimal fine-tuning strategies for this domain-specific task?

1.4 Objective of the report

To address the above mentioned research question, the following are the objectives that are defined.

Dataset preparation: In order to evaluate efficiency of the small Language models using various fine tuning techniques, the dataset is synthetically prepared with requirements which are labelled with gherkin scenarios with the help of state of the art GPT-4 model.

Dataset size Impact for fine tuning validation: The second objective is to validate the performance Small language models fine tuned with different sizes of dataset ie., 100, 400 and 1000,

Fine Tuning Methods comparison: The third objective is to compare the performance of Small language models fine tuned with DORA against the same models fine tuned with KD-LORA fine tuning.

Comprehensive performance evaluation: Finally, the fourth objective is to measure the performance of the fine tuned Small Language models by comparing metrics such as BLEU ,ROUGE and BERT score.

1.5 Structure of the Report

The report will be adhering to the following structure:

Section I is in the introduction which comprises of background for the research, its importance, research questions and objectives. Section II includes various research papers on the effectiveness of Small language models and various fine tuning techniques and hence, motivation to have chosen the selected fine tuning techniques. Section III focuses on the methodology giving a complete overview of the experiments performed starting from dataset preparation, and choice of models. Section IV focuses on the design and implementation of the experiments. Section V discusses about the various inference obtained from various fine tuning techniques for the following experiments:

Experiment 1: Comparison of fine tuning efficiency for the models Phi, Starcoder, Deepseek and Tinyllama , when fine tuned with 100,400,1000 requirement to gherkin scenario pair.

Experiment 2: Comparison of fine tuning efficiency for the models Phi, Starcoder, Deepseek and Tinyllama , when fine tuned with DORA and IA3-LORA fine tuning methodology.

Finally , the final section is about the conclusion and future works.

2 Related Work

Automated test case generation has undergone significant transformation, where natural language processing has played a key role in converting the requirements to test cases. This section delves into the key developments in this area.

Early foundation for automating test case generation

The challenges posed by manual test case generation is that they are time consuming , as well as error prone. [3] claims that test case design claims around 40-70% of the total effort taken for software testing process. This challenge was discovered as early as 2013 and a methodology was devised for automating test case generation, in which handcrafted rules were designed after the requirements were subjected to POS tagging , parsing and constructing several knowledge graphs. Since, graph relied on fixed patterns, parsing these knowledge graph resulted in test cases which were incomplete and ambiguous, thereby offering less flexibility. Further, this methodology struggled to achieve test case coverage, which is a crucial part of test case generation.

[4] introduces UMTG framework for automating acceptance test case creation, for industrial use cases, as acceptance test cases was mandated by international standards for evaluation of safety critical systems. In this method, NLP was used to preprocess and parse the data and further formal constraints were created to capture the conditions required to trigger scenario execution, which aided in comprehensive scenario coverage, which was a limitation in [3].

A comprehensive overview of many research papers on Natural language processing , depicted that during the early 2020, NLP preprocessing techniques such as tokenization, POS(Part of speech) tagging, and TF-IDF(Term Frequency – Inverse documentation) were most commonly used , along with BERT for text generation. However, this paper also denoted the persistence of critical challenges like ambiguity , which posed a significant hurdle for text generation.

Evolution of LLMs – a comprehensive overview

The emergence of large language models have contributed towards significantly in software testing processes, particularly in test case generation. In [5], a comparative study of six general purpose LLMs ie.,BARD, ChatGPT3.5, Claude, Gemini , ChatGPT-4 and Llama3, revealed that LLMs can successfully generate test cases from requirements. Among these models BARD seemed to show consistent lower performance across various scenarios and Gemini and Llama3 were the models which successfully handled requirements to test case generation. Further , the authors highlighted that various prompting techniques proved to be some of the key factors in yielding good model performance, suggesting that improving prompting techniques in future could lead to performance enhancement.

Another key research paper which compared the performance of combination of primitive models T5 and GPT-3 in [6]. In this study, T5 was used for understanding the system requirements and boundary condition analysis ensured that good test case coverage was achieved. In combination with T5, GPT-3 was used for test case generation and this combined model achieved good coverage in test case generation, which was a limitation in majority of the papers discussed before. But , the evaluation technique for this study did not involve any metrics based evaluation, rather it only involved expert evaluation of test cases, highlighting that metrics also plays a significant role in evaluating test case generated.

The efficiency of large language models is not limited to only text generation in nat-

ural language, but it has seen to be quite efficient in generating structed code. [7] investigates about the learning capabilities of LLMs in learning and using new programming languages, merely from examples/descriptions/actual code of functions provided in the input prompts. It was observed that this prompting technique worked well with inexpensive models like Llama-2 and starcoder and models were seen to learn quite well from learning about the functions, rather than examples. Interestingly, text based models like Llama-2 performed well with text instructions, whereas starcoder preferred seeing actual code. Thus, it is evident from this paper that efficient prompting techniques provide best result, for giving structed output.

Enhancing Language Models with fine tuning.

Since, gherkin scenarios follow a structured format for test case generation, models which were used for coding could prove to be effective for generating gherkin scenarios as well. Moreover, several open sourced models seemed to provide the same performance as closed sources models like GPT. [8] compares the effectiveness of the models Deepseek-coder and GPT . This research extends the context window to 16k tokens, to improve the model's ability to handle lengthy scenarios. Compared to the base model, the instruction fine tuned model was 41% more efficient in code generation, laying a strong foundation that fine tuning on open sourced models could effectively increase the model's performance. Although efficient, higher computational resources required is seen as a limitation for Large Language models.

Small Language models which are trained with lesser than 3 billion parameters and trained only for specific tasks, offer a resources efficient way to generate text .While LLMs offer promising results, they often face barriers in deployment , as they require massive computational requirements(multiple 80GB A100/H100 GPUs) but small language models have the capability to run on standard hardware (eg., RTX 3090) but lack multistep reasoning capabilities. TinyLlama introduces a dense 1.1B parameter language model in [9] that defies traditional scaling methods by training smaller models on massive datasets (3 trillion tokens). In contrast to compute-hungry large language models, TinyLlama is competitive in performance but uses very little computational power. The model uses a three-stage training procedure, leading to specialist variants such as a Math&Code version that is specifically designed for programming. Evaluation displays better performance than comparable-sized models such as OPT-1.3B and Pythia, as the Math&Code version delivers 15.24% on HumanEval benchmarks. TinyLlama's capacity to execute on commodity hardware without loss of code generation functionality makes it best suited for automated test case generation in low-resource settings.

Parameter efficient fine tuning techniques

As more and more pre-trained models continued to evolve from being trained with few million to several billion parameters, fine tuning these models for a specific task became expensive and memory intensive. Traditional approaches for fine tuning included training these models for each task, which made costly for many organisations. Hence, many parameter efficient fine tuning techniques were introduced , in which the model's original weights were frozen , and new parameters were introduced in order to adapt the model to the particular task.

[10] discussed about an empirical method of comparing the performance of various models such as codeBERT and code T5, on various tasks such as coding summarisation , clone detection and code generation. It was observed that unlike full fine tuning, PEFT fine tuning techniques showed superior performance by utilising just 0.5% of additional parameters. Furthermore, it was observed that although PEFT fine tuning techniques

worked quite efficiently for code summarisation tasks and also for cross domain learning than full fine tuning, while it performed inferior to full fine tuning in tasks such as code generation.

One more key finding in LORA fine tuning is that , efficiency of LoRA fine tuning is not only dependent on the ranks , but also on hyperparameter tuning. [11] explores the learning capacity and measures forgetting of base model knowledge in full fine tuning vs LoRA, in Llama-7B model. It was observed that LoRA 's efficiency increased with ranks, but when compared to full fine tuning , it had better knowledge retention capacity from the base model which is evident from code CPT score of 0.545 for full fine tuning as compared to 0.617 for LoRA . Additionally, this model emphasized that apart from ranks, hyperparameter tuning is also an important factor which contributes to the performance and LoRA offers more diverse generation.

While LORA achieves significant performance improvement by just tweaking the new parameters, DORA(Weight decomposed Low Rank Adaptaion) is yet another fine tuning which offers better fine tuning capability than LORA, as it combines the power of LORA to fine tune the direction of the parameters and optimises the magnitude of the existing weights. DORA fine tuning was studied in [12], through a comprehensive analysis of VL-BART fine tuning patterns. It was discovered that LORA exhibhited positive correlation between magnitude and direction updates, and full fine tuning depicsts a negative correlation between magnitude and direction. Thus, DORA was able to show superior performance in comparison to LORA by 1-4%, while maintaining the same computational efficiency.

DORA has also been evaluated on the COLA dataset in [9], where the high rank LoRA layers are being structured into single Rank components , and the less important components are being dynamically updated during training. With only 0.3% of the original parameters from RoBERTa and BART models, DORA is able to achieve an increase in +1.48% and +1.73% respectively over the baseline models.

IA3 Fine tuning strategies

As evident from the previous papers, that LoRA exceeds in knowledge retention capacity, but struggles in learning capacity in lower ranks, IA3 fine tuning is found to be an effective solution for training in computationally constrained environment, since IA3 uses only 0.01% of the parameters used in LoRA. The development of IA3 fine-tuning approaches was exemplified by [13] and [14]. By utilising only 0.05% of the parameters in base model , the fine tuned model gave considerable performance with minimal storage overhead. Further, [14] claimed that IA3 uses only 2.62GB for fine tuning, whereas full fine-tuning requires 5.38GB for RoBERTa-base.

Thus, the related work establishes the convergence of 2 critical stream, which contribute to this thesis ie., automated test case generation from pretrained models (which were trained with lesser than 3 B parameters) and efficient fine tuning techniques which would yield better output in terms of gherkin scenarios. Further, it also underlines the importance of choosing the right metrics such as BLEU , ROUGE and BERT score to validate the quality of the gherkin scenarios generated across the models StarCoder, Phi , DeepseekCoder and Tinyllama.

3 Methodology

3.1 Dataset

For the scope of this project, the requirements were taken from :

https://data.mendeley.com/datasets/7zbk8zsd8y/1, where the requirements are defined in .txt file, in single line format. To make it more domain specific, around 1050 requirements were taken , belonging to the financial domain. These requirements are labelled with gherkin scenarios ie., one or many positive or negative scenarios per requirement. For the purpose of labelling, the state of the art gpt-4 model was employed , which automatically generated gherkin scenarios and mapped it with the requirements.

Chain of thought prompting

One of the key factor which determines the successful generation of gherkin scenarios, is the selection of suitable prompting technique. Chain of thoughts prompting technique leverages structured reasoning to parse through the requirements, understand it and generate structured BDD gherkin scenarios for the same. Chain of thought prompting breaks down the complex task of constructing gherkin scenario into multiple sub tasks, such as :

Actor identification: to determine who would be performing the action
Precondition identification: to establish the initial state of the application
Action identification: to determine the action to be performed in the application
Outcome identification: to determine the expected result from the gherkin scenario.

Additionally, by assigning the role of "expert QA engineer" to the mode, the model mimics the thought process of the expert QA engineer,and constructs the gherkin scenario with the best practices and nuances of a QA engineer.Furthermore, the instruction provided in the prompt also provides the structure of the gherkin scenario such as instructions for constructing Feature, Scenario , Given , When, Then ,And,But, which aids the model in converting the requirement to a well structure gherkin scenario.

```
# Define the chain-of-thought prompt format
chainofthought_template = """
You are an expert QA engineer. Convert the requirements into positive and negative gherkin scenarios.
In case , there are no negative scenarios, ensure that only positive scenarios are given to the user.

Instruction
Identify the actors , preconditions, actions and expected outcomes . In case of any numbers involved, keep the scenarios generic.
Ensure that there are no adjacent '\\n'
The Gherkin scenario should be constructed with the following consideration:
(i)Feature - Mandatory.Should give a brief description about the requirement being tested.
(ii) Scenario - Mandatory. Should give a brief description about the scenario from the requirement being verified.
(iii) Given - Should give the preconditions of the scenario
(iv) When - Should give the action , which would be performed in the scenario
(v) Then - Should give the expected outcome from the scenario
(vi) And - Will be used along in Given/When/Then condition as and when required
(vii) But - Will be used along in Given/When/Then condition as and when required

Ensure the following best practices are followed:
(i) The scenarios should be independent and atomic
(ii) Use Active voice for step description

Now convert this to :

Requirement: {requirement}
Gherkin Scenarios:
"""
```

Figure 1: Role based chain of thought prompt

The OpenAI api parses through the requirement inserted in the prompt and the model understands the task and constraints involved in generating gherkin scenario. The model decomposes the requirement into structured gherkin scenario based on the instruction provided.

Interannotator agreement

To assess the reliability of the gherkin scenarios generated, a representative sample of 59 requirement-gherkin scenarios were selected on the basis of around 10 features. To assess the quality of the gherkin scenarios, they were assessed on the basis of the following criteria: syntactic correctness, completeness and structure of gherkin scenario. 2 independent annotators who are expert in software testing are considered for this assessment. Each annotator independently determined whether scenarios met the expected criteria and cohen kappa score was calculated to measure the interannotator agreement. This assessment resulted in a score of 0.899, suggesting that the gherkin scenarios are suitable as training dataset.

3.2 Small Language Models

As per [1], small language models are seen to have good potential in text generation, as they offer the advantage of being computationally extensive, inspite of being trained with lesser than 3 billion parameters. The following models were selected on the basis of their architectural design and strength to handle requirement to gherkin scenario generation.

Microsoft Phi-1.5

The Phi1.5 model was a small language model developed by microsoft,which was optimised for exceptional reasoning and problem solving tasks. This model was trained with curated synthetic dataset and instructional data, making it robust for genrating structured text. It is known to produce syntactically and semantically corrrect text, owing to model being trained on high quality and filtered dataset.

Tinyllama-1.1B Tinyllama-1.1B is an extremely light weight model , derived from Llama architecture. It is known for its ability for dialogue based instructions,as it has been extensively trained on conversational contexts . It has observed that it is very efficient in low-letency inference, making it ideal for realtime gherkin scenario generation. It has good capacity to handle natural language processing, hence it has seen to be efficient with ambigious or incomplete requirement.

bigcode/starcoder2-3b Starcoder2-3b is a model trained by big code project, and it is known to be very much effective in coding related tasks, as it has been trained with a vast corpus of source code and natural language document . It demonstrates strong performance for code completion and code synthesis tasks , as it has been trained on multiple programming languages. Since , gherkin scenario also follows a structured format like coding, this model could prove to be very much efficient in handling gherkin scenario generation, by understanding natural language requirements.

deepseek-ai/deepseek-coder-1.3b-base Deepseek coder is yet another model trained for coding tasks, since it has trained on a wide mixture of programming languages, yet very light weight, since it is trained with only 1.3B pararameter. Owing to these capabilities, this model is very much capable of creating syntactically and semantically valid gherkin scenarios.

3.3 Fine Tuning Techniques

Low-Rank Adaptation (LoRA):

LoRA was chosen as the primary fine-tuning method because of its established effectiveness in domain-specific adaptation of pre-trained models. The approach freezes the initial model weights and adds trainable low-rank decomposition matrices to every transformer layer. LoRA drastically decreases the amount of trainable parameters (usually

by 99%) without any compromise on the performance compared to complete fine-tuning. To facilitate gherkin scenario generation, LoRA allows the models to acquire the particular patterns and structures necessary for translating natural language requirements into structured BDD format without catastrophic forgetting of the capabilities of the base model.

Weight-Decomposed Low-Rank Adaptation (DoRA):

DoRA was added to overcome deficiencies seen in typical LoRA fine-tuning. This sophisticated method breaks down weight updates into magnitude and direction, offering finer control over parameter adaptation. DoRA has shown better performance than LoRA across different natural language tasks, achieving 1-4% improvement at the same computational efficiency. For requirement-to-gherkin conversion, DoRA's larger learning capability should be able to more accurately represent the subtle interactions between requirement semantics and gherkin scenario structure.

Infused Adapter by Inhibiting and Amplifying Inner Activations (IA3):

IA3 was chosen because of its novel strategy to scale inner activations instead of introducing additional parameters. The method has low overhead by learning scaling vectors that modify key and value representations in attention mechanisms and feed-forward networks. IA3's parameter efficiency (only using 0.01% of original parameters) makes it especially well-suited for resource-restricted environments while still supporting effective task adaptation. Its activation-based model is well-adapted to ensure the linguistic comprehension abilities necessary for requirement interpretation.

3.4 Experimental Design

As a part of this study, 2 experiments are conducted, which aid in evaluating the performance of fine tuning techniques on various Small Language models.

3.4.1 Experiment 1: Data Scaling Analysis

The first experiment analyses the impact which different size of dataset has on Small Language models which are fine tuned with LoRA technique. This experiment was performed to comprehend the relation between data size and the ability of the model to adapt, based on the data size.

Three different dataset were taken into consideration:

Small Scale (100 samples): Small dataset to test the ability of basic adaptation

Medium Scale (400 samples): Intermediate-sized dataset for usual small-domain situations

Large Scale (1000 samples): Complete dataset showing extensive domain coverage.

Each of the models (phi1_5, TinyLlama-1.1B-Chat-v1.0, starcoder2-3b, deepseek-coder-1.3b-base) was fine tuned using same parameters with LoRA fine tuning, across the 3 volumes of dataset. Selecting same parameters across all models ensures a fair comparison is being made. Higher LoRA rank of 64 was configured, to facilitate the model to learn complex patterns such as identifying the actor, setting the preconditions etc.

Training Loss Analysis

The training loss curve for each data size depicts a unique behavior across models:

Fine tuned with 100 Samples: All models exhibit quick initial convergence with

comparatively stable loss curves. DeepSeek and TinyLlama have the smallest final training losses (∼0.8-1.0), while StarCoder has more volatility with loss values of 2.0-2.5.

Fine tuned with 400 Samples: Larger dataset size results in more robust learning dynamics. DeepSeek has better convergence properties, with stable loss less than 0.7. StarCoder is highly unstable at the beginning but later converges to about 1.0. Phi and TinyLlama remain moderately consistent.

Fine tuned with 1000 Samples: The entire dataset shows the sharpest distinction among model abilities. DeepSeek has the smoothest learning curve with optimal convergence to ∼0.6. TinyLlama and Phi have comparable stable performance around 0.7-0.8, whereas StarCoder, in spite of the initial high volatility, stabilizes at 0.9.

3.4.2 Experiment 2: Fine-Tuning Technique Comparison

Experiment-2 compares the performance of baseline models of the small language models (phi1_5, deepseek-coder-1.3b-base, starcoder2-3b and TinyLlama/TinyLlama-1.1B-Chat-v1.0), fine tuned with the parameter efficient fine tuning techniques DORA, IA3 and LoRA.

Fine-Tuning Configurations

LoRA Configuration:

The trainable parameter count was reduced by applying low-rank decomposition for both attention and feed forward layers of the model. Higher rank-64 was used to adapt the model to learn more complex patterns and a scaling factor of 16 was used to ensure that the weight was updated with ample effective yet stable adaptation.

DoRA Configuration:

The parameters were mirrored as in LoRA for a fair comparison between the models, additionally use_dora parameter was configured so that the weight decomposition was in terms of magnitude and direction components.

IA3 Configuration:

Unlike LORA and DORA, this IA3 fine tuning does not add additional parameters, but rather it reduces the 1-2% overhead for fine tuning from LoRA/DoRA method, by using only 0.01% of the original model parameters. IA3 directly works operates on the inner activations of the model, where it focuses on key, value and feed-forward component for rescaling the activations. IA3 efficiently fine tunes the model via activation scaling rather than updating full model weights. While the models Deepseek, starcoder, tinyllama use the same target modules (attention and feedforward network components, “q_proj”, “k_proj”, “v_proj”, “o_proj”, “gate_proj”, “up_proj”, “down_proj”, for phi, the following target modules were used “q_proj”, “k_proj”, “v_proj”, “dense”, “fc1”, “fc2”.

Comparative Training Loss Analysis

Comparative analysis across fine tuning techniques

LoRA and DoRA

The training loss curve for both LoRA and DoRA show similar convergence patterns, and for both the fine tuning techniques, the loss curve converged at similar loss values, with DoRA showing slightly smoother convergence in certain instances, quite notably for starcoder where the initial volatility is high, suggesting that the model is sensitive to noisy gradient, but however eventually it converged to smoothly.

IA3

The training loss for IA3 denotes a slightly different training pattern. While Deepseek, Tinyllama, Starcoder denote a similar training pattern as observed in LoRA/DoRA, IA3

has a different impact on phi denoting a higher loss variance all through training.

Model-Specific Observations

DeepSeek-Coder-1.3B: has the best training loss among all fine-tuning approaches and dataset sizes. The coding-specific pre-training of the model seems especially well-suited for the formal structure of Gherkin scenarios.

TinyLlama-1.1B: Although the smallest model, exhibits very stable training under all conditions. The conversational training basis works well for requirement-to-scenario translation.

StarCoder2-3B: Displays the greatest training loss variance, most notably in initial training stages. Nevertheless, the code generation potential of the model is realized as training advances towards eventual convergence to satisfactory loss levels.

Phi-1.5: Shows stable moderate performance for all of the experimental conditions. The synthesized training data basis offers stable but not very good adaptation for the Gherkin generation task.

3.5 Implementation Details

3.5.1 Hardware Configuration

This experiment was carried out on MacBookPro with Apple M4 Max chip and 64 GB unified memory. Metal Performance Shaders was utilised to provide acceleration to GPU computational capabilities for model training. This configuration proved to be of advantage for performing parameter efficient fine tuning by eliminating memory transfer bottle necks and promotes optimal resource utilisation. This memory management was leverages via gradient accumulation strategies to address computational challenges which is a key factor in training transformer-based models. Training precision was standardized to FP32 (32-bit floating point) across all model configurations to prioritize numerical stability over computational speed.

3.5.2 Training Pipeline

Formatting and data partitioning

The raw requirement-gherkin scenario was formatted using a zero shot prompt "For the following requirement, generate Gherkin scenarios: {item['requirement']}\n{item['generated_gherkin
to ensure that the model uses the prompt to learn the pattern and the same will later be used in the inference to get the response. The data is split using stratified splitting, since the data is a mixture of requirement-gherkin scenarios from finance domain and 80% of the data is used for training and 20% of the data is used for testing, for both the experiments.

Initialising model and tokenizer

The base models which are set are tokenised using their own tokenizer from the transformer library and EOS padding is configured to ensure that equal length is maintained throughout and to preserve the autoregressive pattern.

PEFT fine tuning

PEFT parameters were initialized with LORA/DORA/IA3 with specific attention to target module specification for each fine tuning methodology and model.

Training execution and monitoring

SFTTrainer was used performing data formatting, tokenization, padding and peft fine tuning, which were configured in the previous steps with hyperparameters such as learning

rate and AdamW optimizer, cosine learning rate scheduling, and gradient accumulation. Training metrics such as training loss, learning rate and gradient norms were captured via customised logging. Memory clean up operations were performed during training runs to prevent accumulation related degradation in performance.

Saving model for inference

Once the training is completed, the fine tuned model and the adapter weights are preserved, along with the tokenizer components as tokenizer.model, tokenizer.json and other configuration files. Base model tokenizers were also saved since the experiment would also be comparing the performance of fine tuned models against its base model. This comprehensive model preservation ensures that inference could performed seamlessly.

3.5.3 Evaluation

Both the finetuned and base model were used for evaluation, where the metrics such as BLEU score, ROUGE-L score, ROUGE-2 score and BERT score were used for evaluating the 20% dataset, which was split in data preprocessing step. Further, the gherkin scenario which was generated for each model using different fine tuning techniques were also manually reviewed, to check the effectiveness of fine tuning.

3.6 Results

3.6.1 Experiment 1: Data Scaling Impact

This experiment evaluates the metrics across different dataset sizes: ie., 100, 400 and 1000 for fine tuning models with LoRA fine tuning.

BERT Score Comparative Analysis

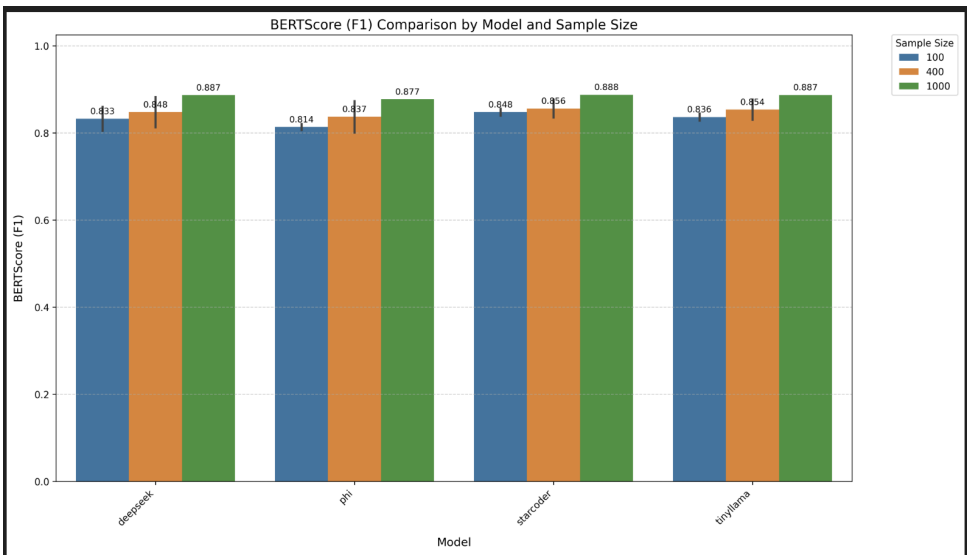

Figure 2: BERT Score Comparative Analysis by model and size

BERT Score F1 score depicts a continuous increase in scores from 100 to 1000 samples dataset. Starcoder shows the highest BERT score of 0.888 progressing from 0.848, while Tinyllama and deepseek are closer behind with a score of 0.887 from 0.833 and 0.836 respectively. Phi is observed to have moved from 0.814 to 0.877. While Starcoder has the highest BERT score, Phi shows the greatest increase in score of about 7.7%.

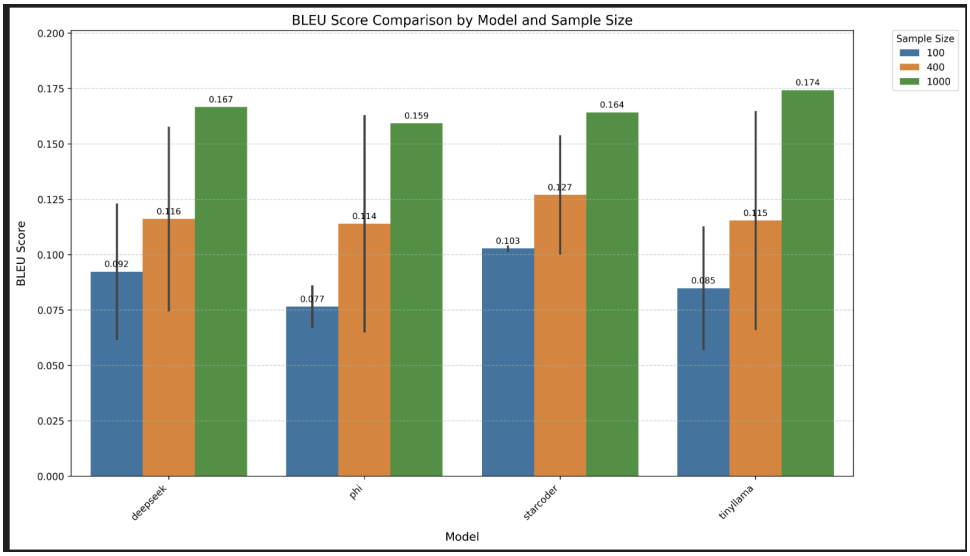

Figure 3: BLEU Score Comparative Analysis by model and size

BLEU Comparative Analysis

Tinyllama demonstrates the most significant scaling from 0.085 to 0.174, showing a 105% increase in BLEU score, Deepseek coder and starcoder show a considerable increase to 0.167 and 0.164 respectively.

ROUGE-L and ROUGE-2 Comparative Analysis

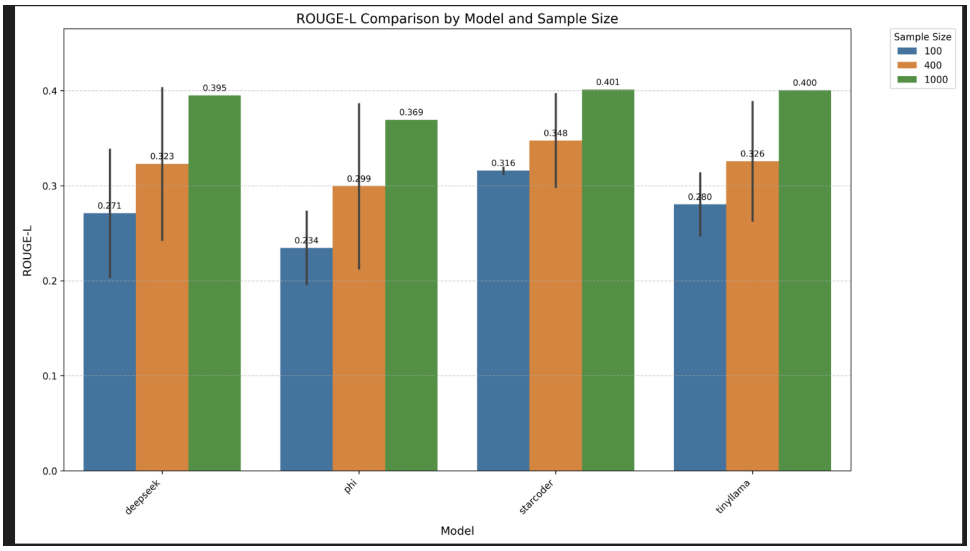

Figure 4: ROUGE-L Score Comparative Analysis by model and size

ROUGE-L scores show monotonic scaling for all models, with the most significant gains to be found between the shift from 400 to 1000 samples. DeepSeek attains the highest final ROUGE-L score of 0.395, while StarCoder and TinyLlama reach similar levels at 0.400-0.401.

ROUGE-2 scores reflect comparable scaling behavior, with notably improved performance increases for DeepSeek (0.147 to 0.258) and StarCoder (0.173 to 0.265). These results suggest larger datasets allow models to learn longer subsequence patterns necessary for coherent Gherkin scenario generation.

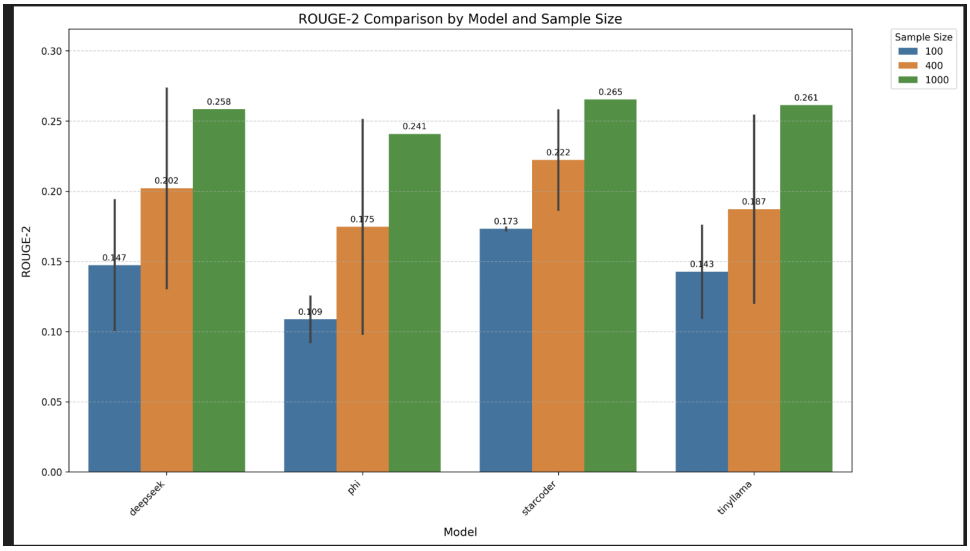

Figure 5: ROUGE-2 Score Comparative Analysis by model and size

3.6.2 Experiment 2: Fine-Tuning Technique Comparison

Base model vs Fine tuned model comparison

BERT score comparative analysis

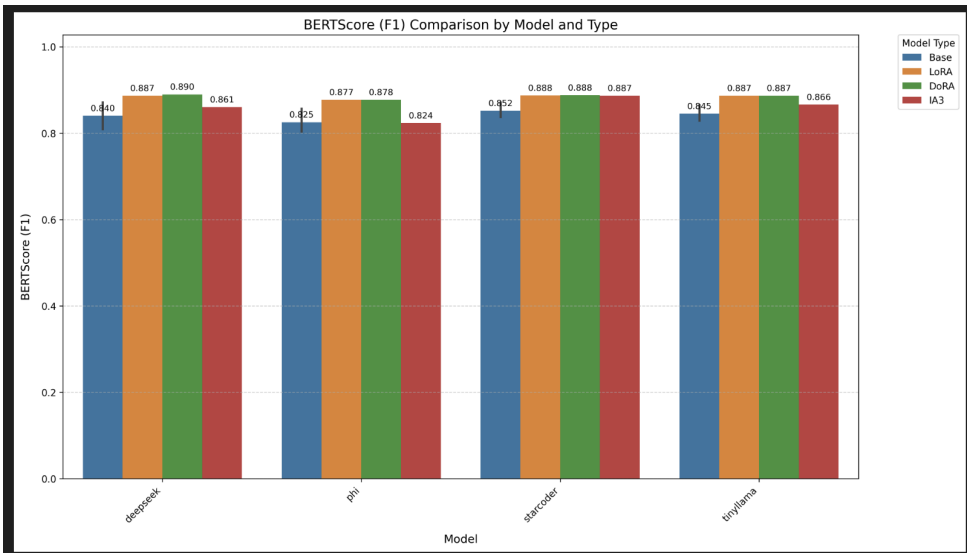

Figure 6: BERT Score Comparative Analysis by model and fine tuning method

BERT score improvements range from 4.2% increase for starcoder (0.852 to 0.888) to 6.2% increase for deepseek model (from 0.840 to 0.890). All fine tuning techniques showed significant performance with DoRA showing consistently high performance, across all models.

BLEU score comparative analysis

The BLEU scores across all models denote really strong performance, for all fine tuning techniques. Deepseek shows an increase from 0.104 to 0.167 (66% improvement), while Tinyllama shows a 74% increase (from 0.100 to 0.174). These significant amount of increase indicate that domain specific fine tuning greatly enhances the model's capability to generate Gherkin scenarios from requirement.

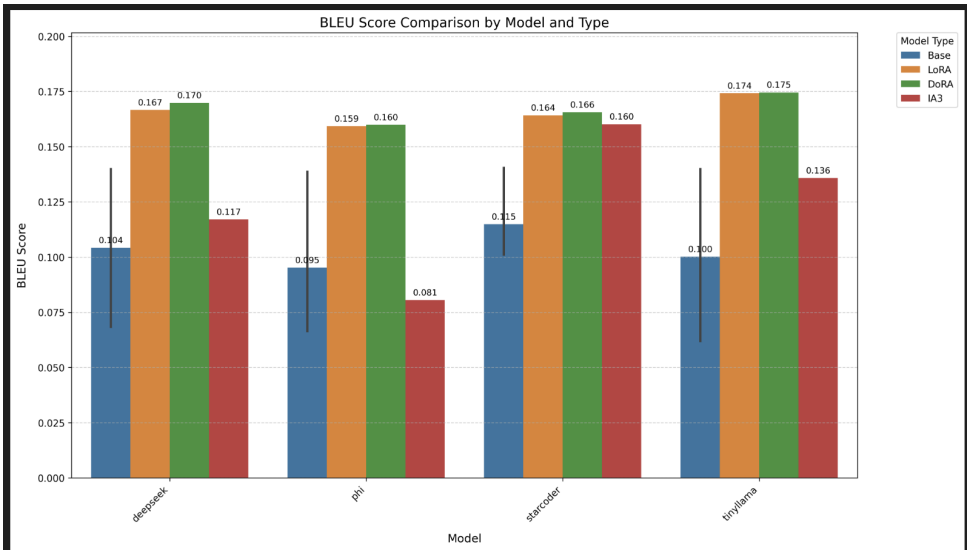

Figure 7: BLEU Score Comparative Analysis by model and fine tuning method

ROUGE-2 and ROUGE-L score

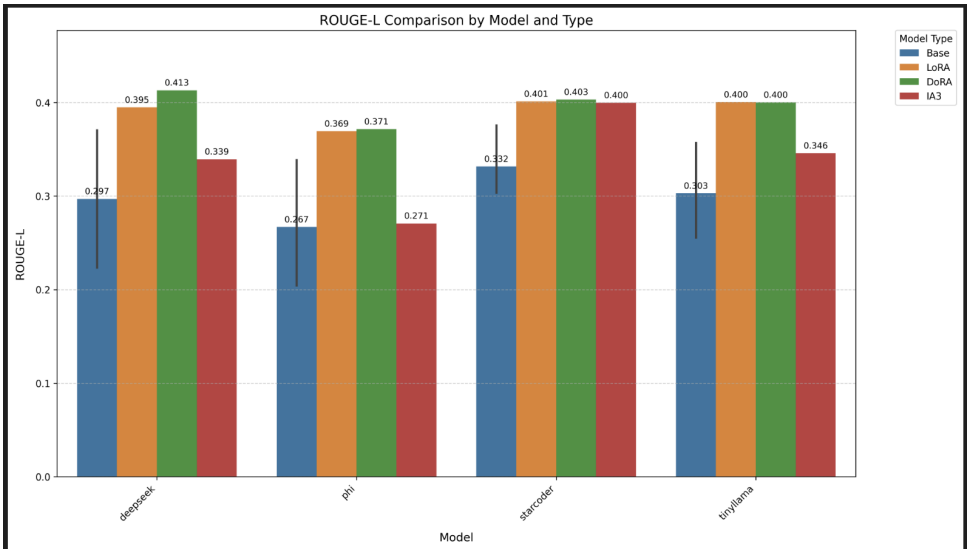

Figure 8: ROUGE-L Score Comparative Analysis by model and fine tuning method

ROUGE-L results are significant for all models, with increases from 21.1% (StarCoder: 0.332 to 0.401) to 33.6% (DeepSeek: 0.297 to 0.395).

ROUGE-2 results reflect even more stark improvements, with DeepSeek seeing a 47.4% rise (0.175 to 0.258) and StarCoder posting a 33.7% increase (0.198 to 0.265).

Comparison of Fine Tuning Technique Performance

LoRA vs DoRA vs IA3

LoRA and DoRA show very close numbers in terms of all metrics, with minimal variation in performance. BERT score is almost similar for all the models, and shows only marginal difference of about 0.003. BLEU scores show similar patterns, but DoRA looks to have advantages in some of the models (such as Deepseek coder with 0.167 and 0.170 for LoRA and DoRA respectively).

Rouge Metrics also does not show significant difference between LoRA and DoRA,

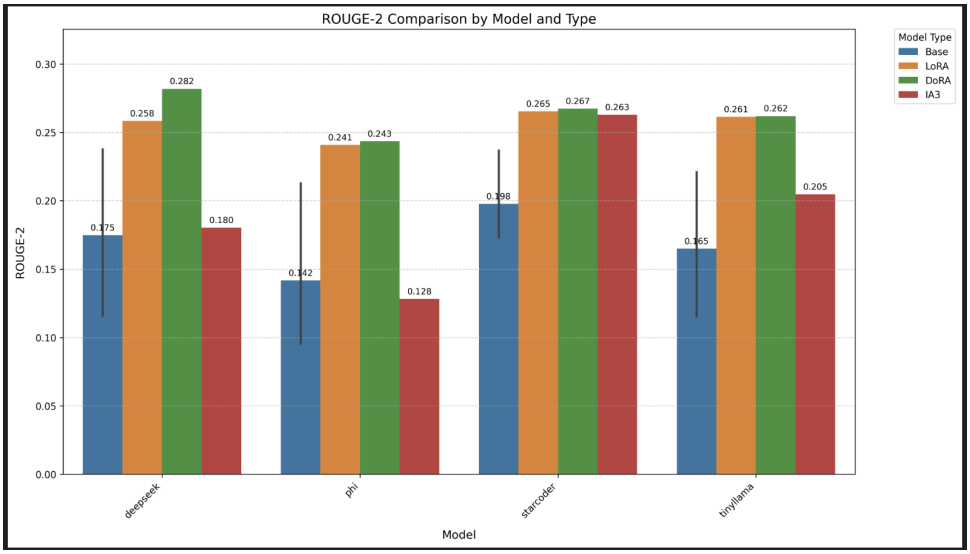

Figure 9: ROUGE-2 Score Comparative Analysis by model and fine tuning method

while in some models like deepseek and starcoder, it demonstrates slightly better performance.

IA3 fine tuning metrics shows a very different picture, when compared to LoRA and DoRA. Although, IA3 underperforms across all metrics, when compared to LoRA/DoRA, starcoder appears to have achieved metrics closer to LoRA and DoRA fine tuning. While, Deepseek and Tinyllama metrics for IA3 demonstrates better performance over baseline models, surprisingly phi is seen to underperform for IA3 fine tuning compared to even the baseline model. Thus, the performance of IA3 varies based on the model.

Model Specific Performance Analysis

Deepseek

Deepseekcoder consistently yielded better performance with fine tuning techniques. Across all metrics DoRA was seen to have outperformed other models with an increase in 5.9%, 63.4%, 61.1%, 39.05% in the metrics BERT score, BLEU score, ROUGE-2 and ROUGE-L score respectively. Thus, as denoted in [15], the deepseek coder model is good in its ability to generate structured gherkin scenarios, since it was pre-trained on 87% of coding source. This gave deepseekcoder an edge to perform better with fine tuning.

StarCoder

StarCoder is yet another model, which performed quite well on semantic understanding tasks. The model scored the maximum BERTScore F1 of 0.888 using DoRA fine-tuning, which is a 4.2% gain over the baseline model (0.852). In BLEU scoring, StarCoder revealed steady performance against LoRA (0.164) and DoRA (0.166), which means a 42-45% gain over baseline (0.113).

ROUGE scores showed StarCoder's well-balanced ability to extract both local and global text patterns. ROUGE-L scores were enhanced from 0.332 (base) to 0.401 (LoRA/DoRA), which showed a 21% improvement. ROUGE-2 performance was from 0.198 to 0.265 (DoRA), which showed a 33.8% improvement. IA3 fine-tuning worked competitively well for StarCoder and was observed to yield metrics closer to LoRA/DoRA than other models, which indicates greater compatibility with activation scaling methods.

TinyLlama

TinyLlama showed the most extreme scaling behavior, with a superlative lexical overlap score performance. The model showed outstanding BLEU score gain from 0.100

(base) to 0.174 (LoRA), which was a whooping 74% improvement - the highest among all models which were compared. This indicates TinyLlama's excellent ability to learn fine-grained token-level patterns in Gherkin scenario generation.

BERTScore gains were significant, with the model progressing from 0.843 (base) to 0.887 (DoRA), reflecting better semantic coherence. ROUGE-L performance improved consistently to 0.400, whereas ROUGE-2 scores achieved 0.261-0.262 on LoRA/DoRA methods. Fine-tuning IA3 indicated lesser gains (0.136 BLEU, 0.205 ROUGE-2), implying the model gains more from parameter addition than activation scaling.

Phi

Phi exhibited inconsistent performance across various fine-tuning methodologies. The model realized substantial BERTScore improvements from 0.825 (base) to 0.878 (DoRA) - which was a 6.4% improvement. BLEU scores were enhanced from 0.095 to 0.160 (DoRA) – which gave an increase of 68%.

ROUGE metrics indicated Phi's ability to capture longer sequences of dependencies, with ROUGE-L being improved from 0.267 to 0.371 (DoRA) - an increase of 39%. ROUGE-2 also reflected the same trends, increasing from 0.142 to 0.243 (DoRA). But the greatest fluctuation in IA3 scores was seen for Phi, with significantly lower scores (0.081 BLEU, 0.128 ROUGE-2) that even dropped below baseline performance from time to time. This indicates that Phi's architecture is no less incompatible with activation scaling methods, signalling that phi's architecture has delicate balances in attention/normalization that finds it challenging for extra scaling.

The model's 7.7% gain in BERTScore from 100 to 1000 samples (the best among all models) shows excellent scalability with more training data, which indicates that Phi gains a lot from larger fine-tuning data.

3.6.3 Analysis of Generated Gherkin Scenarios

Base Model

Deepseek: Gherkin scenario has incomplete and incorrect sentences. Gherkin structure is incorrect, with no feature defined

Phi: Gherkin structure not followed, neither are any scenarios framed for the requirement

Starcoder: Gherkin structure is not maintained and neither are scenarios created. Unrelated content is generated

Tinyllama: Gherkin structure is partially correct and duplicate scenarios are repeated multiple times.

The response from the base model clearly shows these models were not pre-trained effectively for generating gherkin scenarios, hence, fine tuning methods such as DoRA, LoRA and IA3 were leveraged to improve the model's capability to generate gherkin scenarios. The following section shows the response from the fine tuned models.

Models finetuned with DoRA

Deepseek: The structure format of the gherkin scenario is well maintained. Positive and Negative scenarios are captured well. The sentences are complete and correct. Although, edge negative cases were captured, it fails to capture the direct negative scenario ie., in perfect system condition, the application takes more than one minute to receive market data.

Phi: The first 2 scenarios show Good structure in gherkin scenario and capture positive and negative gherkin scenario effectively. Additional junk information (which do

not relate to gherkin scenario) are obtained from the model, in the last 2 lines. The third scenario has flaws in gherkin structure and also hallucinates to mention that the market prices should not be fetched, if the delay is greater than 1 second.

Starcoder: Positive and negative scenarios are captured efficiently, in fewer scenarios. Gherkin structure is well maintained. The Feature information is lengthy and contains redundant content, although not incorrect.

Tinyllama: Gherkin scenario is well maintained. This model hallucinates with additional information such as "And the application has been recently launched with new timestamp". Positive scenario is captured correctly, but the negative scenario takes the application in a different direction, possibly validating the application against incorrect application behavior.

DoRA fine tuning was found to be effective on starcoder, which generated well structured and scenarios aligned with requirements, with not much hallucination. Tinyllama and Deepseek also provided better quality gherkin scenarios than the base model, but around 10% hallucination were seen in the gherkin scenarios generated. Phi had significantly improved from its base model response, but still was lagging behind in terms of hallucinating unrelated content, within gherkin scenarios as well as after the completion of gherkin scenarios.

Models finetuned with LoRA

Deepseek: The gherkin scenario is same as the one generated via DoRA

Phi: Positive and negative scenarios were generated, but with lot of hallucination in all scenarios. For instance, the requirement cites that live market prices should be fetched, but the generated gherkin scenario mentions that the live market price should be fetched on specific currency. Additional junk information (which do not relate to gherkin scenario) are obtained from the model, in the last 1 lines, as in DoRA fine tuning

Starcoder: The scenarios are similar to the ones generated in DoRA fine tuning.

Tinyllama: Gherkin scenario structure is well maintained. The model does not hallucinate, as seen in DoRA fine tuning. Incomplete scenarios are generated.

LoRA fine tuning was found to be effective on starcoder and Deepseekcoder, which generated well structured and scenarios aligned with requirements, with not much hallucination. Tinyllama generated better quality scenarios without hallucination when compared to DoRA fine tuning, but the only drawback is that it generated incomplete scenario at the end. The gherkin scenario generated from phi was of inferior quality, similar to the one generated via DoRA fine tuning.

Models finetuned with IA3

Deepseek: The gherkin scenario is same as the one generated via LoRA/DoRA, except that the final scenario was incomplete.

Phi: The gherkin scenario structure is not well formed (Feature and Given keyword are missing). Also, the content against when and then keyword are also incorrect or includes a lot of hallucination. Incomplete scenario is seen at the end

Starcoder: The scenarios are similar to the ones generated in DoRA fine tuning.

Tinyllama: Gherkin scenario structure is well maintained. The content in Given is incorrect and the negative scenario does not cover direct negative scenario ie., the scenario which hampers response received in 1 second.

IA3 fine tuning was found to be effective on starcoder and Deepseekcoder, which generated well structured and scenarios aligned with requirements, with not much hallucination, except for one setback in deepseek, which is incomplete scenario generation at the end. Tinyllama generated well structured gherkin scenario, but the statement against

the keywords Given, When, Then were incorrect and further negative scenarios were not effectively captured, depicting a deterioration in the quality of scenarios generated when compared to LoRA and DoRA. Phi consistently underperformed in generating in gherkin scenarios, with quality inferior to the ones generated in LoRA and DoRA.

4 Discussion

This study analysed the data scaling effects and parameter efficient fine tuning techniques (LoRA, DoRA, IA3) on small language models for Gherkin scenario generation.

4.0.1 Data Scaling Experiment

Experiment 1- Sample size 100 vs 400 v1000												
Model Name	BLEU score			BERT score			ROUGE-L score			ROUGE-2 score		
	Samplesize 100	Samplesize 400	Samplesize 1000	Samplesize 100	Samplesize 400	Samplesize 1000	Samplesize 100	Samplesize 400	Samplesize 1000	Samplesize 100	Samplesize 400	Samplesize 1000
Deepseek	0.092	0.116	0.167	0.833	0.848	0.887	0.271	0.323	0.395	0.147	0.202	0.258
Phi	0.077	0.114	0.159	0.814	0.837	0.877	0.234	0.299	0.369	0.109	0.175	0.241
StarCoder	0.103	0.127	0.164	0.848	0.856	0.888	0.316	0.348	0.401	0.173	0.222	0.265
Tinyllama	0.085	0.115	0.174	0.836	0.854	0.887	0.28	0.326	0.4	0.143	0.187	0.261

Figure 10: Comparative Analysis of Metrics for Data Scaling Experiment

It was observed that all the metrics showed considerable increase in performance when fine tuning with 400 and 1000 data samples. The training loss curve for the data scaling experiment indicated that all models converged smoothly towards the final iteration. This includes the highly volatile star coder, which eventually converges smoothly.

Among the inference metrics, showed a the models phi and tinyllama showed a dramatic increase of 106.5% and 104.7% respectively for the BLEU score. Similarly, the ROUGE-2 and ROUGE-L score, for phi and tinyllama are noted to be remarkably high ie., 121 and 57% for phi and 82.5 and 42.9% for tinyllama. Inspite of its dramatic increase, phi is seen to construct Gherkin scenario with inappropriate scenarios and additionally also provides irrelevant content in the end. Likewise, while the quality of gherkin scenarios is considerably high, it was still giving incomplete scenarios. This suggests that BLEU scores and ROUGE-2/ROUGE-L scores, which measure overlapping n-grams and longest common subsequences respectively, do not effectively capture model performance, as these overlap-based metrics do not necessarily correlate with optimal output quality. In comparison, although the BERT score across models are close enough with each better, they correlate slightly better with the quality of gherkin scenarios generated. The higher BERT score for starcoder, is seen to correlate with the better quality of gherkin scenario generated for starcoder.

4.0.2 Performance Comparison across Fine Tuning Techniques

Experiment 2- LoRA vs DoRA vs IA3 fine tuning																
Model Name	BLEU score				BERT score				ROUGE-L score				ROUGE-2 score			
	Base	LoRA	DoRA	IA3	Base	LoRA	DoRA	IA3	Base	LoRA	DoRA	IA3	Base	LoRA	DoRA	IA3
Deepseek	0.104	0.167	0.17	0.117	0.84	0.887	0.89	0.861	0.297	0.395	0.413	0.339	0.175	0.258	0.282	0.18
Phi	0.095	0.159	0.16	0.081	0.825	0.877	0.878	0.824	0.267	0.369	0.371	0.271	0.142	0.241	0.243	0.128
StarCoder	0.115	0.164	0.166	0.16	0.852	0.888	0.888	0.887	0.332	0.401	0.403	0.4	0.198	0.265	0.267	0.263
Tinyllama	0.1	0.174	0.175	0.136	0.845	0.887	0.887	0.866	0.303	0.4	0.4	0.346	0.165	0.261	0.262	0.205

Figure 11: Comparative Analysis of Metrics across Fine Tuning Methods

The fine-tuning experiment across LoRA, DoRA, IA3, and base models points to enormous performance gains across parameter-efficient approaches, with LoRA consistently producing higher results. Deepseek experiences a 60.6% BLEU score gain through LoRA (0.167 vs 0.104), while Tinyllama records a stunning 74% boost (0.174 vs 0.100). DoRA follows closely, registering 63.5% improvement for Deepseek. These drastic metric gains carry through to ROUGE scores, with respective dramatic boosts across models.

BERT scores yield higher quality indicators with more consistent reliability, featuring moderate yet significant gains. StarCoder using DoRA obtains the highest BERT score (0.888) and has better correlation with semantic coherence and contextual appropriateness of the produced scenarios.

IA3 is seen to be underperforming than LoRA and DoRA, across models, suggesting that scaling pre-trained activations provides less flexibility than scaling through learnable low-rank parameter updates.

4.0.3 Critical Assessment and Limitations

The dataset size (100-1000 samples) is a set back to evaluation standards, which typically would require 3,000+ samples. Improvements may represent statistical noise rather than genuine scaling effects, particularly without significance testing.

Further, with the aforesaid dataset volume, there was no reliable advantage of LoRA over DoRA observed. While, IA3's underperformance indicates architectural incompatibilities, further investigation could be made to analyse if hyperparameter tuning, could scale the models, effectively. While all the other models, showed substantial increase in metrics from base model, phi's underperformance needs investigation into architectural factors, which might be driving this.

Despite limitations, this study provides practical PEFT technique guidance and quantitative support for code-specialized models in structured text generation, while identifying critical research gaps requiring larger-scale investigation.

References

[1] "The rise of small language models," *IEEE Computer*, vol. 58, no. 1, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/10897262

[2] "The rise of small language models," *IEEE Computer*, vol. 58, no. 1, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/10897262

[3] R. P. Verma and M. R. Beg, "Generation of test cases from software requirements using natural language processing," in *2013 6th International Conference on Emerging Trends in Engineering and Technology (ICETET)*. IEEE, 2013, pp. 140–147. [Online]. Available: https://ieeexplore.ieee.org/document/6754807

[4] C. Wang, F. Pastore, A. Goknil, and L. C. Briand, "Automatic generation of acceptance test cases from use case specifications: An nlp-based approach," *IEEE Transactions on Software Engineering*, vol. 48, no. 2, pp. 585–616, 2022. [Online]. Available: https://ieeexplore.ieee.org/document/9103626

[5] B. Korraprolu, P. Pinninti, and Y. Reddy, "Test case generation for requirements in natural language - an llm comparison study," in *Proceedings of the 18th Innovations*

in Software Engineering Conference, ser. ISEC 2025. ACM, 2025, pp. 1–5. [Online]. Available: https://dl.acm.org/doi/10.1145/3717383.3717389

[6] A. Mathur, S. Pradhan, P. Soni, D. Patel, and R. Regunathan, "Automated test case generation using t5 and gpt-3," in *2023 9th International Conference on Advanced Computing and Communication Systems (ICACCS)*, vol. 1. IEEE, 2023, pp. 1986–1992. [Online]. Available: https://ieeexplore.ieee.org/document/10112971

[7] A. Patel, S. Reddy, D. Bahdanau, and P. Dasigi, "Evaluating in-context learning of libraries for code generation," *arXiv preprint arXiv:2311.09635*, 2024, updated version published on Apr 4, 2024. [Online]. Available: https://arxiv.org/abs/2311.09635

[8] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. Li, F. Luo, Y. Xiong, and W. Liang, "Deepseek-coder: When the large language model meets programming – the rise of code intelligence," *arXiv preprint arXiv:2401.14196*, 2024. [Online]. Available: https://arxiv.org/abs/2401.14196

[9] P. Zhang, G. Zeng, T. Wang, and W. Lu, "Tinyllama: An open-source small language model," 2024. [Online]. Available: https://arxiv.org/abs/2401.02385

[10] Y. Mao, K. Huang, C. Guan, G. Bao, F. Mo, and J. Xu, "Dora: Enhancing parameter-efficient fine-tuning with dynamic rank distribution," 2024.

[11] J. Wang, Y. Yang, and B. Xia, "A simplified cohen's kappa for use in binary classification data annotation tasks," *IEEE Access*, vol. 7, pp. 164 386–164 397, 2019.

[12] J. Liu, C. Sha, and X. Peng, "An empirical study of parameter-efficient fine-tuning methods for pre-trained code models," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2023, pp. 397–408.

[13] M. Xu, Y. Lu, Y. Shen, S. Zhang, D. Zhao, and C. Gan, "Hyper-decision transformer for efficient online policy adaptation," in *International Conference on Learning Representations*, 2023. [Online]. Available: https://arxiv.org/pdf/2304.08487

[14] L. Xu, H. Xie, S.-Z. J. Qin, X. Tao, and F. L. Wang, "Parameter-efficient fine-tuning methods for pretrained language models: A critical review and assessment," *arXiv preprint arXiv:2312.12148*, 2023. [Online]. Available: https://arxiv.org/pdf/2312.12148

[15] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. Li, F. Luo, Y. Xiong, and W. Liang, "Deepseek-coder: When the large language model meets programming – the rise of code intelligence," *arXiv preprint arXiv:2401.14196*, 2024. [Online]. Available: https://arxiv.org/abs/2401.14196