Architecting Next-Generation Shells: AI-Driven Command Assistance and Proactive Safety Mechanisms in the Command-Line

Humza Anwar Khan Ishan Khan Khateeb Aamir Usmani Muhammad Ali

Department of Computer Science and Engineering, Integral University, Lucknow

ABSTRACT
The Command Line Interface (CLI) is a very good tool, but it can scare people away because of the strict commands you have to type in. Command-line interfaces are also very error-sensitive when it comes to typing commands into your computer and if you accidentally run a command that is harmful then you might erase everything on your computer. In addition to this, the basic shells that have been created for many years starting with Bash, usually have little to no help for the user. This leads to the purpose of this review. This review will discuss the ideas behind building a lightweight custom shell, programmed in C using the SAGE (Suggestive Auto-Command Generate Environment) prototype to demonstrate how the performance of a custom shell was developed. The advantages of building a lightweight shell are that you can provide intelligent input correction, proactive safety checks (before actually executing a command), replayable command history, and the ability to monitor system resources before executing commands without overloading your CPU and/or RAM. This article gives a solid foundation for how the SAGE prototype will help evolve the standard command-line interface (CLI) into a more intelligent, sustainable, and productive assistant to the user.

Keywords
Command-line interface, intelligent shell, AI-based command assistance, proactive safety systems, resource-efficient execution, lightweight architecture

1. INTRODUCTION
The Unix command line interface is a powerful environment, but its tools, the shells, are generally slow to evolve, falling behind the needs of the modern user [1]. This creates a false dichotomy for the user to choose from:

a. They can stay in a traditional shell (like Bash or ash) with a minimal environment that is fast and lightweight but provides nearly no assistance, and requires the user to set everything up themselves to be productive
b. Or they can use a modern shell that has many features (like fish or a heavily customized version of zsh) with helpful features such as auto-suggestions and auto-corrections while often having slower startup time and resource consumption.

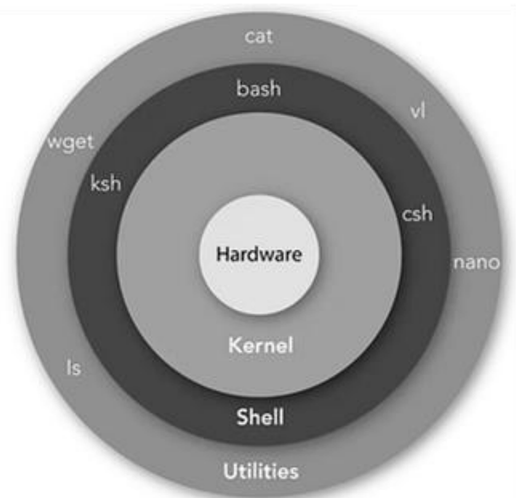

Figure 1: layered architecture of UNIX Operating System

This article explores the potential for designing and creating a custom lightweight shell using C in an efficient and intelligent manner. Both performance and usability can be improved through a well-designed approach rather than accepting trade-offs between them. It is possible to do this by utilizing C's low-level features and communicating directly with the kernel of the operating system. Therefore, the shell can maintain a high level of performance while also supporting new and useful functions.

In particular, the primary emphasis of this study is on five key areas in which features have been developed through the use of low-level programming techniques. As such, the shell is intended to function as more than just a simple command interpreter, but rather a proactive support tool that enhances user experience and productivity by improving five major areas:

1) **Intelligent Input:** This includes any features providing instant, as-you-type suggestions based on command history and contextual completion. It likewise encompasses the utilization of typo-correction algorithms (i.e., Levenshtein Distance) to catch a user's common (and possibly embarrassing) typos (i.e., **"gti status"** becomes **"git status"**) and provide the user with an immediate fix turning a failed entry into a success [3].

2) **Proactive Safety:** This involves proactive real-time safety checks that warn users about command liability that may be risky or destructive before the command actually runs (when there is something explicitly dangerous the shell can identify and take action before double confirmation). The shell evaluates a command prior to executing it. High risk patterns (such as mass deletion and using sudo incorrectly) are checked before executing the command. If a high-risk command is detected, the shell requests confirmation from the user prior to executing the imported/illegal command [4].

3) **Replayable History**: In Replayable History, users don't just see a time-stamped list of typed commands in a document. A Pseudo-Terminal (PTY) can be used to capture important parts of the session which enables you to log user commands and write the OUTPUT from that user command, providing for a complete auditable replay of everything that occurred during the session. This allows a user to debug their shell session, share a shell session with another user, or perform a technical audit of the shell session [5].

4) **Resource Awareness:** After each command is run from the shell, performance data will be generated. Metrics such as CPU usage, memory usage and elapsed (wall) time will be displayed. Elapsed time and CPU time will help give users a quick understanding of the **cost** incurred by their use of the system. As well, elapsed time will provide users with a quick way to evaluate their impact on operations (i.e. how long the command took, and how long it took to finish processing).

5) **Core Efficiency:** This is what gives the shell a responsive feel. Core functionality such as redirection commands (>, |, <) are implemented with low-level C system calls (such as dup2() and pipe()). This ensures that no external dependency is introduced, and as a result, these core functions should be as fast as and as efficient as the kernel itself [6].

This document details the transformation of intelligent terminals from their early stages of development into the advanced systems available today. It is divided into three sections: First, an overview of the different types of user support, based on previous experience and currently available resources; secondly challenges that make it difficult to build a more user-friendly and safe interaction with intelligent terminals; and then a comparison between traditional and modern user command procedures for executing commands with intelligent terminal systems. Finally, the SAGE shell is examined as an application of using artificial intelligence applications to improve the safety and intelligence of a command line interface.

2. EXISTING SOLUTIONS AND RESEARCH INSIGHT

Current solutions along with much of the published research provide significant perspectives into the ongoing evolution of this tool, revealing that there continues to be a lack of safety and ease of use in the Unix shell environment in 2024 [7].

2.1 Traditional Shells and Their Limitations

Shell environments have a purpose and they frequently are a critical tool, but they were created based on technical power and creativity and not safety or ease of use [8].

Design Philosophy and Power:

a. The classic standard shell (e.g., Bourne Shell (sh); C Shell (csh); Korn Shell (ksh)) shells were primarily built with the idea of user flexibility.
b. Classic standard shells allow for the user to customize the extent of control they could place over the computer. The primary goal of shells was textually controlling processes, files, and automate tasks for the user [8].

Usability and Safety Issues:

a. Once again, the shells are poorly designed for a human use case and force users to find ways to address usability, safety, and feedback issues on their own.
b. The shells do not provide basic features, such as user support or situation context.
c. The shell interface only does recall versus recognition, which requires users to memorize syntax (there could be thousands of commands tested by thousands of authors). They do not provide command corrections or safety confirmations in real-time [9].

Severe Errors and Consequences:

a. Having minimal safety features means that shells are prone to making severe errors.
b. Not being attentive to the user protection provides the entire load for safety on the end-user.

2.2 2.2 Customization and User Environment Tools

1. The recognized limitations have motivated numerous user driven and automatic attempts to customize and improve the shell environment, creating a clearly defined need for built-in assistance.
2. Addressing the issue, configuring the shell environment via “dot files” is a barrier when mistakes can cause breaks since even a single small mistake can break the environment, and it is so prevalent it has been called the “dot file virus”.
3. Systems like the user-setup (Elling & Long, 1992) were also developed in response to this deficiency to automate the environment setup process, creating correct by construction shell files for the novice user [10].
4. However, these systems were only aimed at environmental setup, and did not provide intelligent assistance for the runtime.

Data from user aliases:

Schröder and Cito (2021) conducted an analysis of more than 2.2 million alias definitions, which showed common user practices that offset the shortcomings of shell environments [7].

Users will use:

1. Shortcuts and Nicknaming Commands (e.g., alias g='git').
2. Abbreviating subcommands (e.g, alias gs='git status').

3. Error correction in aliases to protect against typos.
4. Defaults that take precedence over usage (i.e., alias rm='rm -i') would indicate an expectation of a "safety net," but this has not been built into the working environments.

These customization characteristics also point to a need for pre-programmed error handling and improved intelligent default action so as to make the system more user-friendly and reduce the chance of errors occurring through bad user actions.

2.3 Modern Developments and Intelligence Gaps

Recent research trends have increasingly shifted toward intelligent terminal designs, but this shift often comes at the cost of performance efficiency or requires users to rely heavily on scripted automation to maintain usability.

Highlighted longer-term systemic deficiencies: Greenberg et al. (2021) previously identified major long-term deficiencies related to the Unix shell ecosystem [11]. These include no formal semantics, designed error-prone shells, and a lack of interactive capacity for the user experience.

Solutions that focus on the performance aspects: Technologies such as PaSh and POSH have attempted to provide performant solutions based on parallelization to increase the efficiency of the shell's performance as it engages with the system process. Although these are evolutionary steps to performance, these projects did not account for usability or safety with respect to the user [12].

Scripting as the mechanism of automation and monitoring: Scripting in the classic tools (Shell, AWK, or Sed) remain prevalent for administering the Unix system. Between developing scripts, system administrators often will rely on manual operations meant to combine the basic tools. For example, an operator may execute df or awk to provide an alert when disk resources are running low on space. While these are effective and efficient, they do represent long-term administrative scripting workflows which, while efficient, are still manually-driven and therefore contain no built-in error protection from the administrator (operator) actions. Such an information gap also exists between scripting and traditional shell: traditional shells return no feedback or "story" of the commands' impact on system performance (CPU/memory) forcing system administrators to begin writing their own scripts to indicate health-related monitoring of the Shell [13].

2.4 Educational and Specialized Shell Implementations

A distinct class of shell-like projects targets domain-specific or pedagogical purposes which, as a part of the reflected design of these implementations, do not commonly include enhanced intelligent features.

Educational Shells: Other projects, like ghz-sh, focused on being educational, aiming at demonstrating some of the basics of shell operation, specifically providing evidence for core mechanics like process management, I/O redirection, and system calls like fork() and exec(). While educational shells initiate the process of exposing shell architecture, there are generally no intelligent or adaptive features that fill a gap in usability.

Specialized Shells: Other projects, such as ShellFish, design and build a new lightweight shell with a concrete domain target, for example Android forensics. Again, the purpose of these shells is for function in a specific domain and do not generalize intelligent user assistance or improvement in core CLI safety.

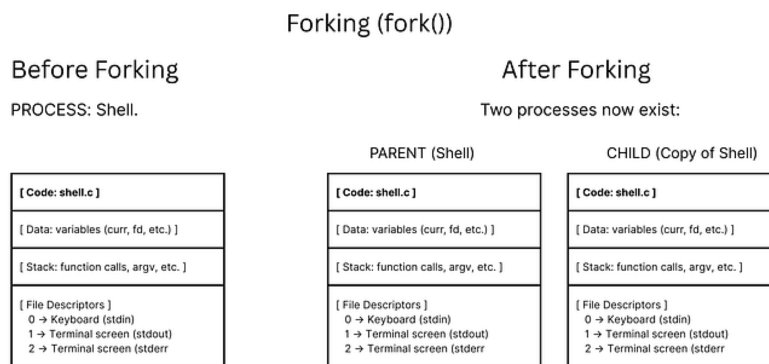

Figure 2: process creation in UNIX using fork()

3. RESEARCH GAP

The literature on the subject of shell usage has demonstrated a clear consensus that traditional shells are poorly designed for humans to use, and users have to recover the usability, safety, and feedback deficits provided by the shell with just their manual dexterity as they employ the shell.

3.1 Assistance Gap:

1) One of the most commonly cited issues is user guidance, or the assistance Gap. In a typical shell application, for any command that doesn't work (typo, syntax error, etc.), there is no useful feedback from the application, leaving the user in a frustrating trial-and-error process [14].
2) This is not a 21st century phenomenon; AI research has investigated predicting UNIX command lines to adapt to user behaviour as far back as 1988. The interface relies on "recall over recognition," compelling the user to recall the exact syntax for thousands of commands.
3) Some empirical evidence supporting this claim is in a study published in 2021 looking at the command tool grep, where participants expressed difficulty using its "complex and unintuitive" syntax.
4) Finally, a large-scale study of over 2.2 million shell aliases conducted by Schröder and Cito found that

the most frequently cited customization practices include shortcuts for nicknaming Commands, (e.g., alias g='git') and abbreviating subcommands (e.g., alias gs='git status'). This provides an observable, but unfulfilled, user want for a more intuitive and forgiving interface [7].

3.2 Safety Gap:

1) Again, the traditional shell environment has an inherent "safety gap" beyond usability that has no built-in safeguards or confirmation prompts against the execution of potentially undesirable commands.
2) This design puts the entire safety burden on the end-user where a simple lack of attention can result in "serious consequences".
3) The alias study substantiates this finding, as a common practice is "overriding defaults" (for instance, alias rm='rm -i'). This is a contradiction to a finding that many users opted for an override of the defaults rm is setting. People are seeking a "safety net" that is absent within the traditional shell environment.
4) Moreover, academic literature review provides additional endorsements to this need to "improve command-line safety through user confirmation mechanisms". The safety gap also applies to shell scripting in that scripts running with elevated privileges are vulnerable to shell injection if user inputs are not properly validated.

3.3 Configuration & Information Gap:

1) The difficulty of configuring the shell itself (the dot files) is a huge barrier, even a single error can break the environment.
2) The problem is so common it has been dubbed the dot file virus. The virus is manifested as non-functional or broken configurations being passed unintentionally and uncritically from user to user, which waste vast amounts of helpline time.
3) Systems such as user-setup were developed to ameliorate the issue, by creating correct-by-construction shell files for novice users. The situation is aggravated by an Information Gap because traditional shells will not give you any feedback about how a command is affecting system performance (CPU/Memory).
4) This forces system administrators to write their own scripts to monitor system health, and in certain fields like bioinformatics and other specialized environments, it is crucial to monitor complex command-line pipelines [15].

3.4 Usability and Accessibility Difficulties:

1) This fundamental gap leads to widespread usability and accessibility difficulties that inhibit general users from adopting the CLI.
2) Suiting the Experts: Traditional shells tend to suit expert users.
3) The CLI is impersonal & very harsh on novices.

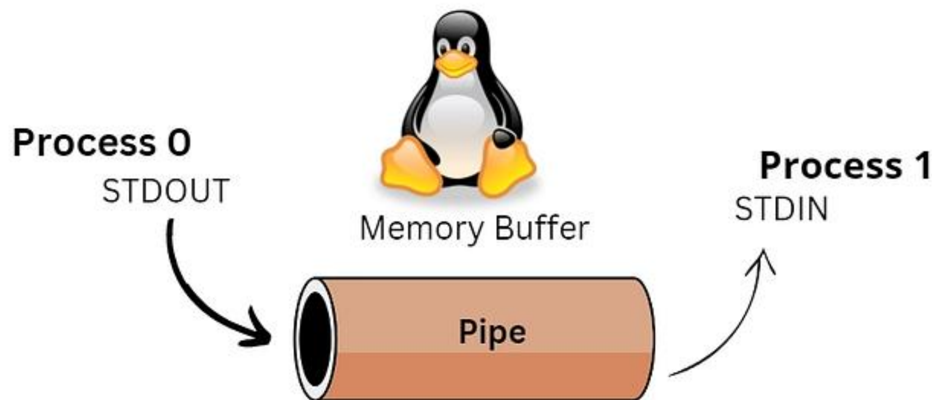

Figure 3: Inter-process communication using pipe in UNIX

4. SAGE (SUGGESTIVE AUTO-COMMAND GENERATIVE ENVIRONMENT) SHELL

The SAGE project exemplifies the real-world implementation of a shell built to address the user's identified shortcomings the literature presented.

4.1 Core Architecture:

The SAGE shell is implemented completely in the C programming language, to ensure maximum efficiency and to gain access to the entire set of POSIX API. The SAGE shell is a traditional shell architecture designed utilizing a Read-Eval-Print Loop (REPL) and executing a continuously running session that interactively responds to user prompts. The SAGE shell has a basic architecture that mirrors traditional shells, using the existing fork() and exec() system calls to create and execute a child process for the external command. In doing so, the developer is able to build out around a created "base" scope.

4.2 Intelligent Features as Native Solutions:

The SAGE shell represents new innovation by creating five identified features that natively address user gaps.

1. Intelligent Input:

Aim: To offer "Intelligent Correction" by identifying a typo, or invalid input, and provide a closest legal command suggestion.

Process: Using one of a few standard typo-correction algorithms, the particularly identified method is known as the Levenshtein Distance method which calculates the distance of alteration between two strings to arrive at a closest legal command [16].

Result: This fixes a common failed entry (gti status) so it is an immediate success (git status) and benefits the user by native fix versus configuring through aliases manually.

2. Proactive Safety:

Aim: To create a safety net by performing real-time, proactive safety checks.

Process: The shell will intercept commands that may lead to unsafe acts (for example: catastrophic commands such as rm -rf or entering a command with sudo) by first parsing the command and identifying high-risk patterns.

Result: The shell will require double confirmation before allowing the user to execute a command that poses risk or harm, thereby eliminating the need for the user to manually add an override for built-in defaults that require human intervention (e.g., rm -i).

3. Replayable History:

Aim: To capture key session information and not simply offer a record in the form of a text-based log.

Process: A shell can leverage a Pseudo-Terminal (PTY) and log user commands, as well as output from the command, with time-stamps throughout.

Result: This provides a fully auditable replayable session which should make it incredibly useful for debugging, sharing sessions with others, and technical audit purposes.

4. Resource Awareness:

Aim: To supply performance insights, therefore closing the Information Gap.

Process: After each command has executed, the shell will display performance information including CPU usage, memory usage, and execution time.

Result: This provides immediate feedback and an inherent cost awareness about resource use which, currently, system administrators must script by hand in a primitive form using tools, such as awk and sed. This information will be resource-based and will separate wall time from CPU time. Thus, it allows the user to recognize the resource impact of executing a command comes with some level of operational cost.

5. Core Efficiency:

Aim: To ensure the shell has a responsive feel

Process: Core interface functionality, specifically redirection commands ($>, |, <$) is implemented in C using direct low-level system calls (e.g. dup2() and pipe()).

Result: This ensures that these core functions are as fast and efficient as the kernel, while not providing any additional dependencies.

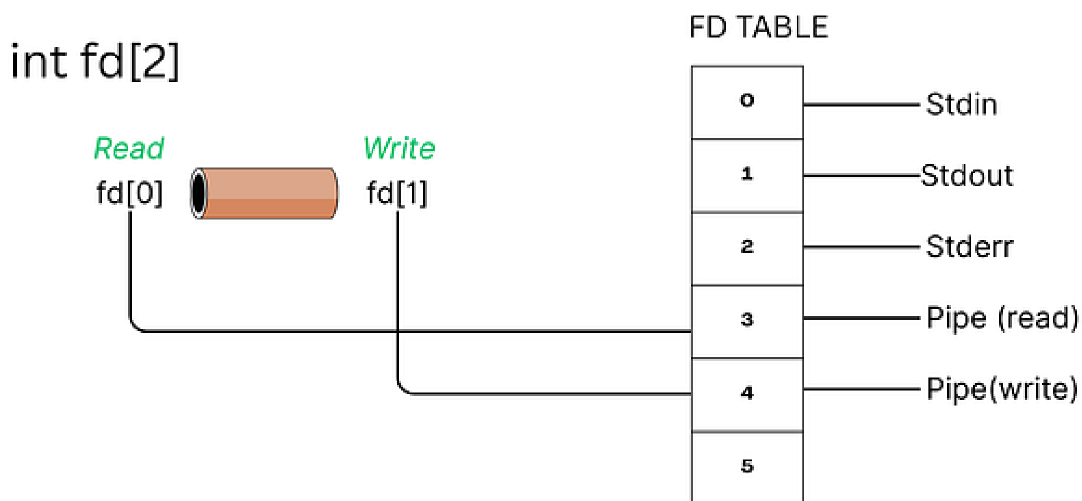

Figure 4: UNIX Pipe Implementation Using File Descriptors

5. ANALYSIS AND DISCUSSION

SAGE (Suggestive Auto-Command Generative Environment) shell improves upon traditional command line shells (Bash, sh) and provides user-friendly features while maintaining high levels of efficiency (high reliability) through embedded intelligence (built with low-level C programming). In contrast to modern shells that provide extensive functionality at the expense of more resource consumption than older shells. SAGE Shell solves this issue by embedding intelligent features into a low resource footprint.

From an end-user standpoint, SAGE Shell reduces user effort and increases user success through reduced user error rate. Significant usability improvements include:

- Automatic correction of typographical errors on commands written by the user
- Context-based command suggestion
- Reduced memory requirements for learning complex syntax.
- A more beginner-friendly command line experience

Industry standards for data backup require built-in safety mechanisms, whereas, traditional command line shells may execute dangerous commands without user notification. With the SAGE Shell, built-in safety checks detect:

- Command types that may cause high-risk operations (ex: delete).
- Command types that require confirmation from the user prior to execution to reduce risk of data loss or incorrect use.
- Reduced risk of accidental loss of data or damage to systems/data.
- Overall increased security of the host operating system.

High performance of SAGE shell is primarily due to its implementation at a low level. Some key attributes of performance include:

- Use of system calls like fork(), exec(), pipe(), and dup2()
- Low dependency on third-party tools
- Faster execution and response time
- Low memory and CPU utilization

Additionally, the system provides resource awareness which allows users to better understand the performance characteristics of their system:

- The real-time display of CPU and memory usage
- The tracking of execution time
- Improved resource management decision-making processes
- The monitoring of resources without the use of external scripts

To demonstrate the performance of the SAGE shell, there are different usage scenarios:

- Normal command execution (i.e. ls, cd) is performed quickly and efficiently
- Wrong command entry (i.e. gti status) is auto-corrected
- Risky command execution (i.e. rm -rf) is flagged with a warning message
- High resource command execution provides performance feedback.

Table 1: Comparison between Traditional Shells, Modern Shells and SAGE Shell

Feature	Traditional Shell (Bash, sh)	Modern Shell (zsh, fish)	SAGE Shell
User Assistance	Very limited	Moderate	High
Auto-Correction	No	Yes (limited)	Yes (intelligent)
Safety Mechanism	No	No	Yes
Resource Monitoring	External tools required	Limited	Built-in
Performance	High	Moderate	High
Resource Usage	Low	Higher	Low
Ease of Use	Difficult for beginners	Easier	User-friendly

Overall, the SAGE shell successfully addresses major limitations of traditional command-line interfaces, including lack of assistance, safety issues, and absence of resource feedback. By combining intelligent features with efficient system design, it provides a balanced and improved command-line experience suitable for both beginners and advanced users.

6. CONCLUSION

In conclusion, this paper shows that a simple command shell can be designed using the C programming language in a very efficient way by using low-level system calls and proper data structures. The SAGE Shell performs really well, with fast response time and low use of system resources. This will enable traditional command-line interfaces to be easier to use and help to provide additional assistance to users. For example, having the ability to fix minor typing errors, and warning users of the dangers associated with running particular commands will make the system safer and more reliable.

The SAGE Shell is also capable of improving a user's productivity by assisting in the management of system resource utilization (i.e. memory) while guiding the user in using the system efficiently. In addition, it has the ability to comprehend what is input by a user, thus providing useful suggestions, making it smarter than traditional command-line tools. Thus, it is capable of making the command-line interface an interactive, easier to use, and more productive environment than is currently the case.

In the future, additional improvements can be made to the system by incorporating technologies such as Artificial

Intelligence and Machine Learning to learn how a user operates and provide even more personalized suggestions to users. In addition, it will be possible to import natural language commands so users do not have to remember complex command-line syntax. Furthermore, the ability to have better compatibility with various operating systems and provide connectivity to cloud computing will enable SAGE to be an extremely powerful tool as computers continue to evolve.

7. REFERENCES

[1] J. T. Humphries *et al.*, "ghOSt: Fast & Flexible User-Space Delegation of Linux Scheduling," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, Virtual Event Germany: ACM, Oct. 2021, pp. 588–604. doi: 10.1145/3477132.3483542.

[2] P. Tyagi, "A Tribute to C Programming Language: History and Modern Applications," *Int. J. Comput. Appl.*.

[3] R. Haldar and D. Mukhopadhyay, "Levenshtein Distance Technique in Dictionary Lookup Methods: An Improved Approach," 2011, *arXiv*. doi: 10.48550/ARXIV.1101.1232.

[4] P. Notaro, S. Haeri, J. Cardoso, and M. Gerndt, "Command-line Risk Classification using Transformer-based Neural Architectures," 2024, *arXiv*. doi: 10.48550/ARXIV.2412.01655.

[5] F. Pritz, "Shell Activity Logging and Auditing in Exercise Environments of Security Lectures using OSS".

[6] W. R. Stevens, "Advanced Programming in the UNIX® Environment".

[7] M. Schröder and J. Cito, "An empirical investigation of command-line customization," *Empir. Softw. Eng.*, vol. 27, no. 2, p. 30, Mar. 2022, doi: 10.1007/s10664-021-10036-y.

[8] A. Mallett, *Mastering Linux shell scripting: master the complexities of Bash shell scripting and unlock the power of shell for your enterprise*, 1st ed. in Community Experience Distilled. Place of publication not identified: Packt Publishing, 2015.

[9] V. Svabensky, J. Vykopal, D. Tovarnak, and P. Celeda, "Toolset for Collecting Shell Commands and Its Application in Hands-on Cybersecurity Training," in *2021 IEEE Frontiers in Education Conference (FIE)*, Lincoln, NE, USA: IEEE, Oct. 2021, pp. 1–9. doi: 10.1109/FIE49875.2021.9637052.

[10] R. Elling and M. Long, "user-setup: A System for Custom Configuration of User Environments, or Helping Users Help Themselves".

[11] M. Greenberg, K. Kallas, and N. Vasilakis, "Unix shell programming: the next 50 years," in *Proceedings of the Workshop on Hot Topics in Operating Systems*, Ann Arbor Michigan: ACM, Jun. 2021, pp. 104–111. doi: 10.1145/3458336.3465294.

[12] N. Vasilakis, K. Kallas, K. Mamouras, A. Benetopoulos, and L. Cvetković, "PaSh: Light-touch Data-Parallel Shell Processing," 2020, doi: 10.48550/ARXIV.2007.09436.

[13] C. Fry, H. Abelson, G. Sussman, and J. Sussman, "Structure and Interpretation of Computer Programs," *Comput. Music J.*, vol. 9, no. 3, p. 81, 1985, doi: 10.2307/3679579.

[14] Aryan Karamtoth, "Custom UNIX Shell for Basic Terminal Operations: A project demonstrating the development of a custom UNIX Shell for basic Linux terminal operations using C," Nov. 13, 2024. doi: 10.31224/4101.

[15] D. T. Logeswari, "PIPES & FILTER – UNIX SHELL PROGRAMMING," vol. 6, no. 2, 2018.

[16] G. Horváth, A. Mészáros, K. Charaf, and P. Szilágyi, "Detecting anomalies in log files using the Damerau-Levenshtein distance metric," *Data Min. Knowl. Discov.*, vol. 40, no. 1, p. 12, Jan. 2026, doi: 10.1007/s10618-025-01182-8.