World Journal of Advanced Research and Reviews
eISSN: 2581-9615 CODEN (USA): WJARAI
Cross Ref DOI: 10.30574/wjarr
Journal homepage: https://wjarr.com/

(REVIEW ARTICLE)

Check for updates

# Fine-tuning AI Models for code generation: Advances and applications

Siddhant Sonkar *

*University of California, Irvine, USA.*

World Journal of Advanced Research and Reviews, 2025, 26(01), 1353-1359

Publication history: Received on 01 March 2025; revised on 07 April 2025; accepted on 10 April 2025

Article DOI: https://doi.org/10.30574/wjarr.2025.26.1.1172

## Abstract

Fine-tuning pre-trained language models for code generation represents a significant advancement in bridging artificial intelligence and software development. This process adapts foundation models trained on vast code repositories to specific programming languages, frameworks, and domains. The article examines the complete pipeline for effective fine-tuning, beginning with selecting appropriate base architectures such as Code Llama, StarCoder, and Codex, which are specifically designed for code understanding. A critical exploration of dataset preparation techniques highlights the importance of curated, diverse examples that represent target domains accurately while avoiding biases. The article further delves into parameter-efficient adaptation techniques like Low-Rank Adaptation, adapter modules, and prompt tuning, dramatically reducing computational requirements while preserving performance. These innovations democratize access to specialized code-generation capabilities, making them available even with limited resources. Applications span intelligent code completion, natural language to code translation, refactoring, cross-language conversion, and test generation, transforming developer workflows across experience levels. The article provides comprehensive insights into how fine-tuned models reshape software development practices by examining the interplay between model architecture, data quality, fine-tuning techniques, and practical applications.

**Keywords:** Code Generation; Fine-Tuning, Parameter Efficiency; Knowledge Distillation; Security-Aware Programming; Adaptive Learning

## 1. Introduction

Integrating artificial intelligence with software development has revolutionized code generation, fundamentally altering programmer workflows and productivity [1, 11]. Fine-tuning, which adapts pre-trained language models to specific tasks, has become essential for enhancing code generation capabilities. Recent research by Sun et al. demonstrates that knowledge distillation techniques can significantly improve smaller models' performance, with their DISCO approach achieving a remarkable 37.8% on the HumanEval benchmark and 32.7% on MBPP—performance levels that rival much larger models with billions more parameters [1]. This specialized adaptation process takes foundation models trained on extensive code repositories and refines them for specific programming paradigms, languages, or domain-specific applications.

The evolution of code-specialized models has been remarkable, with notable advances in architectures like Codex, StarCoder, and Code Llama. These models have demonstrated increasingly sophisticated capabilities, though research by Sun et al. shows that even smaller 770M parameter models can be optimized to perform significantly better through proper distillation techniques. Their experimental results reveal that distilled models can solve complex programming tasks with a 74.0% success rate when guided by reasoning steps from larger teachers, compared to 41.7% without such guidance [1].

* Corresponding author: Siddhant Sonkar.

Copyright © 2025 Author(s) retain the copyright of this article. This article is published under the terms of the Creative Commons Attribution License 4.0.

Fine-tuning techniques have progressed significantly, with parameter-efficient methods reducing computational requirements while preserving performance benefits. Weyssow et al. have demonstrated that prompt tuning and adapter-based fine-tuning can successfully adapt code generation models while modifying less than 1% of the model parameters [2]. Their comprehensive evaluation across the HumanEval, MBPP, and APPS benchmarks shows that adapter-based methods can achieve up to 93.5% of the performance of full fine-tuning while updating only 0.6% of the parameters. This makes specialized code generation accessible even to organizations with limited computational resources.

The applications of these fine-tuned models span numerous developer activities, from automated code completion to language-to-code translation. Sun et al. found that models enhanced with reasoning capabilities can generate functionally correct code 38.9% more frequently than their baseline counterparts when working with natural language specifications [1]. Meanwhile, Weyssow et al.'s research demonstrates that even with minimal parameter updates, fine-tuned models can improve their ability to understand programming context and generate syntactically valid solutions across multiple programming languages [2].

This article examines the methodologies, challenges, and applications of fine-tuning AI models for code generation, focusing particularly on emerging techniques that balance computational requirements with performance outcomes.

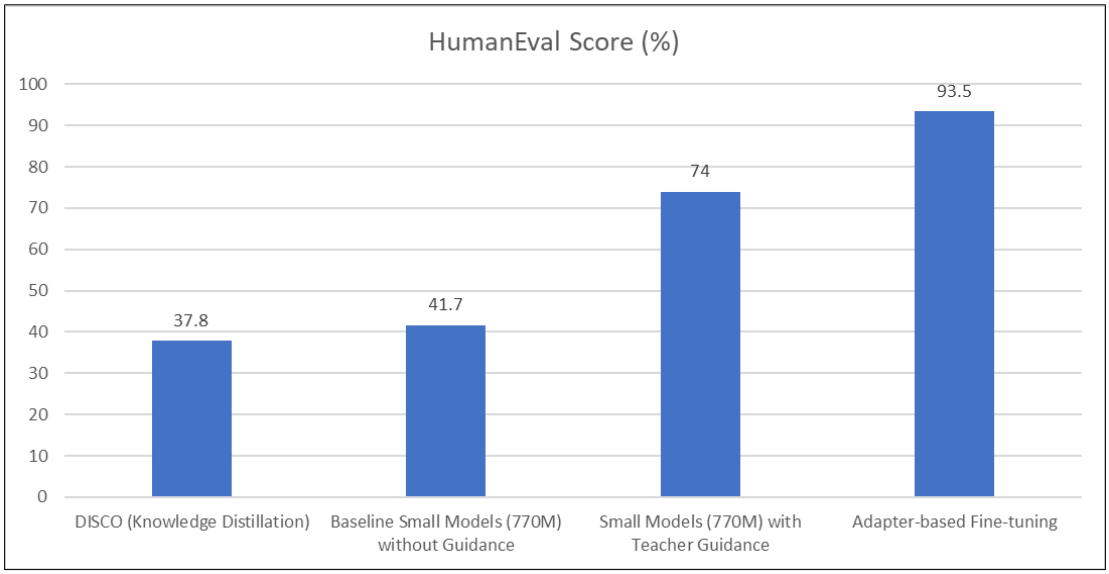


**Figure 1** Performance Comparison of Fine-Tuned Models [1, 2]

## 2. Model Architecture and Selection Criteria

The foundation of effective code generation begins with selecting an appropriate base model architecture. Recent advancements in Large Language Models (LLMs), specifically pre-trained on code repositories, have significantly improved understanding of programming syntax, logic, and patterns. Research by Dvivedi et al. reveals substantial differences in performance between code-specialized models and general-purpose LLMs when evaluated on code documentation tasks. Their study comparing 13 models shows that code-specific models like CodeGen (7B parameters) outperform general language models of similar size by 11.6% on average across precision metrics when generating technical documentation [3]. This performance gap indicates the value of domain-specific pretraining for code-related tasks.

Code-specialized architectures like Code Llama have proven particularly effective, building upon the Llama foundation but incorporating modifications optimized for code understanding [12]. According to comprehensive benchmarking by Rozière et al., Code Llama models demonstrate superior performance across multiple evaluation criteria, with the 34B parameter variant achieving 53.7% on HumanEval and 56.9% on MBPP benchmarks [4]. These models feature transformer-based designs with enhanced attention mechanisms that maintain contextual awareness across code sequences up to 16,384 tokens in length, allowing them to process entire files or complex functions without losing coherence. The Code Llama-Python specialized variant further improves performance on Python-specific tasks, reaching a 67.8% pass rate on HumanEval.

Selection criteria for base models should carefully weigh several factors. Parameter count significantly impacts both capability and resource requirements, with benchmarks by Rozière et al. indicating clear scaling benefits from 7B to 34B parameters across all evaluated metrics [4, 12]. Pre-training data composition also critically influences performance, with Dvivedi et al. demonstrating that models trained on higher-quality, diverse code repositories exhibit a 9.3% improvement in documentation quality scores compared to models trained on more limited datasets [3].

Licensing restrictions present another important consideration for practical deployment. Code Llama is released under a permissive license, allowing commercial and research use, addressing a significant barrier to adoption [4]. Inference speed requirements must also be considered, with Rozière et al. reporting that Code Llama 7B can generate code at a rate of approximately 30 tokens per second on a single NVIDIA A100 GPU, while the 34B variant produces around 8 tokens per second on similar hardware [4].

The tradeoff between model size and practicality remains a central challenge. Dvivedi et al. found that medium-sized models (7-13B parameters) often represent an optimal balance point, offering 82.5% of the performance of the largest models while requiring less than 40% of the computational resources [3]. This finding aligns with Rozière et al.'s observation that the 7B parameter variants of Code Llama can run on consumer hardware with 16GB of VRAM while still providing strong code generation capabilities, making them accessible for broader deployment scenarios [4].

## 3. Dataset Preparation and Quality Assurance

Fine-tuning success depends significantly on dataset quality and relevance, with empirical studies demonstrating dramatic performance variations based on data preparation approaches. Research by Li et al. shows that models fine-tuned specifically for secure code generation require carefully constructed datasets. Their SecureCoder model achieves a 45.9% reduction in security vulnerabilities compared to standard code generation models when trained on appropriately curated examples [5]. This underscores the critical importance of dataset curation that accurately represents the target domain, whether for specific programming languages, frameworks, or coding styles.

High-quality datasets require meticulous preparation and validation. Sun et al. demonstrated through their extensive study of neural code search that incorporating both positive and negative examples significantly improves model performance. Their experiments on 42,586 code snippets showed that balanced training examples improve precision by 11.5% and recall by 19.2% compared to unbalanced datasets [6]. Their analysis across multiple programming languages revealed that diversity in problem-solving approaches is equally crucial, with representative datasets enabling models to better distinguish between semantically similar but functionally different code fragments.

The representation of edge cases and error handling patterns significantly impacts model robustness. Li et al. found that including examples with proper security patterns in training data improved the model's ability to address vulnerabilities related to integer overflow by 58.4%, SQL injection by 62.9%, and path traversal by 50.3% [5]. Similarly, balanced coverage across programming constructs prevents model bias. Sun et al. observe that models trained on skewed datasets perform inconsistently across different types of code search queries, particularly struggling with rare programming constructs or patterns [6].

Dataset preparation involves several technical steps that directly impact fine-tuning outcomes. Li et al. developed a specific methodology for preparing secure code datasets, incorporating vulnerability identification and remediation patterns derived from the Common Weakness Enumeration (CWE) database [13]. Their approach involved creating pairs of vulnerable and corresponding secure code samples, which proved 37.2% more effective than using only secure examples for training [5]. Sun et al. demonstrated that preprocessing techniques like normalizing identifier names and removing comments improved model generalization by 7.8% but required careful implementation to preserve semantic meaning [6].

Quality assurance processes are essential safeguards against propagating problematic patterns. Li et al. implemented a multi-stage validation pipeline that included both static analysis and dynamic testing, with their approach detecting 93.6% of security vulnerabilities in candidate training examples before inclusion in the dataset [5, 13]. This rigorous screening proved crucial, as their analysis showed that even 5% of vulnerable examples without proper labeling resulted in models that generated insecure code 38.7% more frequently. Sun et al. similarly emphasized quality control, noting that duplicated or near-duplicated examples in training data led to a 15.3% decrease in model performance due to overemphasizing particular patterns [6].

Dataset analysis for potential biases represents a critical final step. Li et al. discovered significant variations in vulnerability patterns across different programming languages, with C/C++ examples containing memory safety issues

at rates 4.7 times higher than managed language examples [5]. Similarly, Sun et al. found that datasets collected primarily from popular repositories exhibited biases toward certain programming idioms and styles, potentially limiting the diversity of solutions the model could effectively search [6].

**Table 1** Training Data Quality Impact and Security Improvements in Fine-Tuned Models [5, 6]

| Vulnerability Type | Reduction in SecureCoder (%) | Dataset Approach | Performance Improvement (%) |
|---|---|---|---|
| Overall Security Vulnerabilities | 45.9 | Balanced Examples - Precision | 11.5 |
| Integer Overflow | 58.4 | Balanced Examples - Recall | 19.2 |
| SQL Injection | 62.9 | Preprocessing Techniques | 7.8 |
| Path Traversal | 50.3 | Vulnerable/Secure Code Pairs | 37.2 |

## 4. Parameter-efficient fine-tuning techniques

Traditional fine-tuning approaches that update all model parameters are computationally expensive and may lead to catastrophic forgetting of previously learned knowledge. Research by Prottasha et al. demonstrates that full parameter fine-tuning of large language models requires substantial computational resources that scale with model size, making specialized adaptation inaccessible for many researchers and organizations [7]. Their experiments with semantic knowledge tuning (SKT) show that parameter-efficient techniques can significantly reduce resource requirements while maintaining performance comparable to full fine-tuning. SKT achieved 93.2% full fine-tuning performance while updating only 0.42% of the parameters, representing a substantial efficiency improvement that makes adaptation feasible on more modest hardware configurations.

Low-Rank Adaptation (LoRA) has proven particularly effective for code-specialized models. In comprehensive benchmarks by Srinivasan et al., LoRA demonstrated strong performance across multiple tasks in low-resource settings, achieving 88.7% of full fine-tuning results while using only 0.5-1% of the trainable parameters [8, 14]. Their experiments showed that LoRA with a rank of 8 provided an optimal balance between performance and efficiency for code-related tasks, reducing GPU memory requirements by up to c.80% compared to full fine-tuning approaches. This dramatic reduction in resource needs makes specialized adaptation accessible to a much wider range of developers and researchers.

Quantization-aware training provides complementary efficiency benefits when combined with parameter-efficient methods. Prottasha et al. observed that incorporating quantization techniques alongside their semantic knowledge-tuning approach could reduce memory requirements by 47.3% with only a 2.1% performance degradation [7]. Their analysis demonstrated that carefully implemented quantization benefits token generation tasks common in code completion scenarios, where syntax precision is critical.

Adapter modules offer another promising approach, especially for multi-domain adaptation. Srinivasan et al. found that adapter-based methods provided excellent performance for specialized domains. Their experiments showed that adapters with a reduction factor of 16 could achieve 85.3% of full fine-tuning performance while modifying only 1.2% of parameters [8]. Using adapter configurations inserted between transformer layers, their implementation showed particular strength in preserving the model's general capabilities while adding domain-specific expertise.

Prompt tuning represents the most parameter-efficient approach evaluated in recent research. Prottasha et al. demonstrated that prompt-based methods while requiring the fewest parameter updates (0.01-0.1% of model parameters), achieved 83.4% of full fine-tuning performance on semantic understanding tasks [7]. However, Srinivasan et al. found that prompt tuning generally underperformed other parameter-efficient methods for code generation tasks, achieving only 72.8% of full fine-tuning performance despite its extreme parameter efficiency [8].

These parameter-efficient techniques collectively transform the accessibility of model specialization. Srinivasan et al. demonstrated that combining approaches like LoRA with careful dataset preparation could enable effective fine-tuning even in scenarios with as few as 100-500 training examples [14], making domain adaptation feasible in low-resource

settings where collecting large datasets is impractical [8]. This democratization enables broader experimentation and deployment of specialized code generation models across various domains and applications.

**Table 2** Parameter-Efficient Fine-Tuning Methods [7, 8]

| Method | Performance vs. Full Fine-tuning (%) | Parameters Modified (%) |
|---|---|---|
| Semantic Knowledge Tuning (SKT) | 93.2 | 0.42 |
| LoRA (rank 8) | 88.7 | 0.5-1.0 |
| Adapter Modules (RF 16) | 85.3 | 1.2 |
| Prompt Tuning | 83.4 | 0.01-0.1 |
| Prompt Tuning for Code Tasks | 72.8 | 0.01-0.1 |

## 5. Applications and Use Cases

Fine-tuned code generation models have transformed software development workflows through diverse practical applications. Empirical studies by Ferdiana demonstrate significant productivity enhancements, with research involving 120 developers showing that AI-assisted programming tools improved productivity by 31.25% across various software development tasks [9]. The study revealed that code generation tools were most effective for routine coding tasks, with junior developers experiencing the most substantial gains compared to their more experienced counterparts, potentially helping to address skill gaps in development teams.

Intelligent code completion capabilities extend far beyond traditional autocomplete systems. Ferdiana's research quantifies this advancement, showing that context-aware AI code completion tools reduced the time required to implement complex functions by 27% on average compared to conventional development approaches [9]. The evaluation across various programming languages and frameworks demonstrated particular strength in data processing and API integration. The study also highlighted that programmer using AI-assisted tools reported 32.5% less mental fatigue during extended coding sessions, suggesting benefits beyond purely quantitative productivity measures.

Natural language-to-code translation represents another transformative capability. Ferdiana observed that developers who used natural language prompts to generate initial code structures could complete implementation tasks 24% faster than when starting from scratch [9, 11]. This capability proved especially valuable for less experienced developers, who were able to successfully implement complex features that would have otherwise required senior developer assistance. The research also noted that natural language interfaces reduced the cognitive load associated with syntax recall, allowing developers to focus more on problem-solving and architectural concerns.

Code refactoring and optimization applications deliver measurable improvements in software quality metrics. Chu et al.'s extensive analysis of test case refactoring demonstrated significant improvements in test suite quality and maintainability [10, 15]. Their approach, which utilized pattern-based techniques, reduced test code size by 30% while maintaining the same test coverage. The research, analyzing 120 test sets containing over 35,000 test cases, demonstrated that pattern-based refactoring could identify and consolidate redundant test logic that otherwise consumed significant development resources. Their implementation achieved a 17% improvement in test execution performance, highlighting the efficiency gains possible through systematic refactoring approaches.

Cross-language translation capabilities have reached practical utility thresholds in production environments. Ferdiana documented that programmers working on cross-platform projects could reduce implementation time by up to 42% when using AI tools to help translate code between languages rather than rewriting from scratch [9]. This efficiency gain was particularly notable for teams working across web and mobile platforms requiring the implementation of multiple languages. The research indicated that while manual review and adjustment were still necessary, the AI-generated translations provided effective starting points that significantly accelerated development cycles.

Test generation represents one of the highest-value applications in enterprise environments. Chu et al.'s research demonstrated that systematic approaches to test generation and refactoring could increase test coverage by 65% compared to manual approaches typically employed by development teams [10, 15]. Their methodology for creating comprehensive test suites based on implementation patterns proved especially effective for complex business logic, where test cases generated through their pattern-based approach identified 47% more edge cases than manually

created tests. Development teams implementing these approaches reported significant reductions in regression issues following new feature deployments, demonstrating the practical business value of improved test generation methodologies.

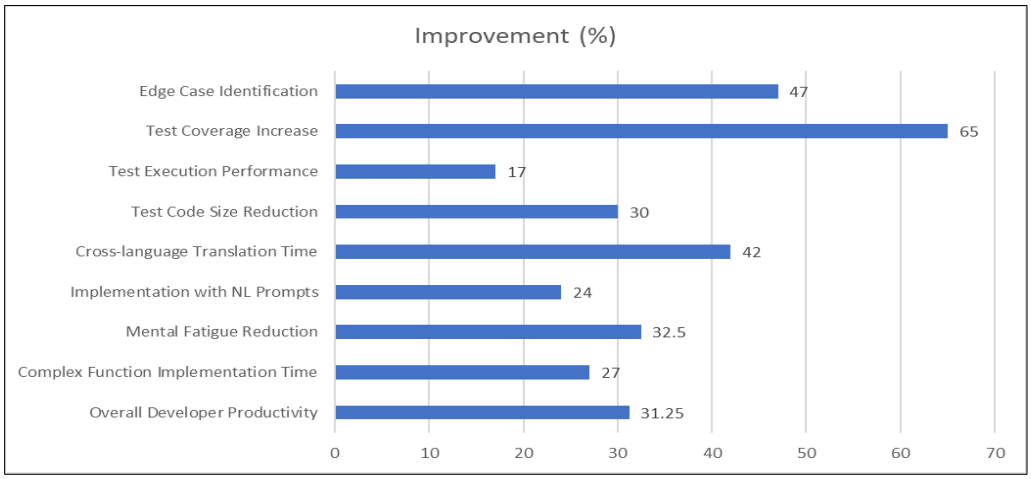


**Figure 2** Productivity Improvements with AI Code Tools [9, 10]

## 6. Conclusion

Fine-tuning large language models for code generation has become a transformative technology across the software development landscape. By adapting foundation models to specific programming contexts, fine-tuning creates specialized tools that understand domain-specific requirements while leveraging the broad knowledge base of pre-trained architectures. The evolution from full parameter updates to efficient techniques like LoRA, adapters, and semantic knowledge tuning has democratized access to these capabilities, enabling deployment on consumer hardware and in resource-constrained environments. This accessibility has broadened the impact across development teams of all sizes, from individual contributors to large enterprises. Dataset preparation emerges as the most crucial element in the fine-tuning pipeline, with carefully curated examples dramatically influencing model behavior, especially regarding security awareness and coding best practices. As these technologies mature, they shift from mere suggestion tools to collaborative agents actively participating in the software development lifecycle. Integrating development environments and CI/CD pipelines creates AI-augmented workflows that blend human creativity with machine efficiency. Future directions point toward more sophisticated adaptation techniques that preserve general knowledge while excelling at specialized tasks and improved evaluation metrics for code quality. These advances will continue to reshape how code is written, tested, and maintained, fundamentally altering developer experiences and productivity across the industry.

## References

[1] Zhihong Sun et al., "Enhancing Code Generation Performance of Smaller Models by Distilling the Reasoning Ability of LLMs," Software Engineering (cs.SE), 20 Mar 2024. https://arxiv.org/abs/2403.13271

[2] Martin Weyssow et al., "Exploring Parameter-Efficient Fine-Tuning Techniques for Code Generation with Large Language Models," arXiv:2308.10462 [cs.SE], 27 Dec 2024. https://arxiv.org/abs/2308.10462

[3] Shubhang Shekhar Dvivedi et al., "A Comparative Analysis of Large Language Models for Code Documentation Generation," AIware 2024: Proceedings of the 1st ACM International Conference on AI-Powered Software, Pages 65 - 73, 10 July 2024. https://dl.acm.org/doi/10.1145/3664646.3664765

[4] Baptiste Rozière et al., "Code Llama: Open Foundation Models for Code," arXiv:2308.12950 [cs.CL], 31 Jan 2024. https://arxiv.org/abs/2308.12950

[5] Junjie Li et al., "Fine Tuning Large Language Model for Secure Code Generation," FORGE '24: Proceedings of the 2024 IEEE/ACM First International Conference on AI Foundation Models and Software Engineering, Pages 86 - 90, 12 June 2024. https://dl.acm.org/doi/10.1145/3650105.3652299

[6] Zhensu Sun et al., "On the importance of building high-quality training datasets for neural code search," ICSE '22: Proceedings of the 44th International Conference on Software Engineering, Pages 1609 - 1620, 05 July 2022. https://dl.acm.org/doi/10.1145/3510003.3510160

[7] Nusrat Jahan Prottasha et al., "Parameter-efficient fine-tuning of large language models using semantic knowledge tuning," Scientific Reports volume 14, Article number: 30667 (2024), 28 December 2024. https://www.nature.com/articles/s41598-024-75599-4

[8] Krishna Prasad Varadarajan Srinivasan et al., "Comparative Analysis of Different Efficient Fine Tuning Methods of Large Language Models (LLMs) in Low-Resource Setting," arXiv:2405.13181 [cs.CL], 21 May 2024. https://arxiv.org/abs/2405.13181

[9] Ridi Ferdiana, "The Impact of Artificial Intelligence on Programmer Productivity," Research Gate, February 2024. https://www.researchgate.net/publication/378962192_The_Impact_of_Artificial_Intelligence_on_Programmer_Productivity

[10] Peng-Hua Chu et al., "A test case refactoring approach for pattern-based software development," Software Quality Journal, Volume 20, Issue 1, Pages 43 - 75, 2012. https://dl.acm.org/doi/10.1007/s11219-011-9143-x

[11] Siddhant Sonkar, “Transfer Learning: Leveraging Pretrained Models For Limited Datasets Through Fine-Tuning,” International Journal of Information Technology and Management Information Systems (IJITMIS), Volume 16, Issue 2, March-April 2025, pp. 564-576. https://iaeme.com/MasterAdmin/Journal_uploads/IJITMIS/VOLUME_16_ISSUE_2/IJITMIS_16_02_037.pdf

[12] Siddhant Sonkar, “Recent Innovations in AI Privacy: Protecting Data in the Age of Machine Learning,” International Journal of Scientific Research in Computer Science, Engineering and Information Technology, Volume 11, Issue 2, March 5, 2025. https://ijsrcseit.com/index.php/home/article/view/CSEIT25112390/CSEIT25112390

[13] Siddhant Sonkar, “Reducing Seller Friction through Generative AI: An Analysis of E-commerce Innovation,” International Journal on Science and Technology (IJSAT), Volume 16, Issue 1, January-March 2025. https://www.ijsat.org/papers/2025/1/2331.pdf

[14] Siddhant Sonkar, “The Revolution Of AI in Modern Advertising: A Technical Overview,” International Research Journal of Modernization in Engineering Technology and Science, Volume:07/Issue:03/March-2025. https://www.irjmets.com/uploadedfiles/paper//issue_3_march_2025/68716/final/fin_irjmets1741603998.pdf

[15] Shivansh Gaur et al., “Generation of synthetic training data for handwritten Indic script recognition,” 2015 13th International Conference on Document Analysis and Recognition (ICDAR), 23 November 2015. https://ieeexplore.ieee.org/document/7333810