National College of Ireland

Configuration Manual

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):	

Configuration Manual – Gherkin Scenario Generation

Shruthi Chandra babu
X23248556

1 Introduction

This manual provides information about the environment in which this project was run.

2 System Configuration

This work was performed on a MacBook Pro with an Apple M4 Max chip and 64 GB unified memory under macOS Sequoia 15.5. Integrated GPU capabilities of Apple Silicon were achieved with Metal Performance Shaders (MPS), which was used for model training and inference acceleration. This setup was beneficial for parameter-efficient fine-tuning by removing memory transfer bottlenecks between CPU and GPU, allowing resource usage to be optimized with the unified memory architecture. The Metal Performance Shaders framework was used to supply GPU computational acceleration to transformer-based model operations, especially useful for the matrix computations of LoRA, DoRA, and IA3 fine-tuning methods.

3 Pre Requisites

- Python 3.8+ is required to run the project.
- pip install -r requirements.txt is run to install all the libraries required for the project. The following libraries will be installed:

Figure 1: Required Python libraries for the project

4 Execution stages

(1) The input data ie., the requirements pertaining to finance domain are collected and stored in the folder data/unlabelledData. This data is labelled with gherkin scenarios using state of the art model : GPT-4 from open-ai, using role based chain of thought prompt.

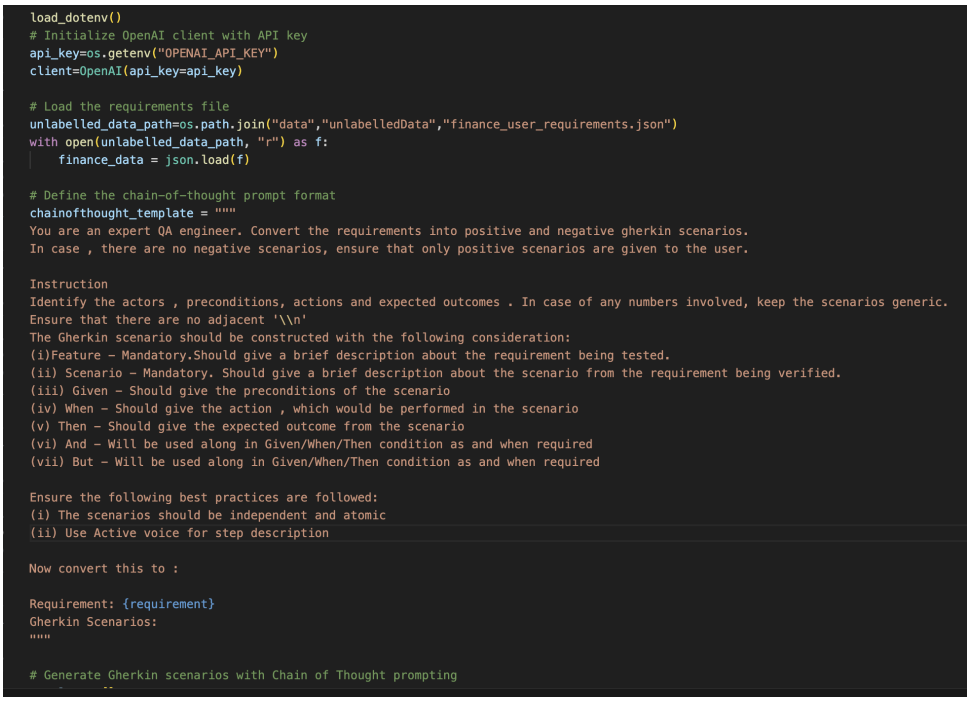

Figure 2: Data collection and labeling process workflow

(2) To assess the quality of the labelled gherkin scenarios against the requirements , inter-annotator agreement was established , using cohen Kappa score . A survey of 50 requirements -gherkin scenarios were taken and its quality was assessed using 2 annotators. The assessed gherkin scenarios were further processed using cohen Kappa method, to verify if the dataset quality is suitable for fine tuning.

```
# Load config.yaml
with open("config/config.yaml", "r") as file:
    config = yaml.safe_load(file)

# Get the Excel file path from config and load it
survey_path = config["path"]["gherkin_scenario_survey"]
survey_path = os.path.abspath(survey_path)
df = pd.read_excel(survey_path)

# Extract annotator columns and map Yes/No  to numbers
annotator1 = df['Annotator 1(Shruthi Chandra Babu)']
annotator2 = df['Annotator 2(Kathija Afrose Sathar)']
mapping = {'Yes': 1, 'No': 0}
annotator1_num = annotator1.map(mapping)
annotator2_num = annotator2.map(mapping)

# Calculate Cohen's Kappa score
kappa = cohen_kappa_score(annotator1_num, annotator2_num)
print(f"Cohen's Kappa Score: {kappa:.3f}")
```

Figure 3: Quality assessment using Cohen's Kappa score

(3) The dataset obtained is taken as input for further experiments:

Experiment 1:

In this experiments, a subset of the dataset were taken to evaluate the performance of LoRA fine tuning with different volumes of data, on small language models. The data set size taken were 100, 400 and 1000.

Illustration code for fine tuning deepseek coder model with data with 100 samples. Main.py is invoked which subsequently invokes the fine tuning script.

The requirement-gherkin scenarios are formatted with prompt and split into testing and training dataset. The data is then tokenized and padded with EOS. LoRA configuration is performed to set parameters like rank, dropout etc.. Following this , PEFT is performed with adamW optimizer and entire pipeline is run using SFT trainer. The model, token and adapters are saved in a folder , for later use in inference.

The finetune script is run using the following command:

```
# Run finetune on DeepSeek model on 100 samples
python3 main.py --model deepseek --mode finetune_100 --input data/input/100_finance_with_chain_of_thought_gherkin.json --config config/config.yaml
```

Figure 4: Fine-tuning script execution command

```python
def run(input_path, config):
    # Load requirements and gherkin scenarios(labelled data)
    with open(input_path, "r") as f:
        raw_data = json.load(f)

    formatted_data = [{
        "text": f"For the following requirement, generate Gherkin scenarios: {item['requirement']}\n{item['generated_gherkin_scenarios']}"
    } for item in raw_data]

    train_data, test_data = train_test_split(formatted_data, test_size=0.2, random_state=42)
    train_dataset = Dataset.from_list(train_data)

    #Save test data for inference
    test_data_path = config["inference"]["test_data_100"]
    os.makedirs(os.path.dirname(test_data_path), exist_ok=True)

    with open(test_data_path, "w") as f:
        json.dump(test_data, f, indent=2)
    print(f"Test data saved to: {test_data_path}")

    # Set MPS device
    device = torch.device("mps" if torch.backends.mps.is_available() else "cpu")

    # Load model and tokenizer
    base_model = "deepseek-ai/deepseek-coder-1.3b-base"
    token=config["huggingface"]["token"]
    tokenizer = AutoTokenizer.from_pretrained(base_model,token=token)
    model = AutoModelForCausalLM.from_pretrained(base_model,token=token)
    model = model.to(device)
    model.config.use_cache = False
    model.config.pretraining_tp = 1

    tokenizer.pad_token = tokenizer.eos_token
    tokenizer.padding_side = "right"
```

Figure 5: LoRA configuration and training setup

```python
    # LoRA config
    peft_params = LoraConfig(
    r=64,
    lora_alpha=16,
    lora_dropout=0.1,
    bias="none",
    task_type="CAUSAL_LM",
    target_modules=[
    "q_proj", "k_proj", "v_proj", "o_proj",
    "gate_proj", "up_proj", "down_proj"
]
    )

    # Output directories from config
    output_dir = config["deepseek"]["finetuned_100_model_dir"]
    logging_dir = config["deepseek"]["training"]["logging_dir"]
    metrics_100_dir = config["deepseek"]["training"]["metrics_100_dir"]

    os.makedirs(output_dir, exist_ok=True)
    os.makedirs(logging_dir, exist_ok=True)
    os.makedirs(metrics_100_dir, exist_ok=True)

    training_params = TrainingArguments(
    output_dir=output_dir,
    num_train_epochs=3,
    per_device_train_batch_size=1,
    gradient_accumulation_steps=16,
    optim="adamw_torch",
    save_steps=25,
    logging_steps=1,
    learning_rate=2e-4,
    weight_decay=0.001,
    fp16=False,
    bf16=False,
    max_grad_norm=0.3,
    warmup_ratio=0.03,
    group_by_length=True
```

Figure 6: Model training and fine-tuning process

```
# Initialize metrics logger
metrics_logger = TrainingMetricsLogger(output_dir=metrics_100_dir)

# Fine-tuning trainer
trainer = SFTTrainer(
    model=model,
    train_dataset=train_dataset,
    peft_config=peft_params,
    args=training_params,
    callbacks=[metrics_logger]
)

# Start training
trainer.train()

# Save the final model
trainer.save_model(output_dir)

# Generate final metrics plot
metrics_logger.plot_metrics()

# Save LoRA adapter weights
trainer.model.save_pretrained(output_dir)

# Save the tokenizer
tokenizer.save_pretrained(output_dir)

# Ensure all tokenizer files are present

base_model_dir = snapshot_download(base_model, allow_patterns=[
    "tokenizer.model", "tokenizer.json", "tokenizer_config.json", "special_tokens_map.json"
])
required_files = [
    "tokenizer.model",
    "tokenizer.json",
    "tokenizer_config.json",
    "special_tokens_map.json"
```

Figure 7: Training pipeline and model saving

Apart from the saved models, the training loss curve is also generated for the fine tuned model. A comparison of all training loss for fine tuned models can be performed using the following script.

```python
def load_metrics(model_dir: Path, sample_size: str) -> Dict[str, Any]:
    """Load metrics from a model's training directory."""
    metrics_file = model_dir / sample_size / 'training_metrics.json'
    with open(metrics_file, 'r') as f:
        metrics = json.load(f)

    if 'step' not in metrics or not metrics['step']:
        metrics['step'] = list(range(len(metrics['train_loss'])))
    for key, value in metrics.items():
        if isinstance(value, (list, tuple)):
            metrics[key] = np.array(value)

    return metrics

Windsurf: Refactor | Explain
def plot_metrics(metrics_dict: Dict[str, Dict[str, Any]], metric_name: str,
                 title: str, output_file: Path, sample_size: str, log_scale: bool = False,
                 figsize=(12, 6)):
    """Plot metrics across different models."""
    plt.figure(figsize=figsize)

    for model_name, metrics in metrics_dict.items():
        if metric_name in metrics:
            clean_name = model_name.replace('_', ' ').replace('finetune', '').strip().title()
            plt.plot(metrics[metric_name], label=clean_name, linewidth=2)

    plt.title(f'{title} (Sample Size: {sample_size})', fontsize=14)
    plt.xlabel('Training Steps', fontsize=12)
    plt.ylabel(metric_name.replace('_', ' ').title(), fontsize=12)
    plt.legend(fontsize=10)
    plt.grid(True, alpha=0.3)

    if log_scale:
        plt.yscale('log')

    plt.tight_layout()
    output_file.parent.mkdir(parents=True, exist_ok=True)
    plt.savefig(output_file, dpi=300, bbox_inches='tight')
```

Figure 8: Training metrics comparison script

```python
Windsurf: Refactor | Explain
def find_metrics_files(base_dir: str, model_name: str, sample_size: str) -> List[str]:
    """Find all metrics files for a given model and sample size."""
    metrics_dir = Path(base_dir) / model_name / sample_size
    if not metrics_dir.exists():
        return []
    return list(metrics_dir.glob("metrics.json"))

Windsurf: Refactor | Explain
def load_metrics_for_model(model_dir: Path, model_identifier: str) -> Dict[str, Any]:
    """Load metrics for a model.

    Args:
        model_dir: Directory containing the model's metrics
        model_identifier: Either 'dora', 'ia3', or a sample size (e.g., '100', '400', '1000')
    """
    metrics_path = model_dir / model_identifier / 'training_metrics.json'
    return load_metrics_file(metrics_path)

Windsurf: Refactor | Explain
def load_metrics_file(metrics_path: Path) -> Dict[str, Any]:
    """Load metrics from a JSON file."""
    with open(metrics_path, 'r') as f:
        return json.load(f)

Windsurf: Refactor | Explain
def generate_comparison_plots(base_dir: str, models: List[str], sample_sizes: List[str], output_base_dir: Path):
    """Generate comparison plots for multiple models and sample sizes."""
    base_path = Path(base_dir)

    output_base_dir = output_base_dir / 'training_metrics_comparison'
    dora_dir = output_base_dir / 'dora'
    ia3_dir = output_base_dir / 'ia3'
    dora_dir.mkdir(parents=True, exist_ok=True)
    ia3_dir.mkdir(parents=True, exist_ok=True)

    all_base_metrics = {size: {} for size in sample_sizes}
    all_dora_metrics = {size: {} for size in sample_sizes}
    all_ia3_metrics = {size: {} for size in sample_sizes}
```

Figure 9: Loss visualization implementation

```
for sample_size in sample_sizes:


    for model in models:
        model_dir = base_path / model

        metrics = load_metrics_for_model(model_dir, sample_size)
        all_base_metrics[sample_size][model] = metrics


        dora_metrics = load_metrics_for_model(model_dir, 'dora')
        all_dora_metrics[sample_size][f"{model}_dora"] = dora_metrics


        ia3_metrics = load_metrics_for_model(model_dir, 'ia3')
        all_ia3_metrics[sample_size][f"{model}_ia3"] = ia3_metrics



for sample_size, metrics in all_base_metrics.items():
    output_dir = output_base_dir / f'sample_{sample_size}'
    output_dir.mkdir(parents=True, exist_ok=True)

    plot_combined_metrics(
        metrics,
        f'Base Models - Training Loss (Sample Size: {sample_size})',
        output_dir / 'training_loss.png'
    )


plot_combined_metrics(
    all_dora_metrics['1000'],
    'DoRA Models - Training Loss (Sample Size: 1000)',
    dora_dir / 'training_loss.png'
)
```

Figure 10: Model performance analysis code

```
    plot_combined_metrics(
        all_ia3_metrics['1000'],
        'IA3 Models - Training Loss (Sample Size: 1000)',
        ia3_dir / 'training_loss.png'
    )


Windsurf: Refactor | Explain
def plot_individual_metrics(metrics_dict: Dict[str, Dict[str, Any]],
                            sample_size: str, output_dir: Path):
    """Plot individual metrics for all models."""

    plot_metrics(
        metrics_dict,
        'train_loss',
        'Training Loss Comparison',
        output_dir / 'training_loss.png',
        sample_size
    )

Windsurf: Refactor | Explain
def plot_combined_metrics(metrics_dict: Dict[str, Dict[str, Any]],
                          title: str,
                          output_path: Path
                          ):
    """Plot combined training loss for multiple models with the same sample size."""
    plt.figure(figsize=(12, 6))

    for model_name, metrics in metrics_dict.items():
        if 'train_loss' in metrics:

            display_name = model_name.split('/')[-1].replace('_', ' ').title()
            plt.plot(metrics['train_loss'], label=display_name)

    plt.title(title, fontsize=14)
    plt.xlabel('Training Steps', fontsize=12)
    plt.ylabel('Training Loss', fontsize=12)
    plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
```

Figure 11: Plotting configuration and setup

```
    plt.grid(True, alpha=0.3)
    plt.tight_layout()


    output_path.parent.mkdir(parents=True, exist_ok=True)
    plt.savefig(output_path, bbox_inches='tight', dpi=300)
    plt.close()

Windsurf: Refactor | Explain
def plot_combined_metrics_all_sizes(metrics_dict: Dict[str, Dict[str, Any]],
                                    title: str,
                                    output_path: Path,
                                    model_type: str):
    """Plot combined training loss for all sample sizes in one plot."""
    plt.figure(figsize=(12, 6))


    colors = plt.cm.tab10.colors
    model_colors = {}


    all_models = set()
    for size_metrics in metrics_dict.values():
        all_models.update(size_metrics.keys())


    for i, model in enumerate(sorted(all_models)):
        model_colors[model] = colors[i % len(colors)]


    for sample_size, size_metrics in metrics_dict.items():
        if not size_metrics:
            continue

        for model_name, metrics in size_metrics.items():
            if 'train_loss' in metrics:
                display_name = f"{model_name.replace(f'_{model_type}', '')} ({sample_size})"
                plt.plot(
```

Figure 12: Results generation and output

```
                plt.plot(
                    metrics['train_loss'],
                    label=display_name,
                    color=model_colors[model_name],
                    linestyle='--' if '100' in sample_size else ('-.' if '400' in sample_size else '-')
                )

    plt.title(title, fontsize=14)
    plt.xlabel('Training Steps', fontsize=12)
    plt.ylabel('Training Loss', fontsize=12)
    plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
    plt.grid(True, alpha=0.3)
    plt.tight_layout()

    output_path.parent.mkdir(parents=True, exist_ok=True)
    plt.savefig(output_path, bbox_inches='tight', dpi=300)
    plt.close()

Windsurf: Refactor | Explain | Generate Docstring
def main():
    parser = argparse.ArgumentParser(description='Compare training metrics across models and sample sizes')
    parser.add_argument('--base-dir', type=str, required=True,
                        help='Base directory containing model metrics')
    parser.add_argument('--models', type=str, nargs='+', required=True,
                        help='List of model names to compare')
    parser.add_argument('--output-dir', type=str, default='comparison_plots',
                        help='Directory to save comparison plots')
    args = parser.parse_args()


    sample_sizes = ['100', '400', '1000']


    output_dir = Path(args.output_dir)
    output_dir.mkdir(parents=True, exist_ok=True)


    generate_comparison_plots(args.base_dir, args.models, sample_sizes, output_dir)
```

Figure 13: Final metrics processing

The above script is run using the command:

```
python3 utils/compare_training_metrics.py \
    --base-dir training_metrics \
    --models deepseek phi starcoder tinyllama \
    --output-dir comparison_plots
```

Once all the finetuned models are created.

5 Inference

The test data which was split earlier was loaded and using the same prompt (which was used to format the model during fine tuning), the inference is generated, by loading the fine tuned model . The inference from the fine tuned model is obtained and the metrics such as BERT score, BLEU score, ROUGE-2, ROUGE-L score are calculated, from the inference . The inference obtained are stored as csv file.

The following command is used to run the inference:

```
# Run inference with DeepSeek model trained on 100 samples
python3 main.py --model deepseek --mode inference_100 --input data/testdata/test_data_100.json --config config/config.yaml
```

Figure 14: Inference command execution

```
Windsurf: Refactor | Explain | Generate Docstring
def run(input_path, config):
    gc.collect()

    if torch.backends.mps.is_available():
        torch.mps.empty_cache()
        device = torch.device("mps")
        pipeline_device = 0
    else:
        device = torch.device("cpu")
        pipeline_device = -1

    # Load test data from config
    test_data_path = input_path
    with open(test_data_path) as f:
        test_data = json.load(f)

    results = []
    references = []
    finetuned_predictions = []
    base_predictions = []

    # Get requriement and gherkin scenario from test_data
    raw_data = [{"requirement": item["text"].split("\n")[0].replace("For the following requirement, generate Gherkin scenarios: ", ""),
                 "generated_gherkin_scenarios": "\n".join(item["text"].split("\n")[1:]).strip()}
                 for item in test_data]
```

Figure 15: Inference script setup and initialization

```
# Load finetuned model and tokenizer
finetuned_model_path = config["deepseek"]["finetuned_100_model_dir"]
finetuned_tokenizer = AutoTokenizer.from_pretrained(finetuned_model_path, trust_remote_code=True)
finetuned_model = AutoModelForCausalLM.from_pretrained(finetuned_model_path, trust_remote_code=True).to(device)
finetuned_pipe = pipeline(
    "text-generation",
    model=finetuned_model,
    tokenizer=finetuned_tokenizer,
    device=pipeline_device,
    max_new_tokens=300,
    do_sample=False
)

# Load base model and tokenizer
base_model_path = config["deepseek"]["base_model_dir"]
base_tokenizer = AutoTokenizer.from_pretrained(base_model_path, trust_remote_code=True)
base_model = AutoModelForCausalLM.from_pretrained(base_model_path, trust_remote_code=True).to(device)
base_pipe = pipeline(
    "text-generation",
    model=base_model,
    tokenizer=base_tokenizer,
    device=pipeline_device,
    max_new_tokens=300,
    do_sample=False
)

for item in raw_data:
    requirement = item["requirement"]
    reference = item["generated_gherkin_scenarios"]
    formatted_prompt = f"For the following requirement, generate Gherkin scenarios: {requirement}\n"

    # Inference from fine-tuned model
    finetuned_outputs = finetuned_pipe(formatted_prompt)
    finetuned_gherkin = finetuned_outputs[0]["generated_text"].replace(formatted_prompt, "").strip()
```

Figure 16: Model loading and inference process

```
    # Inference from base model
    base_outputs = base_pipe(formatted_prompt)
    base_gherkin = base_outputs[0]["generated_text"].replace(formatted_prompt, "").strip()

    results.append({
        "requirement": requirement,
        "reference": reference,
        "finetuned_gherkin": finetuned_gherkin,
        "base_gherkin": base_gherkin
    })

    references.append(reference)
    finetuned_predictions.append(finetuned_gherkin)
    base_predictions.append(base_gherkin)

# Calculate metrics
bleu = load("bleu")
rouge = load("rouge")
bertscore = load("bertscore")

# Calculate metrics for fine-tuned model
bleu_finetuned = bleu.compute(predictions=finetuned_predictions, references=references)
rouge_finetuned = rouge.compute(predictions=finetuned_predictions, references=references)
bertscore_finetuned = bertscore.compute(
    predictions=finetuned_predictions,
    references=references,
    lang="en"
)
```

Figure 17: Metrics calculation (BERT, BLEU, ROUGE scores)

```python
# Calculate metrics for base model
bleu_base = bleu.compute(predictions=base_predictions, references=references)
rouge_base = rouge.compute(predictions=base_predictions, references=references)
bertscore_base = bertscore.compute(
    predictions=base_predictions,
    references=references,
    lang="en"
)

# Format BERTScore results
Windsurf: Refactor | Explain | Generate Docstring
def format_bertscore(score_dict):
    return {
        'precision': sum(score_dict['precision']) / len(score_dict['precision']),
        'recall': sum(score_dict['recall']) / len(score_dict['recall']),
        'f1': sum(score_dict['f1']) / len(score_dict['f1'])
    }

bertscore_finetuned = format_bertscore(bertscore_finetuned)
bertscore_base = format_bertscore(bertscore_base)

# Create a DataFrame and add metrics to it
metrics_data = []
metrics_data.append({
    'model': 'fine-tuned',
    'bleu': bleu_finetuned['bleu'],
    'rouge1': rouge_finetuned['rouge1'],
    'rouge2': rouge_finetuned['rouge2'],
    'rougeL': rouge_finetuned['rougeL'],
    'rougeLsum': rouge_finetuned['rougeLsum'],
    'bertscore_precision': bertscore_finetuned['precision'],
    'bertscore_recall': bertscore_finetuned['recall'],
    'bertscore_f1': bertscore_finetuned['f1']
})
```

Figure 18: Results processing and CSV output

```
    metrics_data.append({
        'model': 'base',
        'bleu': bleu_base['bleu'],
        'rouge1': rouge_base['rouge1'],
        'rouge2': rouge_base['rouge2'],
        'rougeL': rouge_base['rougeL'],
        'rougeLsum': rouge_base['rougeLsum'],
        'bertscore_precision': bertscore_base['precision'],
        'bertscore_recall': bertscore_base['recall'],
        'bertscore_f1': bertscore_base['f1']
    })

    metrics_df = pd.DataFrame(metrics_data)
    output_csv_path = "inference_results_100_deepseek.csv"
    metrics_df.to_csv(output_csv_path, index=False)

    print(f"\nMetrics saved to: {output_csv_path}")


    return output_csv_path
```

Figure 19: Inference results and evaluation metrics

Experiment 2:

The performance of the fine tuning techniques LoRA vs DoRA vs IA3 will be evaluated. As, mentioned in the previous experiment, the fine tuning script will finetune the base model and save the model. The inference script will load the test data and run the inference with the fine tuned model and provide the metrics for evaluation.

Further, apart from the metrics, the fine tuned model is used for generating gherkin scenario for a sample requirement and the gherkin scenario is stored in txt file for manual evaluation. The gherkin scenario will be obtained by running the scripts in the folder: Inference/inference_forGeneratingGherkinScenarios, which has the following content.

```python
from transformers import AutoModelForCausalLM, AutoTokenizer, pipeline
import yaml
import os
with open("config/config.yaml", "r") as f:
    config = yaml.safe_load(f)

# Load the fine-tuned model
model_name = config["deepseek"]["finetuned_1000_model_dir"]
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForCausalLM.from_pretrained(model_name, device_map="auto")

# Sample requirement
requirement = "The application should fetch live market prices with a latency of less than 1 second."

# Create the prompt
prompt = f"For the following requirement, generate Gherkin scenarios: {requirement}\n"

# Run inference
generator = pipeline("text-generation", model=model, tokenizer=tokenizer, max_length=300)
result = generator(prompt)[0]['generated_text']
#Create directory
output_dir = "generated_gherkin"
os.makedirs(output_dir, exist_ok=True)
# Save to file
output_file = "generated_gherkin/lora_deepseek.txt"
with open(output_file, "w", encoding="utf-8") as f:
    f.write(result)
```

Figure 20: Gherkin scenario generation scripts and implementation