SAKARYA UNIVERSITY JOURNAL OF COMPUTER AND INFORMATION SCIENCES
http://saucis.sakarya.edu.tr/
e-ISSN: 2636-8129
Publisher: Sakarya University
Vol. 8, No. 4, 637-650, 2025
DOI: 10.35377/saucis...1722643
RESEARCH ARTICLE

Fine-tuning Large Language Models for Turkish Flutter Code Generation

Buğra Alperen Uluırmak[1*], Rifat Kurban[1]

[1,1] Department of Computer Engineering, Abdullah Gül University, Kayseri, Turkiye, ror.org/00zdyy359

Corresponding author:
Buğra Alperen Uluırmak
Department of Computer Engineering,
Abdullah Gül University, Kayseri, Turkiye
bugraalperen.uluirmak@agu.edu.tr

Article History:
Received: 18.06.2025
Revised: 13.07.2025
Accepted: 14.07.2025
Published Online: 13.10.2025

ABSTRACT
The rapid advancement of large language models (LLMs) for code generation has largely centered on English programming queries. This paper focuses on a low-resource language scenario, specifically Turkish, in the context of Flutter mobile app development. Two representative LLMs (a 4B-parameter multilingual model and a 3B code-specialized model) on a new Turkish question-and-answer dataset for Flutter/Dart are fine-tuned in this study. Fine-tuning with parameter-efficient techniques yields dramatic improvements in code generation quality: Bilingual Evaluation Understudy (BLEU), Recall-Oriented Understudy for Gisting Evaluation (ROUGE-L), Metric for Evaluation of Translation with Explicit Ordering (METEOR), Bidirectional Encoder Representations from Transformers Score (BERTScore), and CodeBLEU scores show significant increases. The rate of correct solutions increased from ~30–70% (for base models) to 80–90% after fine-tuning. The performance trade-offs between models are analyzed, revealing that the multilingual model slightly outperforms the code-focused model in accuracy after fine-tuning. However, the code-focused model demonstrates faster inference speeds. These results demonstrate that even with very limited non-English training data, customizing LLMs can bridge the gap in code generation, enabling high-quality assistance for Turkish developers comparable to that for English. The dataset was released on GitHub to facilitate further research in multilingual code generation.

Keywords: Code generation, Large language models, Fine-tuning, Low-resource languages, Flutter

1. Introduction

Fine-tuning and Low-Rank Adaptation (LoRA) represent two indispensable techniques in the domain of natural language processing (NLP) and more broadly in machine learning, particularly for large language models (LLMs). With the advent of powerful models such as BERT and GPT-3, the need for effective methods to adapt these pre-trained models to specific tasks has become increasingly urgent. The literature surrounding these techniques is burgeoning, underscoring advances in their application, performance efficiency, and associated challenges.

Fine-tuning and transfer learning are closely related but conceptually distinct techniques in natural language processing and machine learning. Transfer learning refers to leveraging knowledge gained from a pre-trained model on one task or domain and applying it to a different task or domain. Fine-tuning, on the other hand, is a specific form of transfer learning that involves continuing the training process of a pre-trained model on a new, typically smaller, task-specific dataset. It has been widely adopted to enhance model performance across various downstream tasks such as sentiment analysis, named entity recognition, and contextual language understanding [1],[2],[3]. However, fine-tuning is not without its challenges; one such issue is catastrophic forgetting, where the model may lose important general knowledge acquired during pre-training, especially if the fine-tuning dataset is small or biased [4],[5].

To mitigate such drawbacks, research has explored various fine-tuning strategies that seek to optimize knowledge transfer from pre-trained models while preserving their foundational capabilities. Low-rank adaptation (LoRA) has emerged as a prominent method among these strategies. Introduced by Hu et al., LoRA freezes the weights of pre-trained models and introduces trainable low-rank matrices within each model layer, significantly reducing the number of parameters that need adjustment during fine-tuning [6]. This approach has proven beneficial in constraining computational and memory requirements while achieving performance comparable to traditional full fine-tuning methods [7].

In the landscape of fine-tuning methods, traditional full fine-tuning, where all model parameters are adjusted, often leads to inefficiencies, especially with large models. In contrast, techniques such as parameter-efficient fine-tuning (PEFT) and adapter layers have gained traction [2],[8]. These methods enable the selective adjustment of portions of the network, introducing new parameters only for the task at hand while preserving most of the model's weights, thereby facilitating a more efficient training regime [9]. Furthermore, research indicates that such methods reduce computational burdens and

Cite as: B. A. Uluırmak and R. Kurban, "Fine-tuning Large Language Models for Turkish Flutter Code Generation," *Sakarya University Journal of Computer and Information Sciences*, vol. 8, no. 4, pp. 637-650, 2025. doi: 10.35377/saucis...1722643

enhance the model's interpretability and robustness by maintaining a broader scope of knowledge from the initial pre-training phase, Sulaiman & Hamzah [10],[11].

Scientific investigations into fine-tuning methodologies have demonstrated their significance in specialized tasks, such as medical question-answering tasks [10] and offensive language detection [12]. These applications demonstrate the ease with which fine-tuned models can adapt to niche domains while leveraging the robustness of their pre-trained counterparts. However, it is essential to note that the efficacy of these fine-tuning approaches can vary significantly based on the dataset quality, the training protocol, and hyperparameter settings, suggesting room for further optimization and experimentation [13],[14].

The diversity of fine-tuning strategies is further underscored by the emergence of prompt-based tuning mechanisms, which aim to encode task-specific instructions directly into the model's inputs rather than adjusting its parameters. This technique has gained popularity as it can often yield competitive performance with minimal computational overhead [15]. By integrating prompts, models can exhibit adaptability that sidesteps some of the more cumbersome aspects of conventional fine-tuning, allowing for streamlined deployment in various practical scenarios.

Moreover, the choice of evaluation metrics for assessing the performance of fine-tuning methods remains pivotal. The literature emphasizes the necessity of employing comprehensive and robust evaluation frameworks that quantify the stability, interpretability, and efficiency of fine-tuned models [16]. Evaluating models trained with LoRA and other PEFT methods should consider both accuracy and factors such as computational efficiency and memory usage, especially as model sizes continue to scale upward [17].

While fine-tuning and LoRA are powerful tools within the NLP framework, they also demand careful management to avert overfitting. Studies have shown that fine-tuned models can exhibit heightened susceptibility to noise and adversarial inputs, necessitating adversarial training or contrastive learning techniques to bolster their robustness under various conditions [18],[19]. In this light, the ongoing evolution of fine-tuning strategies must consider evolving best practices to enhance resilience against adversarial phenomena.

In summary, the rapidly evolving landscape of fine-tuning and LoRA methodologies marks a crucial turning point for the field of NLP. As researchers continue to innovate and refine these techniques, focusing on parameter efficiency, computational effectiveness, and domain adaptability will remain integral to advancing future LLM applications. This literature review presents a comprehensive overview of current fine-tuning and LoRA, aligning foundational theories with practical advancements and implications for future research directions.

In this article, we comprehensively evaluated fine-tuned and base LLMs for Turkish code generation. A new Turkish Flutter Q&A code dataset is introduced, and two representative models —a multilingual 4B and a code-focused 3B —are fine-tuned using Low-Rank Adaptation. Results are reported for standard code-generation metrics (BLEU, ROUGE-L, METEOR), exact-match accuracy, and functional correctness on a test set of coding questions. Model performance trade-offs are analyzed, showing that targeted fine-tuning dramatically improves Turkish code generation capability while incurring only a modest impact on inference speed. This is the first study focusing on code generation for the Turkish language and the Flutter framework, demonstrating how even relatively small models can be adapted to a low-resource language/domain to achieve strong results. All data and training scripts are publicly available to facilitate future research on multilingual and domain-specific code generation.

2. Writing Equations Dataset and Fine-tuning Setup

In this section, the dataset and fine-tuning setup for LLM are described.

2.1 Dataset

The Flutter Code with Questions dataset, a novel collection of Turkish question-and-answer pairs for the Flutter framework (which uses the Dart programming language), is generated within the scope of this study. This dataset comprises approximately 1,500 Turkish questions about Flutter development, each paired with an expert-level answer in the form of Dart code (often with brief explanations or comments). The questions cover a broad spectrum of Flutter topics, from fundamental concepts (UI widgets, state management) to more advanced scenarios (asynchronous operations, platform integration). For example, one question asks: *"Bir ListView içinde liste elemanlarını nasıl yenileyebilirim?"* (translation: *"How can I refresh list items inside a ListView?"*). The corresponding answer is a snippet of Dart/Flutter code implementing the solution (in this case, using a setState() call or a RefreshIndicator widget). Relevant Flutter code samples were automatically collected from public GitHub repositories to construct this dataset. Turkish questions aligned with these code snippets were generated using the Gemini 2.0 Flash language model and subsequently curated for correctness and clarity. Additional entries were derived from translated documentation and original contributions by native Turkish developers. Each entry provides a problem description in Turkish and a code solution in Dart that fulfills the request.

Before use, all answers were verified for correctness and basic functionality. The dataset includes a variety of question types, some of which ask for an explanation or difference between Flutter concepts (yielding an explanatory code snippet or pseudo-code). In contrast, others require implementing a specific function or feature. Each code answer is typically a self-contained function or widget that could be integrated into a Flutter app. For training purposes, each Q&A pair is formatted as a single prompt by concatenating the question and its corresponding answer, in alignment with the formatting conventions used during

the original pre-training of the models. Specifically, special tokens denoting the user query and the assistant response are prepended accordingly. For example, a formatted prompt would be:

<|begin_of_text|> <|start_header_id|>user<|end_header_id|> Bir ListView içinde liste elemanlarını nasıl yenileyebilirim? <|eot_id|> <|start_header_id|>assistant<|end_header_id|>

Followed by the Dart code answer and an end-of-text token. We reserved some of the data for evaluation: 10 representative questions (with known correct solutions) were held out as a qualitative and quantitative assessment test set. These test questions span various difficulty levels to ensure our evaluation covers straightforward API usage and more complex logic or state management scenarios. The remaining ~1,400 Q&A pairs were used for fine-tuning in our experiments. This focused Turkish dataset challenges the model to translate a problem described in a natural language (Turkish) into a correct solution in a programming language (Dart), pushing the model's cross-lingual and domain-specific generalization abilities.

2.2 Fine-Tuning Method

Parameter-efficient fine-tuning is employed to adapt the models to this domain. Low-Rank Adaptation (LoRA) is used to insert small, trainable weight matrices into each layer of a pre-trained model, thereby significantly reducing the number of parameters that must be updated while keeping the original weights frozen. Mathematically, LoRA modifies the original pretrained weight matrix W ∈ ℝ^{d × k} by introducing a low-rank decomposition as follows:

$$W' = W + BA \tag{1}$$

here, B ∈ ℝ^(d×r) and A ∈ ℝ^(r×k) are trainable low-rank adapter matrices, and r ≪ min(d, k) is the low rank chosen for this decomposition. Initially, matrices A and B are initialized to zero, ensuring the model starts from its pretrained state. The gradient updates during training are computed using:

$$\nabla_{a,\beta} L(A, B) = \partial L/\partial W' \times \partial W'/\partial (A, B) \tag{2}$$

This approach can match the performance of full fine-tuning methods at significantly reduced computational and memory costs.

Quantized Low-Rank Adaptation (QLoRA) further improves efficiency by quantizing the model weights into lower precision (typically 4-bit) during training. Specifically, quantization compresses the weights using the following formula:

$$Q(W_{ij}) = round(((W_{ij} - W\_min) / (W\_max - W\_min)) \times (2^4 - 1)) \tag{3}$$

Where W_min and W_max denote the minimum and maximum values of weights, respectively. During backpropagation, gradients are approximated through these quantized weights using straight-through estimators (STE):

$$\partial L/\partial W_{ij} \approx \partial L/\partial Q(W_{ij}) \tag{4}$$

This quantization significantly reduces memory consumption, enabling fine-tuning large-scale models even on limited hardware resources, such as single GPUs. These advances have enabled efficient fine-tuning of large code models on relatively small and specialized datasets, dramatically boosting task-specific performance.

Parameter-efficient fine-tuning is employed to adapt the models to this domain. Low-Rank Adaptation (LoRA) is used to insert small, trainable weight matrices into each layer of a pre-trained model, thereby significantly reducing the number of parameters that must be updated while keeping the original weights frozen [13]. This technique can match the performance of full fine-tuning at a tiny fraction of the compute cost [13]. Quantized Low-Rank Adaptation (QLoRA) further improves efficiency by quantizing the model weights to 4-bit during training and backpropagating through these compressed weights [14], enabling even 65B-parameter models to be fine-tuned on a single GPU [14]. These advances have enabled fine-tuning large code models on relatively small, specialized datasets. For example, the WizardCoder project applied an Evol-Instruct fine-tuning method to an existing code model (StarCoder), dramatically boosting its performance on code benchmarks and surpassing some closed-source models [15].

In this study, two pre-trained LLMs – a 4B multilingual model (Gemma-3 4B, from Google's Gemini research) and a 3B code-oriented model (Qwen-2.5-Coder-3B, from Alibaba) – are fine-tuned on our Turkish Flutter dataset using LoRA/QLoRA. The Gemma-3 model is designed to support over 140 languages and has a 128K token context window. Although it was not primarily trained on code, its broad multilingual knowledge makes it a strong candidate for adaptation to Turkish prompts. Qwen-2.5-Coder is a specialized code generation model with 3B parameters; it was pre-trained on a massive multi-language code corpus and instruction-tuned for coding tasks, which gives it strong out-of-the-box performance in code writing [10]. However, Qwen's training data skewed heavily toward English problem descriptions, like most code models. By fine-tuning Qwen on Turkish prompts, we aim to impart an understanding of Turkish phrasing and Flutter-specific terminology that it otherwise lacks. Conversely, fine-tuning Gemma-3 (which likely has strong Turkish comprehension due to multilingual training) should teach it the Dart/Flutter API patterns it hasn't seen. Comparing these two models provides insight into the benefits of a multilingual base model versus a code-trained base model on the task, as well as how each benefits from targeted fine-tuning.

Both models were fine-tuned using Low-Rank Adaptation (LoRA) and 4-bit weight quantization (QLoRA). Trainable adapter matrices with a rank of 16 were inserted into each transformer layer while keeping the original weights frozen [13]. LoRA was applied to all major components of the self-attention and feed-forward networks, including the query, key, value, and

output projection matrices. These adapter layers were initialized with zeros to allow fine-tuning to begin from the pre-trained state. To minimize memory consumption, gradient checkpointing was enabled. For the Qwen-2.5-Coder model, the NF4 quantization format [14] was employed to load 4-bit weights, and LoRA was applied on top, following the QLoRA approach. This configuration enabled the 3-billion-parameter model to be fine-tuned efficiently on a single 24 GB GPU. The Gemma-3 model (4B) was fine-tuned similarly; although it can be trained with 16-bit precision without quantization, 4-bit precision was chosen to approximately double the training throughput and maintain consistency across models. The Unsloth training framework was utilized, offering memory-efficient optimizers and incorporating FlashAttention and gradient checkpointing to accelerate training. The training setup included three epochs over approximately 1,400 training samples (after test splitting), an effective batch size of eight sequences (achieved through a per-device batch size of four with two gradient accumulation steps), the AdamW optimizer, a learning rate of 2×10^{-4}, linear warmup for the first three steps, and cosine learning rate decay. Early stopping was not applied due to the limited number of epochs and the absence of overfitting. Fine-tuning was completed in approximately 20 minutes for Qwen-2.5-Coder and 1.5 hours for Gemma-3 on a single NVIDIA A100 GPU with 40 GB of memory. The longer context window and more complex architecture of Gemma-3 contributed to its increased training time.

Each Turkish question was provided as input during fine-tuning, and the model was trained to generate the corresponding Dart answer. A maximum sequence length of 2048 tokens was set to accommodate multi-line code outputs, although most examples were significantly shorter. SentencePiece tokenizers were used for both the Gemma and Qwen models, and no modifications were made to their vocabularies. This decision was based on the observation that common Turkish words and Dart syntax tokens were either already included or could be effectively processed through subword tokenization. This assumption was validated, as both models could tokenize Turkish text and Dart code without issues. Upon completion of training, the LoRA adapter weights were merged into the base model weights to produce standalone fine-tuned models suitable for inference. Additionally, the fine-tuned Gemma model was converted into the GGUF format to enable efficient deployment on CPUs, while the Qwen model was evaluated directly in 4-bit precision using a GPU.

Figure 1 illustrates the fine-tuning pipeline employed in this study. Each Turkish question in the dataset was converted into a model prompt (user message), and the corresponding Dart code was transformed into the expected completion (assistant message). The pre-trained models, Gemma-3 or Qwen-2.5, were then fine-tuned on these prompt–completion pairs using Low-Rank Adaptation (LoRA) and 4-bit weight quantization. The diagram shows that LoRA adapter modules were inserted into the transformer layers and trained to minimize discrepancies between the generated and reference code outputs. This efficient fine-tuning approach enabled effective domain adaptation despite limited training data and computing resources.

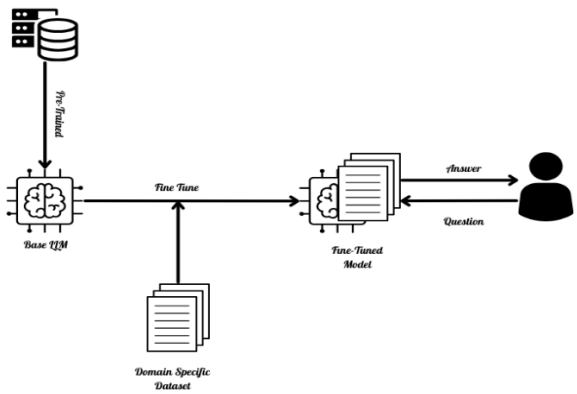

Figure 1. Fine-Tuning Pipeline.

Figures 2 and 3 show the fine-tuning loss curve and gradient norms for Gemma-3, and Figures 4 and 5 show the same for Qwen-2.5-Coder. The loss steadily decreases for each model without divergence, and gradient norms remain well-behaved, indicating stable training. Fine-tuning successfully adapted both LLMs to the Turkish Flutter domain.

The training loss for Gemma-3 (4B) over 615 steps (3 epochs) is plotted in Figure 2. The loss starts around 1.3 and steadily decreases to ~0.54 by the end of training, as depicted by the pink curve (with a dark pink smoothed trend). This smooth downward trend indicates that the model is learning effectively from the Turkish Flutter dataset without divergence. A rapid decrease in loss is observed within the first ~100 steps (to ~0.8), after which a more gradual decline is noted. By approximately step 400, the loss is seen to flatten near 0.55–0.60, suggesting that convergence has been reached. The final training loss (≈0.53) is considered quite low, given the complexity of code generation, indicating that Gemma-3 effectively fits the training data.

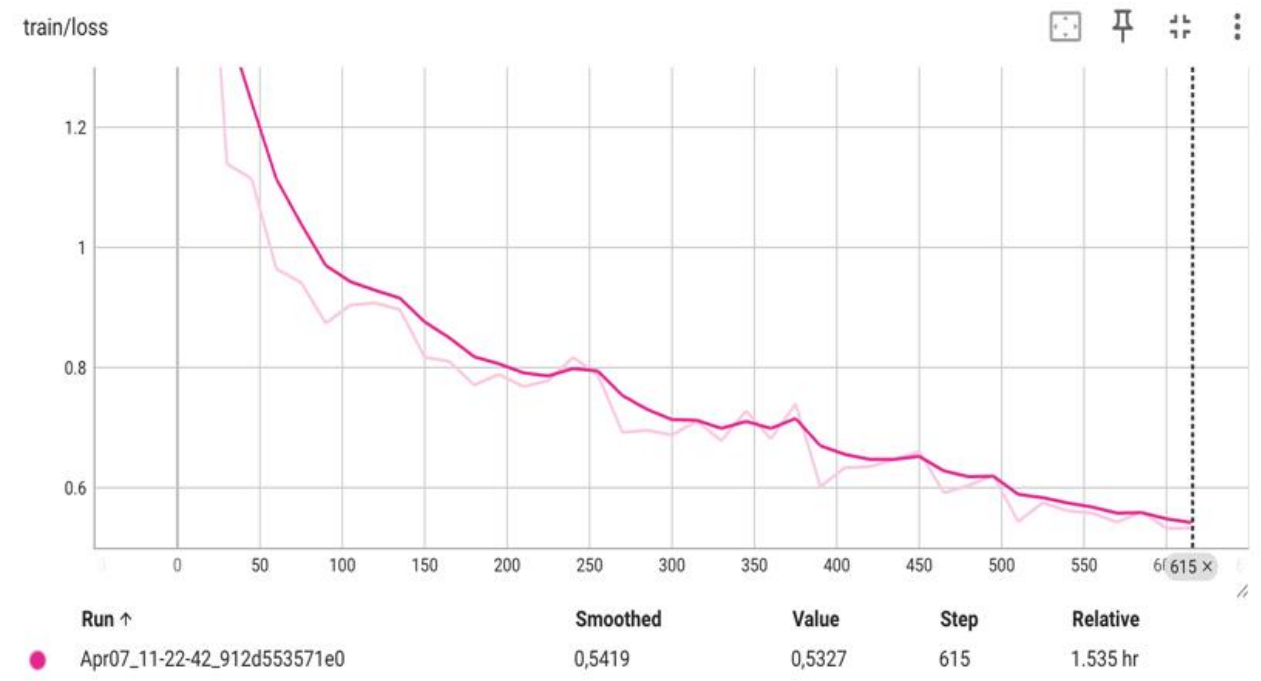

Figure 2. Gemma-3 Training Curve

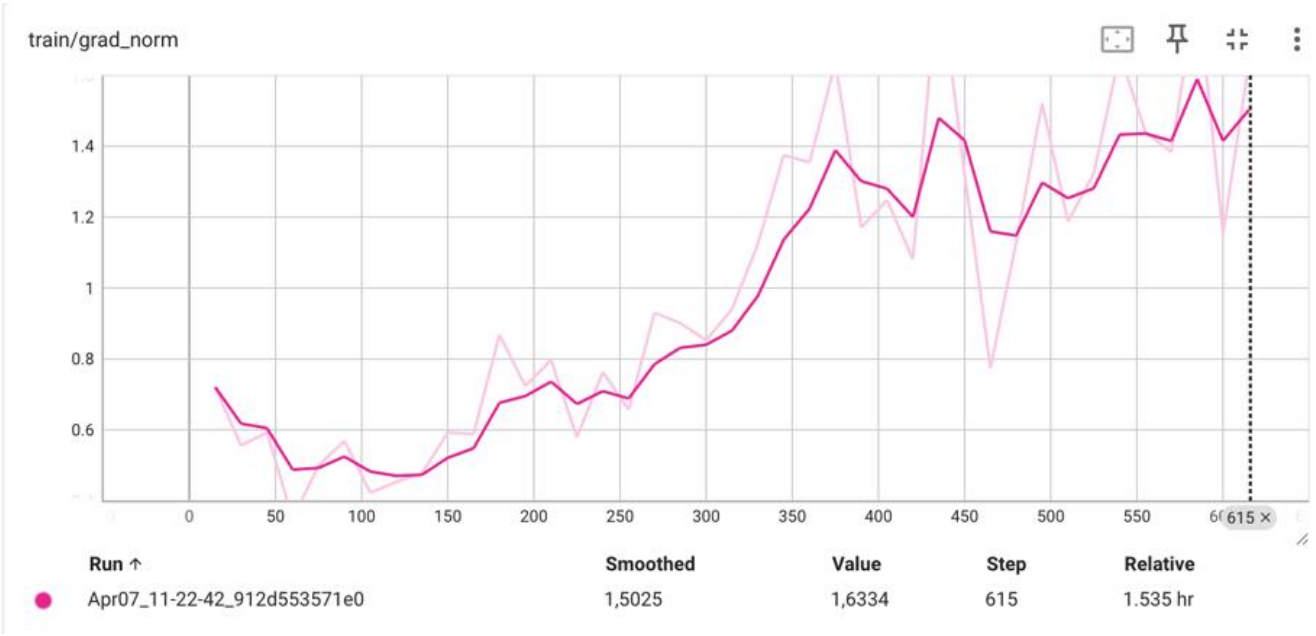

Figure 3. Gemma-3 Gradient Norms

The norm of gradients per training step for Gemma-3 fine-tuning is shown in Figure 3. Initially, gradients have moderate magnitude (~0.7), dipping below 0.5 within the first 50 steps as the optimizer finds a stable region. Throughout training, the gradient norms remain reasonable (mostly between 0.4 and 1.5). There are some oscillations – for example, a spike to ~1.6 around step 360 and a sharp dip near step 480 – but no uncontrolled growth. The smoothed trend (dark pink line) rises slightly over the epochs, ending around 1.5. This suggests that as Gemma's loss plateaued, gradients did not vanish but maintained some magnitude (perhaps learning subtle details). Importantly, no gradient explosion occurred; training remained stable. Despite the small dataset, the controlled gradient norms indicate that LoRA fine-tuning did not introduce instability into Gemma-3.

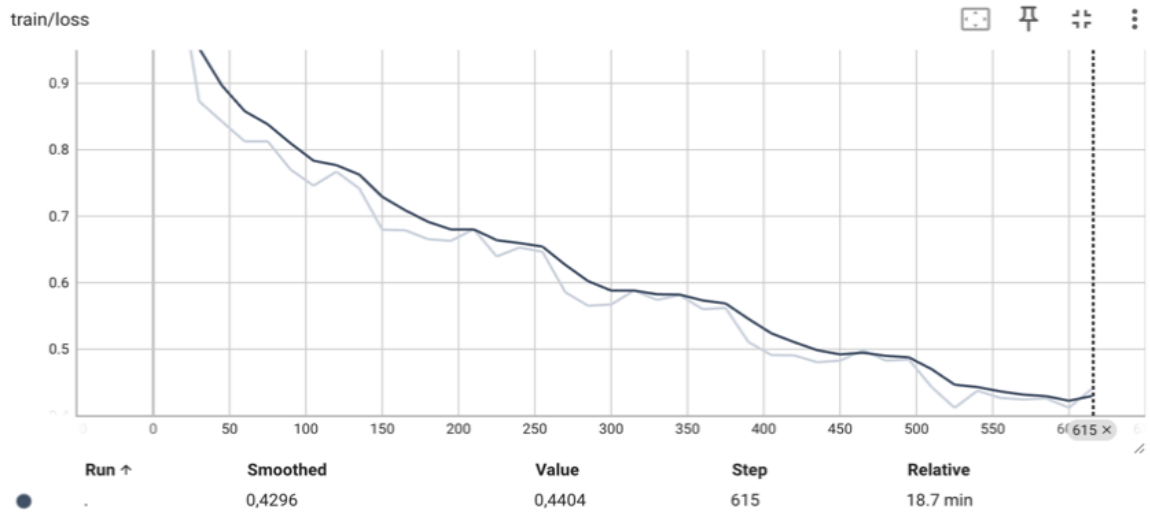

Figure 4. Qwen-2.5-Coder Training Loss Curve

The training loss for Qwen-2.5-Coder-3B over the same number of steps (615) is even lower and drops faster than Gemma's, as shown in Figure 4. Starting at ~0.9, it falls to ~0.7 within 50 steps and continues downward to ~0.44 by the end. By step ~250, the loss is already around 0.6, and it smoothly declines into the low 0.5s and high 0.4s thereafter. The final smoothed loss (~0.43) indicates that Qwen, being pre-trained on code, had less difficulty adapting to the Flutter data. Its loss curve suggests excellent generalization to the training examples, likely benefiting from prior familiarity with coding patterns. The quick convergence (under 20 minutes of training) aligns with Qwen's smaller size and code specialization. This low final loss also foreshadows strong performance in generating correct solutions.

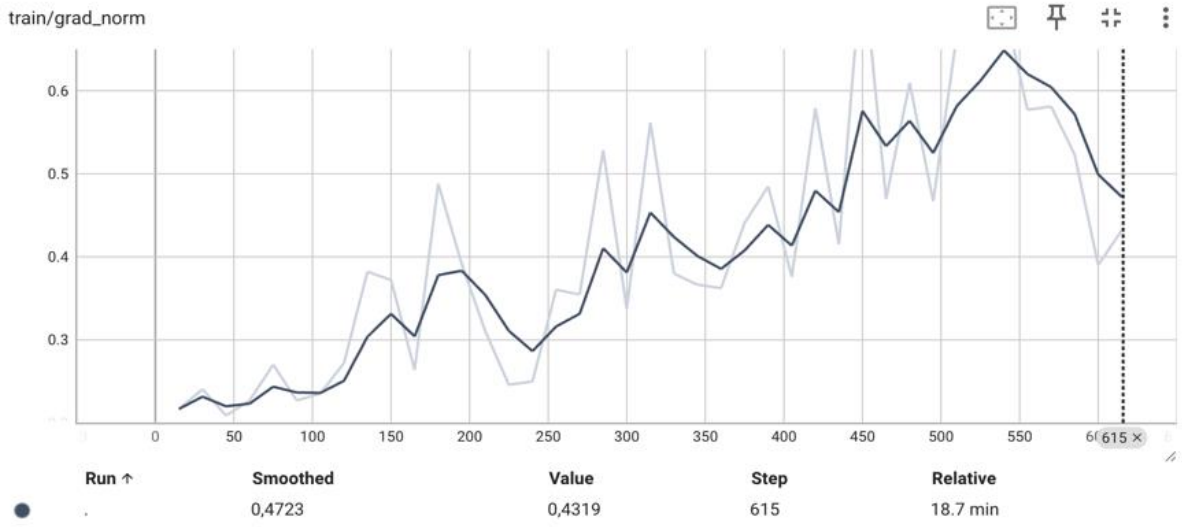

Figure 5. Qwen-2.5-Coder Gradient Norms.

Qwen-2.5-Coder Gradient Norms is illustrated in Figure 5. Gradient norms for Qwen-2.5-Coder remain quite low throughout fine-tuning. They begin around 0.25–0.30, indicating very small initial gradients, likely because the model's pre-trained weights already produce sensible outputs for some examples, given its coding knowledge. The norms fluctuate modestly between ~0.2 and ~0.6. There is a slight upward trend as training progresses (reaching ~0.5–0.6 by the end), but the values stay much lower than Gemma's. The consistently low gradient norms suggest that fine-tuning Qwen was an easier optimization problem; the model required only slight adjustments. No spikes or instabilities are observed; the training process was very smooth. This stability can be attributed to QLoRA (4-bit quantization often stabilizes training) and Qwen's strong prior on code. In summary, Qwen's gradients reflect a gentle, stable fine-tuning process that successfully hones the model's performance on the new language and domain.

3. Experimental Results

Following fine-tuning, both models were evaluated for code generation quality, inference performance, and the accuracy of answering held-out questions. Comparisons were made across five model variants: Gemma-Base (unsloth/gemma-3-4b-it-unsloth-bnb-4bit, 4B multilingual, before fine-tuning), Gemma-Finetuned (NoirZangetsu/gemma-3-flutter-finetune, 4B multilingual, after fine-tuning), Qwen-Base (unsloth/Qwen2.5-Coder-3B-Instruct-bnb-4bit, 3B code-focused, before fine-tuning), Qwen-Finetuned (NoirZangetsu/unsloth-Qwen2.5-flutter-turkish-trained, 3B code-focused, after fine-tuning), and

Qwen-General (unsloth/Qwen2.5-3B-Instruct, 3B general-purpose model without code specialization). The analysis focused on the improvements achieved through fine-tuning and the distinctions observed between the Gemma and Qwen families.

3.1 Evaluation Metrics

The output quality of the models was initially assessed using a test set of 100 Turkish Flutter questions (distinct from the training data). Code answers were generated for each model, and multiple evaluation metrics were calculated by comparing the outputs to the reference solutions. These include BLEU, ROUGE-L, METEOR, BERTScore, and CodeBLEU. These metrics are mathematically described as follows:

BLEU (Bilingual Evaluation Understudy) measures the similarity between generated text and a reference by evaluating the modified precision of n-grams. BLEU is computed as:

$$BLEU = BP \times exp(\sum_n w_n \times log(P_n)) \tag{5}$$

Where BP is the brevity penalty calculated as:

$$BP = 1 \quad if\, c > r$$
$$BP = exp(1 - r / c) \quad if\, c \leq r \tag{6}$$

Here, P_n is the n-gram precision, w_n is the weight for each n-gram (typically uniform, e.g., 1/4), c is the candidate's length, and r is the reference length.

ROUGE-L (Longest Common Subsequence) evaluates similarity based on the longest common subsequence (LCS) between the candidate and reference text. It computes:

$$P\_lcs = LCS(X, Y) / |X|$$
$$R\_lcs = LCS(X, Y) / |Y| \tag{7}$$

Then combine these into a harmonic mean:

$$F\_lcs = ((1 + \beta^2) \times P\_lcs \times R\_lcs) / (R\_lcs + \beta^2 \times P\_lcs) \tag{8}$$

β is typically set to 1 to give equal importance to precision and recall.

METEOR (Metric for Evaluation of Translation with Explicit Ordering) scores outputs based on exact matches, stemming, and synonyms, and penalizes fragmentation in word order. It calculates:

$$P\_m = |m| / |c|$$
$$R\_m = |m| / |r| \tag{9}$$
$$F\_mean = (10 \times P\_m \times R\_m) / (R\_m + 9 \times P\_m)$$

Then, it applies a penalty based on the number of chunks:

$$Penalty = \gamma \times (chunks / |m|)^{\delta}$$
$$METEOR = F\_mean \times (1 - Penalty) \tag{10}$$

Here, γ and δ are empirically tuned parameters (commonly γ = 0.5, δ = 3), |m| is the number of matched words, and “chunks” is the number of matched segments.

BERTScore (Bidirectional Encoder Representations from Transformers Score) compares a candidate and reference sentence using contextual embeddings from a transformer model. It computes:

Precision:

$$P = (1 / |x|) \times \Sigma_x \max_y \cos(E(x), E(y)) \tag{11}$$

Recall:

$$R = (1 / |y|) \times \Sigma_y \max_x \cos(E(y), E(x)) \tag{12}$$

F1 Score:

$$F1 = (2 \times P \times R) / (P + R) \tag{13}$$

here, $E(x)$ and $E(y)$ are token embeddings and $\cos(\cdot,\cdot)$ is cosine similarity.

CodeBLEU extends BLEU by incorporating code syntax and structure. It combines four components:

$$CodeBLEU = \alpha \times S\_bleu + \beta \times S\_weighted + \gamma \times S\_syntax + \delta \times S\_semantic \quad (14)$$

Where S_bleu is the standard BLEU score, S_weighted is the weighted n-gram match, S_syntax is the abstract syntax tree match, S_semantic is the semantic similarity (e.g., data flow), and weights satisfy: $\alpha + \beta + \gamma + \delta = 1$ (typically $\alpha=\beta=\gamma=\delta=0.25$).

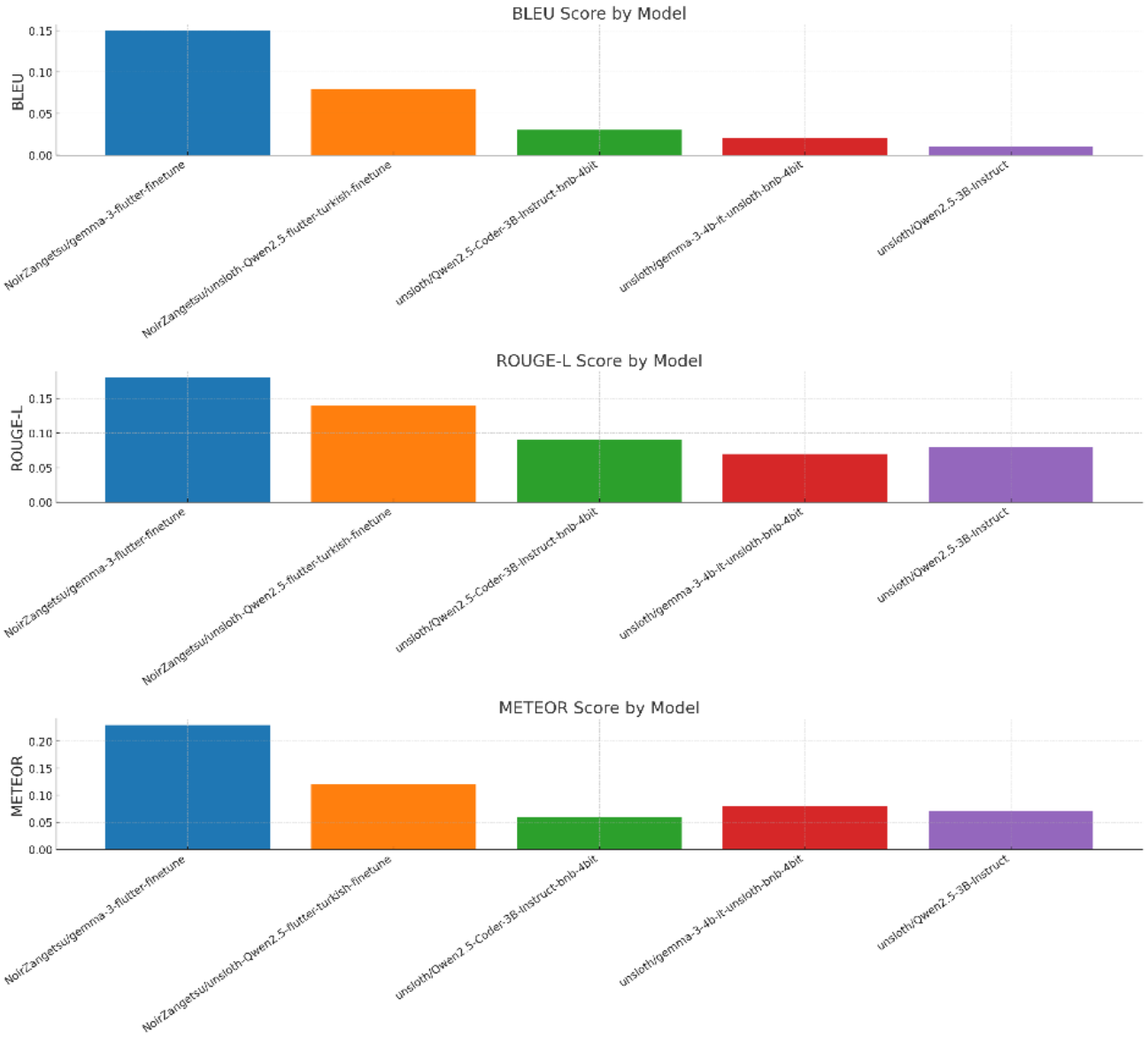

Figure 6a. Comparison of BLEU, ROUGE-L, and METEOR scores for base vs. fine-tuned models on Turkish Flutter code generation.

Table 1. Benchmark Results

Model	BERTScore	CodeBLEU	BLEU	ROUGE-L	METEOR
gemma-3-finetune	0.294	0.188	0.15	0.18	0.23
Qwen2.5-Coder-3B-Finetune	0.666	0.267	0.08	0.14	0.12
Qwen2.5-Coder-3B	0.352	0.177	0.03	0.09	0.06
gemma-3-4b	0.124	0.094	0.02	0.07	0.08
qwen2.5-3b	0.168	0.107	0.01	0.08	0.07

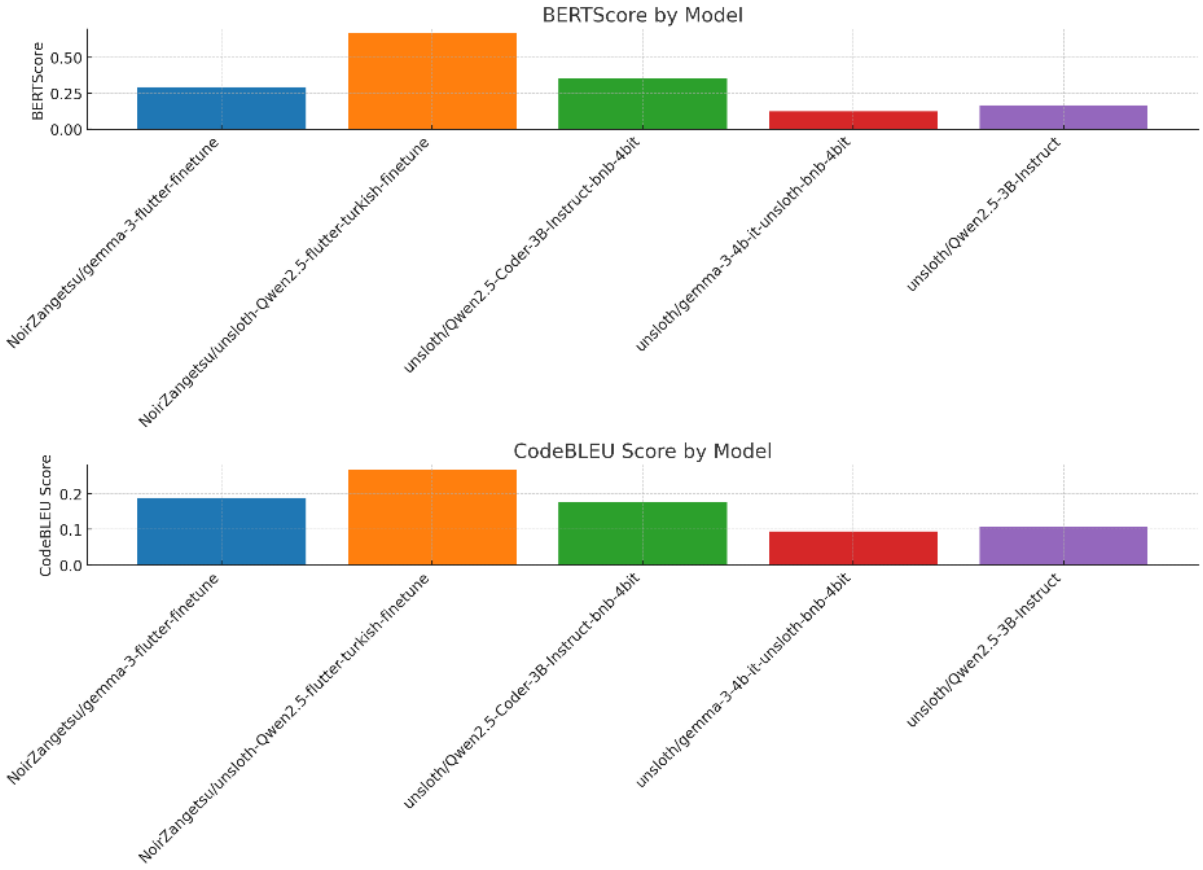

Figure 6b. Comparison of CodeBLEU and BERTScore scores for base vs. fine-tuned models on Turkish Flutter code generation.

The output quality of the models was evaluated using a held-out test set of 100 Turkish Flutter development questions, which were not included in the training data. The generated code outputs of each model were compared to reference implementations using five evaluation metrics: BLEU, ROUGE-L, METEOR, BERTScore, and CodeBLEU. These metrics capture different aspects of output quality, ranging from surface-level token overlap (BLEU, ROUGE-L, METEOR) to deeper semantic similarity (BERTScore) and structural/code-aware alignment (CodeBLEU). The results, summarized in Figure 6 and Table 1, reveal that the fine-tuned models achieved substantial gains across all metrics compared to their base counterparts. Specifically, Qwen2.5-Coder-3B-Finetune attained the highest BERTScore (0.666) and CodeBLEU (0.267), demonstrating superior semantic fidelity and syntactic correctness in code structure. Meanwhile, Gemma-3-Finetuned outperformed all other models in BLEU (0.15), ROUGE-L (0.18), and METEOR (0.23), highlighting its strength in generating fluent, token-accurate code aligned with Turkish phrasing. This divergence indicates that Qwen's code-specialized pretraining prioritizes structural and functional accuracy, whereas Gemma's multilingual capacity offers stronger alignment with natural language prompts. Notably, both models achieved significant absolute improvements: Gemma improved its BLEU from 0.02 to 0.15, and Qwen's BERTScore jumped from 0.352 to 0.666 after fine-tuning.

In contrast, the base and general-purpose models displayed limited competence in Turkish code generation. Gemma-Base, lacking exposure to code in its pretraining phase, performed the worst in all metrics, with a BLEU score of only 0.02 and a BERTScore of 0.124. Qwen2.5-Coder-3B, while specialized in code generation, still struggled with Turkish-language prompts, achieving moderate results (e.g., CodeBLEU 0.177, ROUGE-L 0.09). The general instruction-tuned model Qwen-2.5-3B-Instruct, not explicitly trained on coding tasks, showed poor alignment in both syntax and semantics (e.g., CodeBLEU 0.107, BERTScore 0.168). These findings clearly demonstrate the dual necessity of both code-centric and language-specific adaptation. Fine-tuning closed these gaps and, in some cases, reversed baseline advantages. Although Qwen-Base outperformed Gemma-Base due to its coding knowledge, Gemma-Finetuned surpassed Qwen-Finetuned on token-level metrics after adaptation. This trade-off gives developers flexibility in practical deployment scenarios: Qwen-Finetuned may be preferred for semantically robust code generation. At the same time, Gemma-Finetuned may yield more fluent and human-aligned code in Turkish. The results emphasize that fine-tuning even small-scale LLMs on limited domain- and language-specific data can yield highly capable models that rival commercial assistants, thereby bridging the accessibility gap for under-resourced developer communities.

3.2 Inference Performance

The inference speed and throughput of each model were evaluated by generating Flutter code answers. Efficient deployment was recognized as critical, particularly considering that fine-tuning can alter model size or runtime characteristics (e.g., if LoRA adapters are not merged). Two deployment scenarios were assessed: CPU inference (for Gemma, using the quantized

GGUF model in llama.cpp) and GPU inference (for Qwen and other models, utilizing 4-bit PyTorch), to simulate a range of real-world applications. Figure 7 summarizes average inference times per question, and token generation rates were recorded. On CPU, Gemma-Finetuned exhibited substantial latency, averaging approximately 210 seconds per answer (~5 tokens/s), attributed to its 4B model size and 128K context length, which impose significant overhead on CPU decoders. In contrast, Qwen-Finetuned, executed on GPU, achieved faster performance with an average of ~80 seconds per answer and ~25 tokens/s, while Qwen-Base (quantized and GPU-executed) demonstrated even higher efficiency (~60 seconds per answer, over 30 tokens/s). Gemma-Base (CPU, 4-bit) was observed to be the slowest variant, exceeding 220 seconds per answer, whereas the general 3B instruction model on GPU (Qwen-General) exhibited moderate speed (~100 seconds, 15 tokens/s). The throughput results emphasized a clear trade-off: Qwen models attained 20–32 tokens/s, while Gemma, limited by CPU execution, remained within single-digit tokens/s. Fine-tuning was found to have a negligible impact on throughput, with architectural design and runtime environment emerging as the primary determinants. Gemma's performance limitations were primarily associated with CPU deployment, and it was estimated that GPU deployment could significantly improve its token generation rate (~15–20 tokens/s). In equivalent settings (GPU 4-bit), Qwen's smaller size conferred a notable speed advantage. This revealed an important performance trade-off: Gemma-Finetuned offered higher accuracy at the cost of slower inference, whereas Qwen-Finetuned provided faster responses with slightly reduced accuracy. Deployment requirements thus dictate model selection, favoring fine-tuned Qwen for faster interactions on limited hardware or Gemma for applications demanding maximum correctness. Furthermore, it was observed that code-specialized architectures, such as Qwen's, enhanced throughput compared to general-purpose models like Gemma-Base. Although Gemma's latency could be mitigated through more aggressive quantization or model distillation, within semi-interactive assistant scenarios, its ~3.5-minute worst-case latency may remain acceptable when accuracy is paramount. In contrast, Qwen serves as a viable, faster alternative.

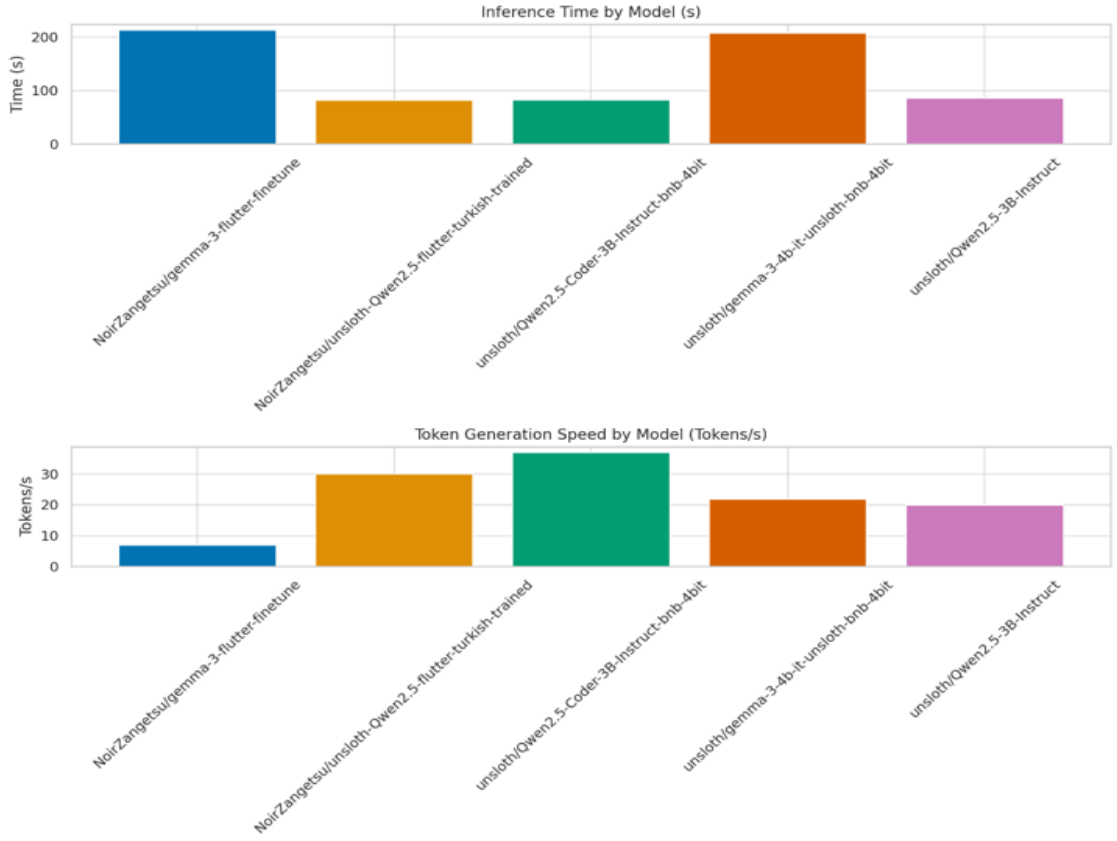

Figure 7. Inference speed comparison.

Figure 7 illustrates the comparison of inference speeds. Top: Average inference time per question (in seconds) for Gemma and Qwen models (base vs. fine-tuned). Bottom: Token generation throughput (tokens per second, higher is better). Blue – Gemma-Finetuned; orange – Qwen-Finetuned; green – Qwen-Base; red – Gemma-Base; pink – Qwen-General. Gemma-Base and Gemma-Finetuned (blue/red) exhibit the highest latencies and lowest speeds on the CPU, whereas Qwen (green/orange) is 2–4 times faster. Fine-tuning has negligible impact on speed (compare blue vs. red, orange vs. green). Running on the GPU (orange/green/pink) is faster than running on the CPU (blue/red). These results highlight the efficiency advantage of smaller, specialized models and the value of quantization for deployment.

3.3 Qualitative Evaluation

The accuracy of the models was evaluated on 10 held-out Turkish questions, which were excluded from the training set and designed to require either a single correct code-based implementation or a precise conceptual explanation. Manual annotation was employed to determine correctness, with answers considered correct if they fulfilled the intended functionality or

conveyed the appropriate explanation. As illustrated in Figure 8, the fine-tuned models significantly outperformed their base counterparts: Qwen-Finetuned correctly answered all 10 questions, while Gemma-Finetuned followed closely with 8 correct responses. In contrast, Qwen-Base (code-specialized but not fine-tuned) achieved 7 correct answers, the general-purpose Qwen-General model reached 5, and Gemma-Base answered only 3 correctly. These results align with previous metric-based evaluations and confirm that fine-tuning notably enhances model performance in practical Turkish code understanding tasks. Qwen-Finetuned's perfect score highlights the effectiveness of its concise and reference-aligned code generation, which is attributed to its strict, code-focused training. Meanwhile, Gemma-Finetuned's occasional divergence from reference answers, such as providing extended explanations or alternate valid implementations, led to slight penalties in exact match scoring, despite functional correctness. The robust zero-shot performance of Qwen-Base (7/10) suggests strong pretraining, although some failures were due to partial misunderstandings of Turkish prompts. On the other hand, Gemma-Base, lacking prior code exposure, produced mostly generic or irrelevant outputs. The improvements achieved through fine-tuning are evident: Gemma improved from 3 to 8 correct responses, and Qwen from 7 to 10. Overall, both fine-tuned models demonstrated high reliability, with Qwen-Finetuned achieving flawless performance and Gemma-Finetuned offering near-complete correctness. These findings confirm that combining code specialization with multilingual understanding via fine-tuning is essential for achieving robust performance in Turkish code generation, and that properly adapted open models can approach the effectiveness of proprietary assistants such as GitHub Copilot.

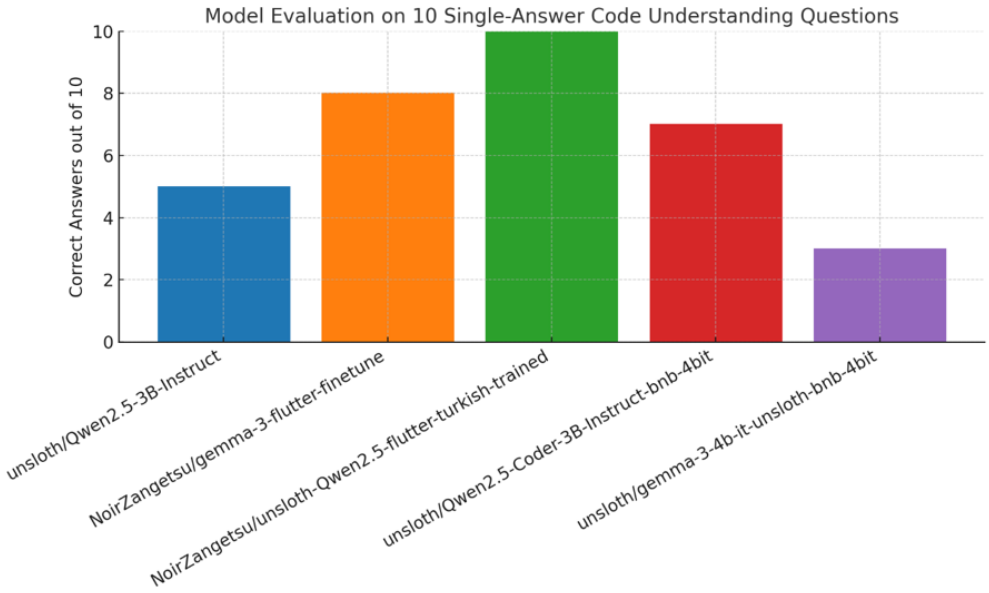

Figure 8. Evaluation on 10 held-out Turkish Flutter questions (exact match accuracy).

The bar chart shows the number of questions (out of 10) answered correctly by each model: Blue – Qwen-General with 5/10 correct; orange – Gemma-Finetuned with 8/10; green – Qwen-Finetuned with 10/10; red – Qwen-Base with 7/10; purple – Gemma-Base with 3/10. This evaluation confirms that fine-tuning enables the models to correctly interpret and solve novel Turkish queries in the Flutter domain in most cases. In contrast, the untuned models often fail or produce incomplete solutions. Each error made by the fine-tuned models was minor, and in some cases, the output was functionally correct but marked wrong only because it didn't match the expected answer format. Overall, both fine-tuned models drastically improve correctness over their base versions.

The above experimental results highlight a key point: a strong code-oriented base model (like Qwen) can greatly benefit from fine-tuning to follow non-English instructions more effectively. In contrast, a strong multilingual base model (like Gemma) requires exposure to code to be useful. Once both needs are met via fine-tuning, performance becomes excellent. The fine-tuned models can solve nearly all test queries, whereas the base models struggle with many of them. This validates that combining multilingual understanding and code knowledge (achieved via fine-tuning) is necessary for success in this Turkish code generation task.

4. Conclusions

Using the Flutter framework as the target domain, we comprehensively evaluated fine-tuned and base large language models (LLMs) for code generation in a low-resource language setting, specifically Turkish. Using a newly constructed Turkish Flutter Q&A dataset, we observed that fine-tuning significantly improved performance: BLEU, ROUGE-L, and METEOR scores increased nearly tenfold, and the correct solution rate improved from 30% to 70% (base models) to 80–90% after fine-tuning. These results underscore the crucial role of task-specific fine-tuning in maximizing a model's full potential for low-resource languages. While Qwen-2.5-Coder demonstrated strong coding ability, it struggled to interpret Turkish prompts. On the other hand, Gemma-3 demonstrated excellent Turkish comprehension but lacked coding proficiency. Only after fine-tuning did both models perform Turkish-to-Dart generation effectively.

We also found that code-oriented pretraining offers a considerable advantage. Before fine-tuning, Qwen-Base consistently outperformed Gemma-Base, underscoring the value of code exposure during pretraining [20],[21]. However, after fine-tuning, Gemma-3 slightly surpassed Qwen, showing that language-rich models can excel in code generation when adapted properly. The fine-tuned Qwen may be preferable for latency-sensitive applications due to faster inference. In contrast, the fine-tuned Gemma model may be ideal when correctness is prioritized, and sufficient computational resources are available.

We demonstrated the feasibility of achieving high code generation performance under low-resource conditions. Turkish, nearly absent from earlier code datasets, served as our target language. With only ~1.5K examples, we successfully adapted generic models into effective Turkish Flutter assistants. These results validate modern adaptation techniques, such as LoRA and QLoRA, which enable high specialization with minimal data and computational requirements. They also reinforce multilingual code generation literature findings, confirming that LLMs can transfer coding abilities across languages [22],[23]. For Turkish Flutter developers, and more broadly, developers using other under-resourced languages, our approach shows that AI-based code generation can be as effective as it is for English-speaking developers.

Despite these promising outcomes, we recognize several limitations. Our dataset, although carefully curated, is small (~1.5K Q&A pairs) and narrowly focused on the Flutter/Dart domain, which limits its generalizability to other frameworks or languages. Additionally, while most Turkish prompts were manually reviewed, some were machine-generated and may carry linguistic noise or bias. Our evaluation focused on short-code completions and did not include end-to-end app generation, which limited its real-world assessment. Moreover, inference benchmarking used different hardware (CPU for Gemma vs. GPU for Qwen), which may affect speed comparisons. Future work should address these by expanding the dataset, adopting multilingual prompts, and introducing execution-based evaluations and standardized benchmarks.

We plan to expand the dataset in terms of both size and diversity. We also aim to fine-tune smaller base models to support deployment in mobile and resource-constrained environments. Knowledge distillation, which transfers knowledge from a large model, such as Gemma or Qwen, into a 1B or 2B student model, may help maintain accuracy while improving efficiency. Additionally, execution-based training (e.g., reinforcement learning from test case feedback) could improve real-world performance. We also propose experimenting with dual-language prompts (Turkish + English) to explore the benefits of multilingual multitask learning. Lastly, we plan to assess our models on broader, open-ended Flutter tasks, such as full app generation, to evaluate their generalization capabilities.

Our findings show that fine-tuned LLMs can achieve high-quality code generation even in low-resource domains and languages. We showed that fine-tuning bridges the gap between underrepresented languages and effective AI coding support. This work demonstrates the potential to democratize AI-assisted coding tools for non-English-speaking developers, making advanced software generation technology more accessible and inclusive globally. As LLMs evolve, we anticipate that multilingual and domain-specific fine-tuning will play a central role in expanding access to AI-powered software development. With the right techniques, even small datasets can deliver meaningful results.

References

[1] J. He, C. Zhou, X. Ma, T. Berg-Kirkpatrick, & G. Neubig, "Towards a unified view of parameter-efficient transfer learning", 2021. doi: 10.48550/arxiv.2110.04366

[2] N. Houlsby, A. Giurgiu, S. Jastrzębski, B. Morrone, Q. Laroussilhe, A. Gesmundoet al., "Parameter-efficient transfer learning for nlp", 2019. doi: 10.48550/arxiv.1902.00751

[3] X. Liu, P. He, W. Chen, & J. Gao, "Multi-task deep neural networks for natural language understanding", 2019. doi: 10.18653/v1/p19-1441

[4] M. Anschütz, D. Lozano, & G. Groh, "This is not correct! negation-aware evaluation of language generation systems", 2023. doi: 10.18653/v1/2023.inlg-main.12

[5] Lodha, G. Belapurkar, S. Chalkapurkar, Y. Tao, R. Ghosh, S. Basuet al., "On surgical fine-tuning for language encoders", 2023. doi: 10.18653/v1/2023.findings-emnlp.204

[6] J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wanget al., "Lora: low-rank adaptation of large language models", 2021. doi: 10.48550/arxiv.2106.09685

[7] Y. Hu, Y. Xie, T. Wang, M. Chen, & Z. Pan, "Structure-aware low-rank adaptation for parameter-efficient fine-tuning", Mathematics, vol. 11, no. 20, p. 4317, 2023. doi: 10.3390/math11204317

[8] N. Dhinagar, S. Ozarkar, K. Buwa, S. Thomopoulos, C. Owens-Walton, E. Laltooet al., "Parameter efficient fine-tuning of transformer-based masked autoencoder enhances resource constrained neuroimage analysis", 2025. doi: 10.1101/2025.02.15.638442

[9] H. Wu, "Large language models capsule: a research analysis of in-context learning (icl) and parameter-efficient fine-tuning (peft) methods", Applied and Computational Engineering, vol. 43, no. 1, pp. 327-331, 2024. doi: 10.54254/2755-2721/43/20230858

[10] N. Sulaiman and F. Hamzah, "Optimizing llama 7b for medical question answering: a study on fine-tuning strategies and performance on the multimedqa dataset", 2024. doi: 10.31219/osf.io/g5aes

[11] J. Bogaert, E. Jean, C. Bodt, & F. Standaert, "Fine-tuning is not (always) overfitting artifacts", 2023. doi:

10.14428/esann/2023.es2023-152
[12] G. Wiedemann, S. Yimam, & C. Biemann, "Uhh-lt at semeval-2020 task 12: fine-tuning of pre-trained transformer networks for offensive language detection", pp. 1638-1644, 2020. doi: 10.18653/v1/2020.semeval-1.213
[13] Aghajanyan, S. Gupta, & L. Zettlemoyer, "Intrinsic dimensionality explains the effectiveness of language model fine-tuning", 2021. doi: 10.18653/v1/2021.acl-long.568
[14] L. Feng, Y. Yang, M. Tan, T. Zeng, Z. Li, H. Tanget al., "Adaptive multi-source domain collaborative fine-tuning for transfer learning", 2023. doi: 10.20944/preprints202311.0124.v1
[15] F. Ullah, U. Azam, A. Faheem, F. Kamiran, & A. Karim, "Comparing prompt-based and standard fine-tuning for urdu text classification", pp. 6747-6754, 2023. doi: 10.18653/v1/2023.findings-emnlp.449
[16] M. Mosbach, M. Andriushchenko, & D. Klakow, "On the stability of fine-tuning bert: misconceptions, explanations, and strong baselines", 2020. doi: 10.48550/arxiv.2006.04884
[17] X. Li and P. Liang, "Prefix-tuning: optimizing continuous prompts for generation", 2021. doi: 10.18653/v1/2021.acl-long.353
[18] X. Ma, C. Santos, & A. Arnold, "Contrastive fine-tuning improves robustness for neural rankers", 2021. doi: 10.18653/v1/2021.findings-acl.51
[19] L. Pan, C. Hang, A. Sil, & S. Potdar, "Improved text classification via contrastive adversarial training", 2021. doi: 10.48550/arxiv.2107.10137
[20] Chen M., Tworek J., Jun H., Kaplan J., Yuan Q. and Zarinelli E., “Evaluating Large Language Models Trained on Code”, arXiv preprint arXiv:2107.03374, (2021). doi: 10.48550/arXiv.2107.03374
[21] Xu X., Sharma P., Kinne J. F., O’Neill M., Mazaitis K. and Bhatia S., “A Systematic Evaluation of Large Language Models of Code”, Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 662-678, (2022). doi: 10.48550/arXiv.2202.13169
[22] Wang Z., Cuenca G., Zhou S., Chen T., Lin B. and Matsuo Y., “MCoNaLa: A Benchmark for Code Generation from Multiple Natural Languages”, Findings of the Association for Computational Linguistics: EACL 2023, 265-273, (2023). doi: 10.48550/arXiv.2203.08388
[23] Cassano F., Gouwar J., Nguyen D., Bartolo M., Serrano S. and Sabour A., “MultiPL-E: A Scalable and Extensible Approach to Benchmarking Neural Code Generation”, arXiv preprint arXiv:2208.08227, (2022). doi: 10.48550/arXiv.2208.08227

Article Information Form

Authors Contributions

This study was conducted by Buğra Alperen Uluırmak**,** who was responsible for designing the study, collecting the data, analyzing the data, implementing the experiment, and drafting the manuscript. Rifat Kurban provided supervision, guidance, and consultancy throughout the study, contributing to the refinement of the manuscript and its content improvement.

Conflict of Interest Notice

In this section, the statements of the authors regarding the conflict of interest will be included. The authors declare that there is no conflict of interest regarding the publication of this paper.

Ethical Approval

It is declared that this study was prepared in accordance with scientific and ethical principles. All referenced studies have been appropriately cited in the bibliography. No ethics committee approval was required as this study did not involve human or animal subjects.

Availability of data and material

The datasets used in this study are publicly available on GitHub:

Turkish Dataset: https://github.com/NoirZangetsu/Flutter-Code-with-Questions-Dataset-Turkish

English Dataset: https://github.com/NoirZangetsu/Flutter-Code-with-Questions-Dataset-English-

Each repository contains curated question–answer pairs in Flutter/Dart, formatted for model training and evaluation. The training scripts, evaluation tools, and instructions for reproduction are also included to support future research in multilingual code generation.

Artificial Intelligence Statement

Artificial intelligence tools (e.g., ChatGPT) were employed during the manuscript preparation solely for proofreading, grammatical corrections, and linguistic refinement purposes. The content, analysis, experimental results, and conclusions presented in this article are entirely original and reflect the intellectual contributions of the listed authors.

Plagiarism Statement

This article has been scanned by iThenticate ™.