Advanced International Journal for Research (AIJFR)
E-ISSN: 3048-7641 ● Website: www.aijfr.com ● Email: editor@aijfr.com

NixGen: A Framework for Natural Language-Driven NixOS Configuration

Yash Ghogre[1], Divya Hingawe[2], Kinshuk Thapa[3]

[1,2,3]Computer Technology,
Yeshwantrao Chavan College of Engineering,
Nagpur, India

Abstract

This paper introduces NixGen, a generative AI framework that converts high-level user query into low-level declarative NixOS configuration. The system is designed to ease the switch to NixOS by reducing the difficulty involved in writing and understanding the configurations. Experimental evaluation demonstrates the effectiveness of NixGen, achieving 92.3% syntax validity and a CodeBLEU score of 0.72 on complex networking modules. By reducing the gap between intent-driven inputs and precise Nix expressions, NixGen significantly lowers the entry barrier for developers and system administrators, making the advantages of declarative, reproducible, and scalable system configuration more accessible to a wide range of audience.

Keywords: NixOS, declarative configuration, large language models, code generation, natural language processing, system administration, QLoRA, fine-tuning.

1. Introduction

This Configuration management has undergone a notable transformation with the rise of declarative systems. Unlike traditional approaches—where administrators define a set of commands to achieve a desired state—declarative systems focus on identifying the final state itself, allowing the system to determine how to achieve it. A prime example of this model is NixOS, a Linux distribution built on a purely functional and declarative model that guarantees system reliability, atomic upgrades, and reproducibility. In NixOS, every aspect of the system—varying from installed packages and running services to kernel level parameters and user accounts—is defined by configuration files written in the Nix expression language. By removing configuration drift and enabling seamless system rollbacks, this approach provides a sturdy foundation for managing complex and dynamic computing environments.
But there is a big paradox brought about by this power. The very features that give NixOS its strength also make it difficult for novices to learn and cause inefficiencies for even experienced administrators. Many people are not familiar with the Nix language's distinct syntax and functional programming ideas. Wider adoption is hampered by the difficulty and time required to master the extensive ecosystem of modules and options needed to configure a system.

A. Problem Statement

The inherent challenge of manually authoring and maintaining NixOS configurations is the main issue this study attempts to address. In addition to taking a long time, this procedure is very prone to mistakes.

A small syntactical error, like a misspelled attribute or a misplaced brace, can invalidate the entire system configuration and cause build failures. Furthermore, a thorough understanding of the NixOS module system is necessary for managing dependencies and making sure that various configuration elements don't conflict. Routine maintenance becomes a high-stakes task because users often struggle with misconfigurations, broken dependencies, and unexpected system behavior, even with thorough documentation.

B. Proposed Solution: The NixGen Framework

In order to overcome these obstacles, this paper introduces NixGen, a brand-new end-to-end framework that connects the low-level, exact syntax of Nix code with high-level user intent. NixGen is an intelligent assistant that converts natural language system requirements into legitimate, idiomatic, and secure NixOS configuration snippets by utilizing recent developments in large language models (LLMs). The main premise of this work is that the most time-consuming and prone to mistakes parts of NixOS configuration management can be automated by using a small, domain-specialized LLM to efficiently learn the grammar and semantics of the Nix language. NixGen significantly reduces the technical know-how and cognitive overhead needed to interact with a declarative operating system by enabling users to express their desired state in simple English.
This solution's focus on dependability and safety is a crucial component. Instead of just being appended to files, generated code is parsed and merged by a structure-aware mechanism that is aware of the Nix language's syntax. This guarantees that the user's current configuration will always remain intact. This method makes NixOS manage a conversational process rather than a manual coding task, increasing the system's usability and accessibility for all users.

C. Contributions

The main contributions this paper makes to the field of AI-driven system administration are as follows:

- The development and deployment of NixGen, an end-to-end system that uses natural language prompts to generate and manage NixOS configurations, shows how generative AI can be used to solve a challenging, real-world systems issue.
- A thorough and reproducible process for selecting a unique, domain-specific dataset for NixOS configurations from various open sources. Using the QLoRA technique, this includes a workflow for effectively fine-tuning a compact LLM (Llama 3.2 3B), enabling domain specialization on commodity hardware.
- The use of the Rust-based rnix-parser to create a reliable, AST-aware parsing and merging mechanism. In addition to preventing corruption and offering a secure integration path for AI-generated code, this component guarantees the syntactic integrity and original formatting of existing configuration files.
- An extensive empirical analysis of the NixGen model. Along with a thorough examination of resource efficiency that validates the viability of local, on-device deployment, the results show that the generated code has high syntactic validity (92.3%) and semantic accuracy (88.1%).

NixGen tackles the fundamental problem of mitigating a paradigm shift rather than just syntactical translation. Traditionally, system administrators are taught to think urgently and use commands like apt-get install to make gradual changes to a system's state. Declarative thinking is required by NixOS, in which the intended end state is described. One of the main causes of the steep learning curve is this

cognitive leap. By serving as a "paradigm bridge," NixGen enables users to convey their intent in a recognizable, mandatory format (such as "Install coding fonts"), which the system subsequently converts into the appropriate declarative state change. As a result, adopting and using NixOS requires a much-reduced cognitive load. The success of this model offers a strong framework for democratizing other declarative, complex technologies that are common in contemporary infrastructure, like Terraform, Ansible, and Kubernetes, where adoption is still significantly hampered by a lack of skills.

2. RELATED WORK

A. Declarative System Management and NixOS

Declarative configuration management has emerged as a key idea in contemporary infrastructure-as-code and DevOps methodologies. Dolstra et al. presented NixOS as a fully functional Linux distribution that embodies this paradigm in their seminal work. NixOS's declarative model and functional package manager directly lead to its fundamental tenets of reproducibility, atomic upgrades, and safe rollbacks.3. The "DLL hell" that results from destructive updates to shared components and non-reproducible environments is one of the long-standing problems with traditional imperative package management systems that these features address. The steep learning curve and possible performance overheads are two inherent difficulties of this approach that have prevented its wider adoption and directly drive the need for tools like NixGen, as acknowledged in the same literature.

B. Deep Learning for Code Generation

With the introduction of deep learning, especially transformer-based models, the field of automated code generation has advanced remarkably. According to Yang et al., the field has advanced from conventional rule-based systems to complex neural networks that can produce code from natural language descriptions that is both syntactically and semantically correct. These models have proven to be adept at a wide range of tasks and programming languages. The addition of structural information to the generation process is a significant advancement in this field. To better maintain the structural and semantic integrity of the generated code, Tipirneni et al. presented StructCoder, a transformer model that makes use of Abstract Syntax Trees (ASTs) and Data Flow Graphs (DFGs). The literature's focus on structure supports NixGen's architectural choices, particularly the use of an AST-based parser to securely handle the model's output as opposed to relying on brittle text-based manipulation.

C. Natural Language Interfaces for System Administration

NixGen fits into a larger trend of creating natural language user interfaces to make interacting with complicated software systems easier. Nguyen-Duc et al.'s suggested research agenda emphasizes how Generative AI has the ability to revolutionize every stage of the software engineering lifecycle, from requirements collection to deployment and maintenance. The usefulness of AI assistants in general-purpose programming has already been shown by tools such as GitHub Copilot. This idea is expanded by NixGen to the specific field of declarative system configuration, a task that is particularly well-suited to AI support because of its deterministic nature and well-defined grammar.

D. Research Gap

Although general-purpose code generation has advanced, there is still a large research gap in applying these methods to specialized, domain-specific languages such as Nix. Despite their strength, general-

purpose models frequently lack the specific expertise needed to handle the idiomatic conventions, evaluation semantics, and complexities of the NixOS module system. They might produce configurations that are syntactically correct but functionally flawed or unsafe. By showing that a compact, precisely tuned model can perform better in this field than general-purpose methods, this work closes that gap. The targeted methodology of NixGen, which combines a structure-aware integration pipeline with domain-specific data curation, offers a fresh and practical solution to an issue that the AI for code generation community has not yet addressed.

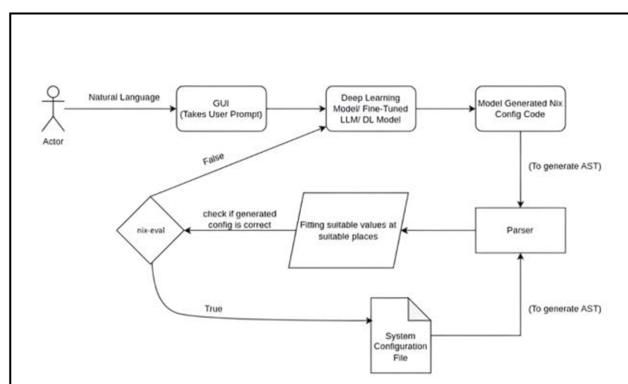
3. The NixGen System Architecture

A user's natural language request is converted into a securely applied change to their NixOS configuration by the multi-stage pipeline that is the NixGen system. Fig. 1 shows the operational flow, which is as follows:

- User Prompt: The user uses a command-line interface to submit a request in plain English, such as "Enable the X11 windowing system and install the Fira Code font."
- Intent Parsing: To structure the request for the LLM, a small initial parser examines the user's prompt to identify important entities and intended actions.
- LLM Generation: The optimized Llama 3.2 3B model receives the structured prompt and produces the relevant Nix code snippet.
- Validation: Nix's built-in evaluation tools, like nix-eval-jobs, receive the generated snippet and perform a preliminary syntax and validity check. This serves as an essential barrier that stops erroneous code from running.
- AST Parser & Merger: The code is sent to the Rust-based parser if it is valid. This component converts both the new snippet and the user's current configuration.nix file into ASTs. The new configuration is then cleverly merged into the current tree.
- File Update: The process is finished by overwriting the original configuration.nix file with the modified AST serialized back into formatted Nix code.

E. Data Curation for a Niche Domain

The creation of a methodology for producing a high-quality, domain-specific dataset in an area where none previously existed is one of this work's main contributions. A multi-stage data curation process was required because there was no pre-made corpus of NixOS configurations appropriate for machine learning:



- Source Identification: The official NixOS handbook, community-maintained wikis, public GitHub repositories with individual configurations (dotfiles), and user forums were all considered as possible sources of high-quality NixOS configurations. The goal of this multi-source approach was to capture a variety of real-world use cases as well as classic, best-practice examples.
- Data Collection: Two essential system modules—"Networking" and "Fonts"—were the focus of a focused collection effort. Numerous public repositories (such as wochap/nix-config and Misterio77/nix-starter-configs) were scraped and cloned using automated scripts, and pertinent code snippets were extracted from documentation. An initial raw corpus of several thousand configuration fragments was produced by this process, and 300 distinct, excellent snippets were eventually chosen for each module.
- Cleaning and Normalization: Leaning and Normalization: A thorough cleaning and normalization pipeline was applied to the raw data. Identical snippets had to be deduplicated, each fragment's syntax had to be checked using nix-instantiate to remove any invalid code, and stylistic conventions (such as attribute ordering and indentation) had to be normalized to guarantee consistency throughout the dataset. In order to create a clean training signal for the LLM, this last step was essential.

For other researchers looking to apply AI techniques to specialized technical domains without established datasets, this documented methodology offers a useful road map.

F. The Generation Core: Fine-Tuning Llama

The heart of NixGen is a large language model fine-tuned specifically for Nix code generation.

Because of its remarkable performance and resource efficiency balance, the Llama 3.2 3B model was chosen. It can follow instructions because it is instruction-tuned, and its small size makes it perfect for local deployment on commodity hardware. Being an entirely open-source model, it provides total control over the deployment and fine-tuning process while protecting data privacy, which is an important factor for any tool that manages system configuration. The decision to use a local-first model is not just a technical one; rather, it is a conscious design choice that supports the self-hosting and privacy-conscious mindset of the Linux and NixOS communities. Sending sensitive configuration data to third-party APIs entails security risks and connectivity requirements that are avoided.

The Unsloth library, an open-source framework that uses hand-optimized Triton kernels for incredibly effective training, was used to fine-tune the model. 4-bit training in mixed precision is made possible by the Quantized Low-Rank Adaptation (QLoRA) technique. By freezing the pre-trained model weights and only training a few flexible "adapter" layers, this significantly lowers the GPU VRAM requirements (by as much as 80%). Because of its efficiency, the model could be optimized on a single 16 GB GPU, like those found on Google Colab. Three training epochs, a logical batch size of 32, and a learning rate of 2×10^{-5} were the main hyperparameters that were employed in order to balance robust generalization with quick convergence.

G. Safe Integration with AST-Based Parsing

The way NixGen integrates generated code into pre-existing files is a crucial safety feature. Because it ignores syntax, comments, and formatting, a straightforward text-based append or search-and-replace operation runs the risk of corrupting the user's configuration.

- **The Role of rnix-parser:** NixGen uses the Rust-based rnix-parser to get around this. Unlike a conventional AST, this library preserves all of the original source text's information, including comments and whitespace, when parsing Nix code into a lossless Concrete Syntax Tree (CST). This is essential for changing the code without erasing the user's initial annotations and formatting. Research demonstrating that adding structural information, like ASTs, enhances the quality and accuracy of generated code validates this structure-aware approach.
- **Merging Logic:** Both the user's current configuration and the LLM-generated snippet must be parsed into their corresponding ASTs as part of the integration process. After that, the system searches the current tree to find the new code's proper location. In order to guarantee that the resulting structure is syntactically valid, it uses clever merging techniques, such as concatenating elements to an existing list (environment.systemPackages) or adding new attributes to an attribute set.
- **Cross-Language Integration:** Rust implements this complex parsing logic for memory safety and performance. The PyO3 crate is used to integrate this with the system's Python-based AI components. PyO3 makes it possible to compile the Rust code into a native Python module that can be imported and used straight out of Python. Combining the best features of both ecosystems—Rust's high-performance, safety-critical systems programming capabilities and Python's extensive AI/ML libraries—this is an advanced software engineering approach.

4. XPERIMENTAL EVALUATION AND RESULTS

H. Experimental Setup

To evaluate the effectiveness, efficiency, and quality of the NixGen framework, a thorough assessment was carried out.

- **Dataset:** The experiment made use of the carefully selected dataset, which included 300 distinct configuration snippets for the "Networking" and "Fonts" modules. To assess the model's capacity for generalization, the data was divided into a 20% validation set (60 snippets per module) and an 80% training set (240 snippets per module).
- **Metrics:** To give a comprehensive evaluation of the system, a set of metrics was established:
  - Exact Match Accuracy: The proportion of produced outputs that precisely match the reference configuration is known as accuracy.
  - Syntax Validity Rate: The proportion of produced outputs that successfully pass nix-instantiate validation without any syntax errors is known as the syntax validity rate. Compared to exact match, this is a more useful indicator of correctness.
  - Semantic Accuracy: The proportion of syntactically sound outputs that result in a system configuration that is both functionally correct and intended, as assessed by functional testing and manual review.
  - CodeBLEU: A composite metric for tasks involving the generation of code. By taking structural similarity into account, it provides a more nuanced evaluation of code quality

than just accuracy. It measures n-gram match, weighted n-gram match, AST match, and data-flow match.

- Resource Utilization: The refined model was converted to the GGUF format in order to measure its appropriateness for local deployment. On a typical CPU, the following parameters were recorded during inference: CPU Usage, Memory Footprint (RAM), Disk Size, and Model Load Time.

I. Performance Analysis

The experimental outcomes show how well the adjusted model and the entire NixGen architecture work. The tables below provide a summary of the main conclusions.

TABLE I. MODEL GENERATION PERFORMANCE

Metric	Value
Exact Match Accuracy	76.5%
Syntax Validity Rate	92.3%
Semantic Accuracy	88.1%
Inference Latency	1.42
Token Efficiency	284

TABLE II. CODE QUALITY BENCHMARK

	Networking Module	Fonts Module
CodeBLEU Score	0.72	0.68

TABLE III. RESOURCE UTILIZATION ON LOCAL HARDWARE

Metric	Value
CPU Usage	68%
Memory Footprint	4.2 GB
Disk Size	1.7 GB
Load Time	4.5 sec

J. Discussion

The outcomes offer compelling proof of the NixGen approach's feasibility.

- **Analyzing the Accuracy-Validity Gap:** One important discovery is the notable discrepancy between the Syntax Validity Rate (92.3%) and the Exact Match Accuracy (76.5%). This is a strength of the model rather than a flaw. It suggests that the LLM is doing more than just copying and remembering training examples. Rather, it has mastered the Nix language's flexible grammar and can produce several textually distinct but functionally equivalent answers to a given prompt.

This is further supported by the high Semantic Accuracy (88.1%), which verifies that the great majority of these syntactically sound outputs also accomplish the user's intended objective. This exhibits a level of "understanding" that goes beyond what can be captured by simple string matching.

- **Module-Specific Performance:** The two tested domains perform differently, according to the CodeBLEU scores. The 'Fonts' module scored 0.68, while the 'Networking' module scored 0.72. The nature of the configuration data itself is a tenable explanation for this disparity. With nested attribute sets and clearly defined options, networking configurations in NixOS are typically very structured. Font configurations, on the other hand, are frequently made up of flat, straightforward lists of package names, which can be less structured and more arbitrary for the model to learn. This implies a relationship between the target domain's structural regularity and the model's performance.
- **Error Analysis and Limitations:** Errors still happen in a small percentage of cases, according to an honest evaluation of the system. Misidentifying obscure package names, creating conflicting service definitions (e.g., enabling two different display managers at the same time), and failing to pay attention to system architecture details were common failure modes during iterative testing. These drawbacks emphasize the significance of the validation phase and imply that, even though AI is capable of handling the great majority of routine tasks, human oversight is still beneficial for configurations that are complex or uncommon.

The viability of the local-first architecture is validated by the resource utilization metrics presented in Table 3. The model can run smoothly on a typical developer laptop with a memory footprint of 4.2 GB and a disk size of 1.7 GB, providing interactive performance with an average latency of only 1.42 seconds. Because of this, NixGen is a useful and approachable tool for daily use.

5. CONCLUSION AND FUTURE DIRECTIONS

K. Conclusion

NixGen, an AI-driven framework that converts plain-English intents into production-ready NixOS configurations, was successfully designed, implemented, and assessed in this project. NixGen tackles the major usability issues and steep learning curve that have traditionally prevented the widespread adoption of the potent NixOS declarative model by utilizing a compact Llama 3.2 3B model that has been refined on a specially selected dataset. The system's high performance, as evidenced by its 88.1% semantic accuracy rate and 92.3% syntax validity rate, confirms that using domain-specialized LLMs for configuration management is effective.

Additionally, the incorporation of a strong, AST-based parser guarantees the safe merging of AI-generated code while maintaining the integrity of user configurations. The final GGUF model's small resource footprint attests to the fact that this potent feature can be implemented locally while maintaining high performance and protecting user privacy. Finally, by reducing the barrier to entry for novices and speeding up routine configuration tasks for experts, NixGen offers a major advancement in the accessibility of declarative systems. It exhibits a potent pattern for using generative AI to improve the safety and usability of intricate technical systems.

L. Future Directions

Several promising directions for further research can be investigated to increase the system's capabilities, building on the strong foundation this work established:

- **Expanding Domain Coverage:** The 'Networking' and 'Fonts' modules were used to train the current model. To create a more complete assistant, it would be obvious to scale the data curation and fine-tuning process to cover a much larger range of the NixOS module system, including services, hardware, virtualization, and security configurations.
- **Enhanced Semantic Validation:** The majority of the current validation is syntactic. AI-powered static analysis tools may be incorporated into future research to carry out more thorough semantic validation. Beyond just being syntactically correct, such a system could proactively alert users to possible security flaws (like opening insecure ports), performance anti-patterns, or logical conflicts in the configurations that are generated.
- **Interactive and Explainable AI:** Explainable AI (XAI) concepts could be applied to improve user control and trust. This would entail creating mechanisms that would allow the AI to provide an explanation, based on pertinent documentation or best practices, for why it produced a specific configuration snippet. Users could ask follow-up questions and work together to improve the generated code through an interactive, conversational interface.
- **Closed-Loop Learning System:** Developing a system with an ongoing learning loop would be a potent extension. The model could be retrained on a regular basis using the results of the validation tools, user corrections, and feedback that is automatically gathered. By doing this, the system would be able to grow and change over time, learning from its errors and keeping up with the rapidly changing NixOS ecosystem.

6. Acknowledgement

The authors express sincere gratitude to their faculty mentor and instructors for their valuable guidance, constructive feedback, and continuous support throughout this research on NixGen: A Generative AI Framework for NixOS Configuration Generation . The authors also acknowledge their institution for providing the necessary resources and academic environment to carry out this work. Appreciation is extended to the open-source community for access to NixOS configurations, documentation, and tools that supported dataset curation and system development. The contributions of developers of foundational technologies, including NixOS and large language model frameworks, are also gratefully recognized. Finally, the authors thank their family and peers for their encouragement and support.

References

1. Dolstra, E., Löh, A., and Pierron, N., "NixOS: A Purely Functional Linux Distribution," Journal of Functional Programming, 21(4-5) (2011) 429-472.
2. Yang, Z., Chen, S., Gao, C., Li, Z., Li, G., and Lv, R., "Deep Learning Based Code Generation Methods: Literature Review," arXiv preprint arXiv:2303.01056 (2023).
3. Zhang, H., Zhang, K., Li, Z., Li, J., Li, Y., Zhao, Y., Zhu, Y., Liu, F., Li, G., and Jin, Z., "Deep Learning for Code Generation: A Survey,"

4. Science China Information Sciences, 67(8) (2024) 182101.
5. Tipirneni, S., Zhu, M., and Reddy, C. K., "StructCoder: Structure-Aware Transformer for Code Generation," arXiv preprint arXiv:2206.05239 (2022).
6. Gallego-Madrid, J., Bru-Santa, I., Sanchez-Iborra, R., and Skarmeta, A., "Integrating Machine Learning Models into the Linux Kernel: Opportunities and Challenges," Proceedings of the 7th International Conference on Mobile Internet Security (MobiSec'23), Okinawa, Japan (2023).
7. Ge, Y., Ren, Y., Hua, W., Xu, S., Tan, J., and Zhang, Y., "LLM as OS, Agents as Apps: Envisioning AIOS, Agents and the AIOS-Agent Ecosystem," (arXiv preprint arXiv:2312.03815 (2023)).
8. Mei, K., Zhu, X., Xu, W., Hua, W., Jin, M., Li, Z., Xu, S., Ye, R., Ge, Y., and Zhang, Y., "AIOS: LLM Agent Operating System," arXiv preprint arXiv:2403.16971 (2024).
9. Zhang, P., Zeng, G., Wang, T., and Lu, W., "TinyLlama: An Open-Source Small Language Model," arXiv preprint arXiv:2401.02385 (2024).
10. Zhang, Y., Zhao, X., Yin, J., Zhang, L., and Chen, Z., "Operating System and Artificial Intelligence: A Systematic Review," arXiv preprint arXiv:2407.14567 (2024).
11. Grigorescu, S., and Zaha, M., "CyberCortex.AI: An Al-based Operating System for Autonomous Robotics and Complex Automation," arXiv preprint arXiv:2409.01241 (2024).
12. Hu, S., Ouyang, M., Gao, D., and Shou, M. Z., "The Dawn of GUI Agent: A Preliminary Case Study with Claude 3.5 Computer Use," arXiv preprint arXiv:2411.10323 (2024).
13. Kulkarni, M., and Kamble, T., "Integration of Machine Learning into Operating Systems: A Survey," International Journal of Creative Research Thoughts (IJCRT), 8(4) (2020).
14. Jia, S., Wang, X., Song, M., and Chen, G., "Agent Centric Operating System a Comprehensive Review and Outlook for Operating System," arXiv preprint arXiv:2411.17710 (2024).
15. Nguyen-Duc, A., Cabrero-Daniel, B., Przybylek, A., et al., "Generative Artificial Intelligence for Software Engineering A Research Agenda," arXiv preprint arXiv:2310.18648 (2023).
16. Grattafiori, A., Dubey, A., Jauhri, A., et al., "The Llama 3 Herd of Models," arXiv preprint arXiv:2407.21783 (2024).