iJRASET
International Journal For Research in Applied Science and Engineering Technology

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Volume: 13 **Issue:** VI **Month of publication:** June 2025

DOI: https://doi.org/10.22214/ijraset.2025.72773

www.ijraset.com

Call: 08813907089 | E-mail ID: ijraset@gmail.com

International Journal for Research in Applied Science & Engineering Technology (IJRASET)
ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 13 Issue VI June 2025- Available at www.ijraset.com

Modern Approaches to Unix Automation: Shell Scripting, Configuration Management, and Security

Sambasiva Rao Madamanchi
Unix/Linux Administrator
Dept of Veterans Affairs (Austin, TX)

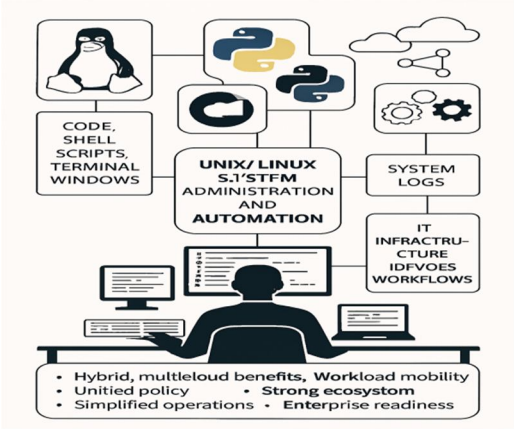
Abstract: ***This study examines key scripting methodologies essential for automating Unix system administration within modern IT infrastructures. As systems grow in complexity, automation becomes crucial for achieving consistency, scalability, and reliability. The study explores Unix shell environments including Bash, sh, and zsh, and introduces foundational scripting elements such as syntax, control flow, I/O redirection, and execution permissions. It highlights practical applications in user account management, backups, monitoring, and software deployment, supported by real world examples. Scheduling mechanisms such as cron, at, and systemd timers are discussed, along with robust practices for logging, modular design, and error handling. Advanced scripting concepts including parameter parsing, exit code evaluation, and text processing with grep, awk, and sed are covered. The integration of Python and Perl, as well as the role of configuration management tools like Ansible and Puppet, are also examined. Security considerations such as input validation and auditing are emphasized. This study concludes by addressing limitations and emerging trends in AI driven automation.***
Keywords: Unix system, Shell scripting, Automation, Configuration management, System monitoring

I. INTRODUCTION

A. Background

System administrators play a critical role in the management, upkeep, and troubleshooting of Unix and Linux environments. These operating systems form the backbone of much of the world's IT infrastructure, from enterprise servers and cloud platforms to embedded systems. With increasing system complexity and the rapid pace of IT service demands, manual administration is no longer scalable or efficient. Routine tasks such as user management, disk monitoring, software installation, backup scheduling, and log analysis must be carried out accurately and swiftly. This is where automation becomes indispensable (Gill et al., 2024).

In Unix/Linux environments, scripting has long been a powerful method for automation. Shell scripts, in particular, allow administrators to chain together standard utilities, manage system processes, and configure system settings with ease. Over the years, the scripting landscape has evolved to include higher level languages like Python and Perl, which offer greater flexibility, readability, and integration with modern APIs and tools. The ability to automate repetitive and error prone tasks not only increases productivity but also enhances system reliability, security, and maintainability (Wang 2019).



Concept of Unix/Linux system administration and automation

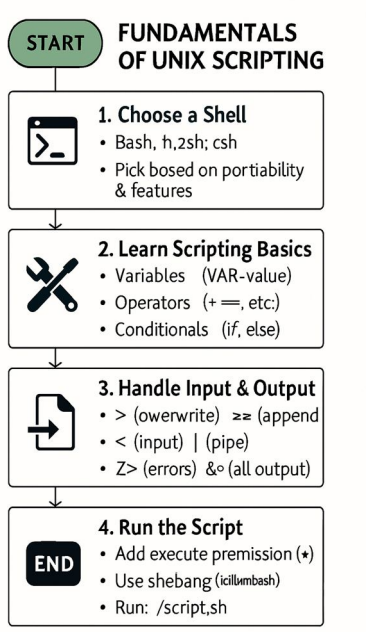
B. Purpose of the Review

The purpose of this review is to examine foundational scripting techniques that are essential for effective Unix system administration. Rather than offering an exhaustive tutorial or reference manual, this review focuses on core concepts and practical applications of scripting for automation. By revisiting fundamental techniques, both new and experienced administrators can better understand how to build efficient and reusable scripts, adopt best practices, and leverage the power of Unix's modular architecture (Gift and Jones 2008). This review also aims to highlight how scripting underpins more advanced automation frameworks. While modern tools like Ansible, Puppet, and Chef have gained popularity for configuration management and orchestration, their underlying logic often traces back to scripting fundamentals. Understanding these basics is therefore a prerequisite for mastering higher level automation and integrating various tools in a DevOps pipeline.

C. Scope

This study concentrates primarily on shell scripting particularly Bash, the most widely used shell in Unix/Linux environments. It covers the syntax, structures, and typical use cases such as conditionals, loops, file manipulation, and process control. Emphasis is placed on writing clean, modular, and maintainable code, with examples that illustrate real world administrative tasks (Pakin 2024). In addition to shell scripting, brief overviews of Python and Perl are provided. These languages have been staples in Unix scripting due to their powerful text processing capabilities and extensive libraries. Python, in particular, has become increasingly favored for its readability and strong community support. Perl continues to be useful in legacy systems and for quick text processing tasks. The review also touches on basic automation tools and utilities such as cron, systemd timers, and job scheduling, which are integral to executing scripts in a timed or event driven fashion (Andelkovic et al., 2020).

II. FUNDAMENTALS OF UNIX SCRIPTING



Flow Chart: Fundamentals of Unix Scripting

A. Overview of Shells

In Unix based systems, the "shell" is the command line interpreter that enables users to interact with the operating system. There are several types of shells, each offering unique features. Bash (Bourne Again Shell) is the most widely used and is the default on many Linux distributions (Kidwai et al., 2021). It supports command history, scripting capabilities, command line editing, and job control. Zsh (Bourne Shell) is the original Unix shell and is known for its portability and simplicity many scripts use it for maximum compatibility. zsh is a feature rich shell favored by power users for its advanced auto completion, plugin support, and themes, especially when combined with tools like Oh My Zsh (Fadhilah and Adrian 2023).

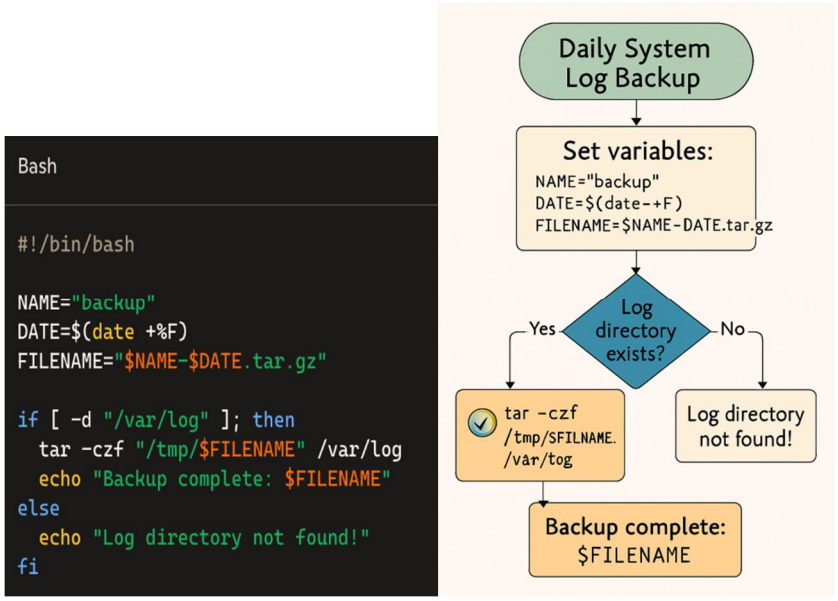
Other shells include csh (C Shell), which resembles the C programming language syntax, and ksh (KornShell), known for scripting enhancements. While all shells interpret commands, their syntax and behavior can differ (Offutt 2011). Understanding the nuances of each shell helps administrators choose the right one for scripting or interactive use. In scripting contexts, Bash is often preferred due to its balance of power and ease of use, while sh is ideal for scripts intended to run on multiple Unix like systems. Choosing the right shell depends on the task, environment, and portability needs (Liu et al., 2011).

B. Shell Scripting Basics

Shell scripting allows Unix administrators to automate tasks by writing sequences of commands in text files. These scripts use a structured syntax that includes variables, operators, conditionals, and loops. Variables in shell scripts are defined without spaces (e.g., VAR=value) and accessed using a dollar sign ($VAR). They allow storage of temporary data like filenames or command output (Platt 2020).

Operators such as =, eq, ne, lt, and gt are used for arithmetic comparisons, while string comparisons use operators like =, !=, and z. Conditional structures (if, then, else, elif) let scripts make decisions. Loops, including for, while, and until, enable repetitive tasks like iterating over files or monitoring processes (Liu et al., 2016)

For example, a basic loop might check the size of a directory every minute and log the result. Scripts can also call external commands, redirect outputs, and use built in functions to organize code. Commenting with # is crucial for documentation and maintainability. By mastering these basics, administrators can write powerful scripts that perform everything from system checks to software deployment efficiently (Ranjan et al., 2024).



Unix scripting concepts from shells

C. Input/Output Redirection

Input/output redirection is a cornerstone of Unix scripting, enabling the manipulation and chaining of data streams. The redirection symbols >, >>, <, and | are used to control where input comes from and where output goes. The > operator redirects standard output to a file, overwriting its contents. For example, ls > filelist.txt stores the output of ls into filelist.txt. In contrast, >> appends the output to the file instead of overwriting it, which is useful for logs or cumulative reports (Jain 2018).

The < operator redirects input from a file to a command. For instance, sort < names.txt reads names.txt and sorts its content. Perhaps most powerful is the pipe operator |, which connects the output of one command to the input of another. This enables complex data processing chains, such as ps aux | grep apache | wc l to count Apache processes (Ahmad et al., 2022)

These operators also support more advanced use cases, such as redirecting standard error (2>) and combining standard output and error (&>). Understanding redirection allows administrators to write efficient scripts that process logs, manage backups, and automate diagnostics with precision and clarity (Derrouazin et al., 2017)

D. Permissions and Execution

Unix enforces strict permission models to ensure system security, which extends to script execution. Before a shell script can be run, it must have the executable permission. This is set using the chmod command for example, chmod +x script.sh makes the script executable. Without this, a script cannot be launched directly from the terminal (Yuranda and Negara 2024)

Equally important is the shebang line, written as #!/bin/bash (or another shell path) at the top of the script. This tells the system which interpreter to use when executing the script. Omitting or misconfiguring the shebang can result in execution errors or unexpected behavior, especially if scripts rely on features specific to a shell (e.g., Bash arrays or Zsh globbing) (Zhang et al., 2013)

Scripts must also be placed in executable directories or referenced with a path (./script.sh) to avoid "command not found" errors. File permissions (rwx) must be properly set to control who can read, write, or execute the script. Misconfigured permissions can lead to security risks or unintentional data exposure. Mastery of these fundamentals ensures that scripts are safely and correctly executed, forming the basis of reliable system automation (Islavath 2020).

III. CORE ADMINISTRATIVE TASKS AND SCRIPTING EXAMPLES

A. User Management

Managing user accounts is one of the most fundamental administrative tasks in Unix based systems. Automating the creation, deletion, and modification of users helps system administrators enforce consistent access policies and save time, especially when managing multiple users or servers. Scripts can be written to create users in bulk, set up their home directories, assign groups, and enforce password policies (like expiration dates or complexity rules) (Kandogan et al., 2009).

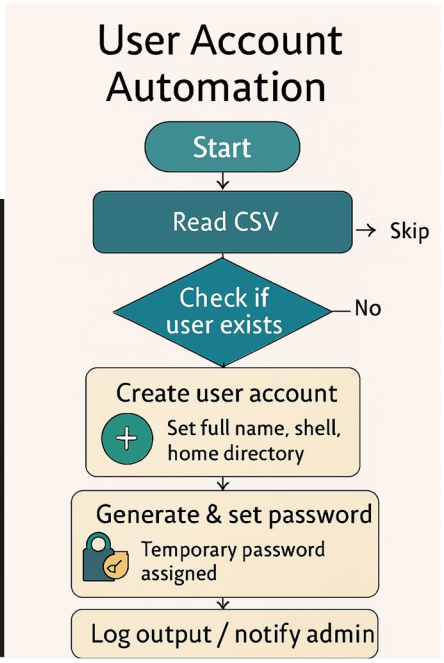
For example, using a simple Bash script, administrators can read a CSV file containing usernames and automatically create each account with predefined parameters. This process can also include generating temporary passwords and forcing password changes on first login. Deletion scripts are similarly useful for deprovisioning users, especially in enterprise environments with high staff turnover or project based access (Michael 2013).

Example (Bulk User Creation in Bash):

```
#!/bin/bash

# Assumes a CSV file: users.csv with "username,fullname" format

while IFS=',' read -r username fullname; do
  if id "$username" &>/dev/null; then
    echo "User $username already exists. Skipping..."
  else
    useradd -m -c "$fullname" -s /bin/bash "$username"
    PASSWORD=$(openssl rand -base64 10)
    echo "$username:$PASSWORD" | chpasswd
    passwd -e "$username"  # Force password change at first login
    echo "Created user: $username with temp password: $PASSWORD"
  fi
done < users.csv
```



Sample Bash Script for Bulk User Creation

This approach ensures security and consistency, reduces human error, and makes user management scalable and auditable.

B. File and Directory Operations

Unix administrators often need to manage files and directories efficiently this includes tasks like scheduled backups, log rotation, and periodic cleanup. These actions, when automated, contribute significantly to system reliability and data integrity. Scripts can be scheduled using cron or systemd timers to back up specific directories to a safe location (local or remote) using tools like rsync or tar. For instance, a nightly script might compress and store /etc/ or /var/log/ to a backup server.

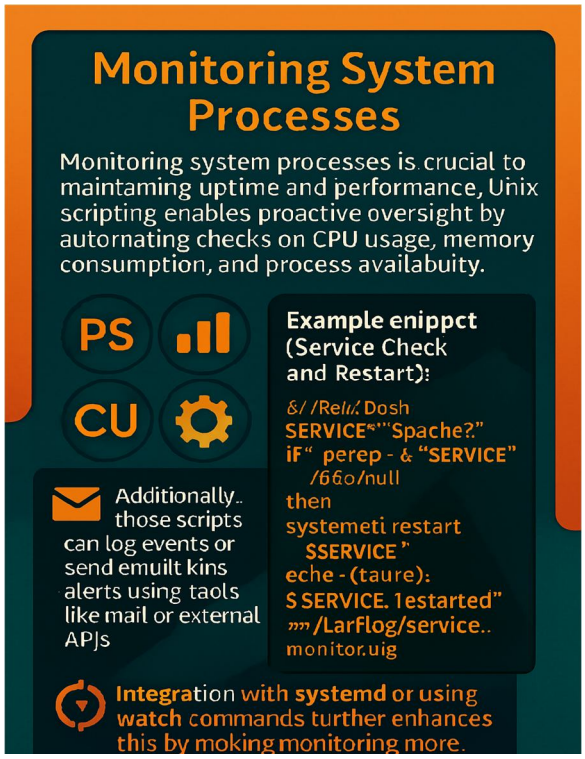
Likewise, log rotation is critical to prevent disk space exhaustion. While logrotate is often used, custom scripts are helpful in cases where third party or legacy applications generate logs (Frisch 2002).

Cleanup scripts help remove old or temporary files based on age or size thresholds. For example, a script can find and delete files older than 30 days from /tmp or /var/tmp to maintain free disk space. These scripts ensure routine maintenance is handled without constant human intervention, reducing operational risk and ensuring that critical files are preserved and redundant data is cleared in a timely fashion (Akin et al., 1987).

C. Process Monitoring and Control

Monitoring system processes is crucial to maintaining uptime and performance. Unix scripting enables proactive oversight by automating checks on CPU usage, memory consumption, and process availability. Admins often use ps, top, free, and uptime commands within scripts to gather data. For example, a script can check whether a service like Apache or MySQL is running and restart it automatically if it crashes (Glass 2004). Additionally, these scripts can log events or send email/SMS alerts using tools like mail or external APIs. Integration with systemd or using watch commands further enhances this by making monitoring more continuous and less resource intensive.

Example snippet (Service Check and Restart):



A visual concept of Unix scripting

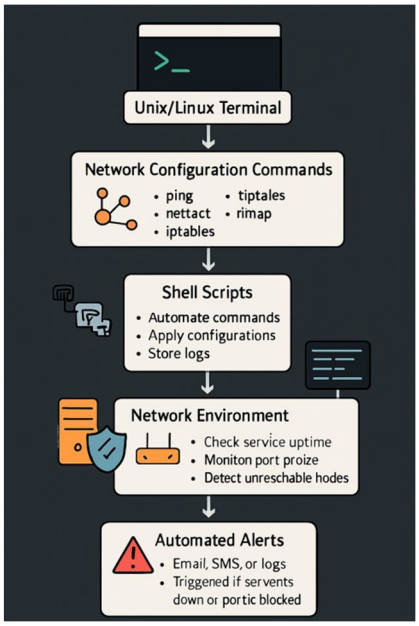
Such process monitoring scripts are often scheduled via cron every few minutes or wrapped into larger observability systems. They reduce downtime and enable early detection of issues, particularly in environments lacking full scale monitoring solutions.

D. Network Configuration and Monitoring

Network related tasks are essential for both performance and security. Unix scripts can assist in everything from checking connectivity to configuring firewall rules. Simple tools like ping, netstat, ss, ip, iptables, and nmap are commonly embedded in scripts to test if services are reachable or if specific ports are open (Jose and Shenoy 2024). Scripts can periodically ping critical servers and alert admins if any are down. They can also validate that certain ports are accepting connections, which is essential when troubleshooting web servers, SSH access, or application endpoints. In terms of configuration, shell scripts can automate the setup of firewall rules using iptables or ufw, which is especially useful when deploying multiple servers with similar security policies. These scripts can also be triggered during server provisioning or instance startup (Zhou et al., 2018).

Example:

```
#!/bin/bash
for HOST in server1.com server2.com
do
  if ! ping -c 2 "$HOST" > /dev/null; then
    echo "$HOST is down!" >> /var/log/network_check.log
  fi
done
```



Network configuration and monitoring via Unix scripting

E. Software Installation and Updates

Software management is a recurring administrative task that benefits greatly from scripting. Unix systems often use package managers like apt (Debian/Ubuntu), yum or dnf (RHEL/CentOS/Fedora), and zypper (SUSE) to install and update software. Scripts can streamline the installation of multiple packages, apply updates, or enforce version consistency across servers (Kan et al., 2025). For example, a provisioning script may install a full LAMP stack (Linux, Apache, MySQL, PHP) with a single command batch. Similarly, update scripts can be scheduled weekly to install security patches, reducing vulnerability exposure.

Example (Install Packages on Ubuntu):

→

```
#!/bin/bash
PACKAGES="nginx mysql-server php"
apt update
apt install -y $PACKAGES
```

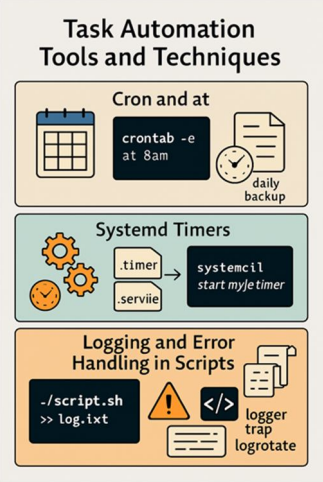
Steps to automate installation of a LAMP stack using a Bash script

IV. TASK AUTOMATION TOOLS AND TECHNIQUES

Such scripts ensure repeatability and can be version controlled for auditing. Combined with configuration management tools or templates, they enable administrators to deploy new servers rapidly with minimal manual effort, ensuring systems remain secure and standardized. In Unix and Linux systems, automating administrative tasks is a key aspect of efficient system management (Leitch and Stefanini 1989). Among the foundational tools for task automation are cron and at. The cron utility is used for scheduling recurring tasks, such as daily backups, system updates, or log cleanups. It operates through the crontab file, where each entry follows a specific time based syntax (Wali et al., 2023). For example, a line like 0 2 * * * /usr/local/bin/backup.sh schedules a script to run daily at 2:00 AM. Each user can maintain a personal crontab, and system wide tasks are typically managed under /etc/crontab or in the /etc/cron.d/ directory. On the other hand, the at command is designed for one time task execution at a specified time, such as scheduling a one time reboot using echo "reboot" | at 3am (Samad and Cofer 2021).

In more modern Linux distributions, systemd timers offer a robust alternative to cron. They are closely integrated with system services and provide advanced features like dependency handling and persistent scheduling. A timer is defined using a .timer file that corresponds to a .service file. For instance, a timer configured with OnCalendar=daily and Persistent=true will ensure that the corresponding service runs once per day and makes up for any missed executions, such as during downtime. These timers are started and enabled using standard systemd commands like systemctl start and systemctl enable (Fry and Potter 2018).

Effective logging and error handling are also crucial in automation scripting. Tools like logger allow scripts to send output directly to system logs, aiding in debugging and auditing. Output and error streams can be redirected to custom log files using syntax like >> /path/to/logfile 2>&1. For error control, the trap command can catch signals or script errors and execute cleanup routines, ensuring graceful failure handling (Zhong et al., 2025).



Task automation tools and techniques in Unix/Linux systems

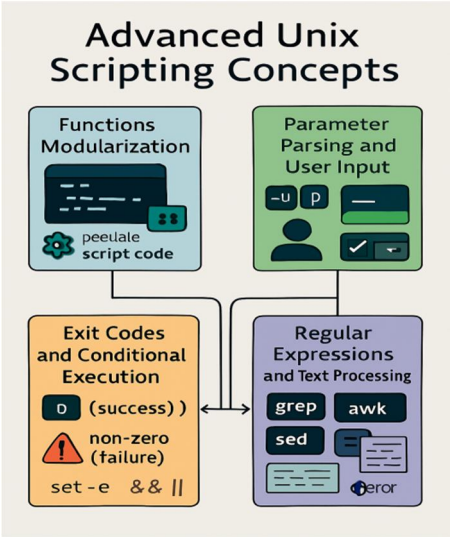
V. ADVANCED SCRIPTING CONCEPTS

In advanced Unix scripting, modularization is a key practice that promotes reusability, clarity, and maintainability. This is achieved by organizing code into functions reusable blocks of logic that perform specific tasks. Functions help avoid redundancy, isolate logic for easier debugging, and improve readability. For instance, a function named backup_files could be written once and used whenever a backup is needed, rather than repeating the same code multiple times. Scripts can include multiple functions grouped logically or even source external function libraries using the source or . command, making complex scripts more manageable and scalable (Nascenzi et al., 2025).

Parameter parsing and handling user input is another advanced scripting concept that greatly improves the flexibility of a script. Tools like getopts are used to handle command line options in a clean and user friendly manner. This allows a script to accept flags like ‘ u’ for username or ‘f’ for filename, making automation more interactive and customizable. Additionally, validating input such as checking for empty fields, ensuring values follow expected formats, or asking for confirmation helps prevent errors and ensures the script behaves as intended in various environments (Amant and Giordano 2023).

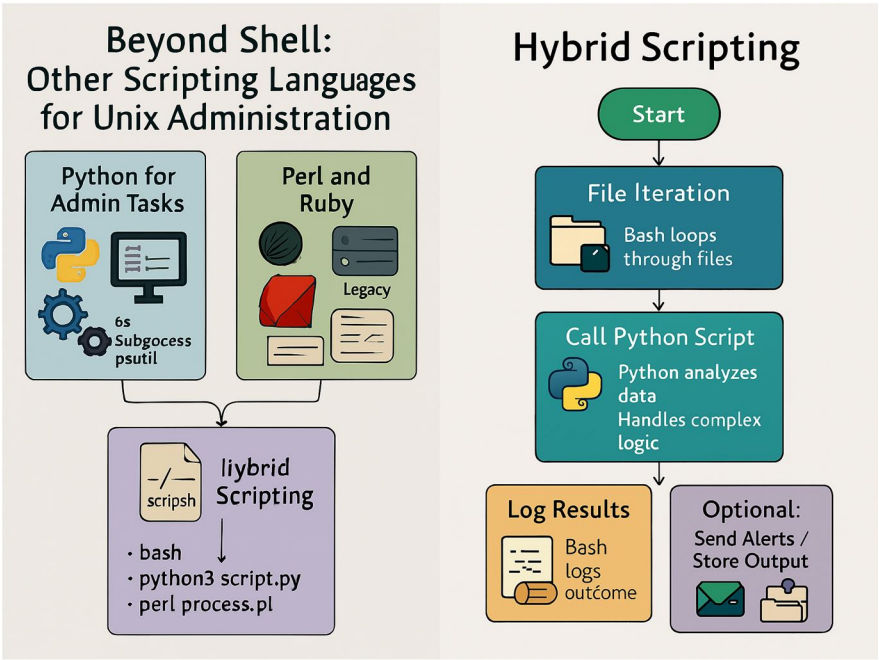
Understanding exit codes and using them effectively is fundamental for building reliable and robust scripts. Every command in Unix returns an exit status: 0 indicates success, while non zero values indicate different types of failure. Good scripting practice involves checking these exit statuses using $? or conditional execution with && and || operators. For example, a script might copy a file and only proceed if the copy was successful. Including set e at the top of a script can force it to terminate on any command failure, which is useful for automation tasks where silent failures must be avoided (Wu et al., 2024).

Lastly, regular expressions and text processing tools are vital for advanced scripting. Commands like grep, awk, and sed enable powerful text manipulation capabilities. grep is used to search for patterns, awk can process text based on delimiters and perform calculations, and sed is used for in place editing of files or streams. Together, these tools allow scripts to automate the parsing, filtering, and transformation of data from logs, configuration files, or command outputs making them indispensable for system administrators and power users (Hofmann et al., 2023).



Flowchart of Advanced Unix Scripting Concepts

A. Python for Admin Tasks



Beyond Shell: Other Scripting Languages for Unix Administration

Python is widely adopted in Unix system administration due to its readability, extensive standard library, and cross platform compatibility. For tasks that require more complex logic or integration with APIs, Python offers a more structured and powerful approach than shell scripting. The os and subprocess modules allow administrators to interact with the operating system and execute shell commands. For example, os.system() can run commands, while subprocess.run() offers better error handling and output management. The psutil library is another valuable tool that provides real time information on CPU usage, memory, disk partitions, and processes something that would require multiple shell commands and parsing if done in Bash (Tansley 2011). Python is particularly beneficial for tasks involving file manipulation, JSON or XML parsing, and automation involving web services or databases. It handles data structures and exceptions more robustly than shell scripts, making it ideal for medium to large scale automation projects. While shell scripts are faster for quick, one liner tasks or managing Unix native operations, Python excels when complexity increases. Choosing Python over shell scripting is advisable when maintainability, portability, or integration with external systems is a concern (Joy et al., 2002).

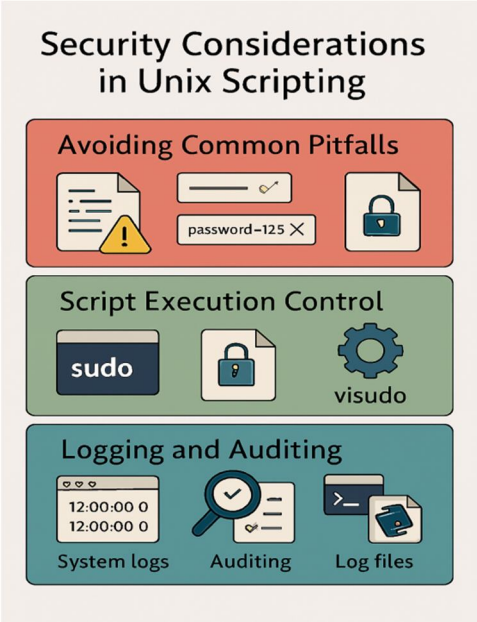
B. Perl and Ruby

Before Python became dominant, Perl and Ruby were the go to scripting languages for Unix administrators, especially for text processing and web deployment scripts. Perl, often described as the "duct tape of the Internet," excels at regular expressions and complex pattern matching, making it ideal for parsing logs, transforming data, and automating tasks involving large volumes of text (Liu et al., 2014). Perl's CPAN repository contains thousands of modules tailored for networking, system interaction, and more, giving it wide ranging capabilities even today. Although its syntax can be dense, Perl remains in use in legacy systems and long established infrastructure. Ruby, known for its elegant syntax and object oriented approach, became popular in the early 2000s with frameworks like Ruby on Rails. In system administration, Ruby is used for configuration management tools like Chef, and some sysadmins still favor it for custom tools (Thomas et al., 2009).

C. Hybrid Scripting

Hybrid scripting involves combining multiple scripting languages within the same automation workflow. This is often done to leverage the strengths of each language. For instance, a shell script might manage high level orchestration like scheduling or file I/O while delegating complex processing or external API calls to a Python or Perl script. This method allows administrators to use the right tool for each task. A typical example might involve using a Bash script to iterate over a directory of files and calling a Python script to analyze their contents, then logging the results using native Linux tools like logger (Haija 2023).

VI. SECURITY CONSIDERATIONS



Security Considerations in Unix Scripting

Security begins with proper input validation and avoiding risky practices like hardcoding credentials. Scripts that accept user input must sanitize and validate that input to prevent command injection or unintended system behavior. For instance, failing to quote variables like $1 in rm $1 can lead to catastrophic results if special characters or wildcards are passed. It's also crucial to avoid placing usernames, passwords, or tokens directly in scripts (Beuchelt 2017). Instead, use environment variables, secure vaults, or encrypted files with strict access controls. External configuration files should also have proper file permissions to prevent unauthorized reading. Additionally, error messages and logs should not reveal sensitive information such as directory structures or authentication failures. Ensuring that scripts fail safely without exposing systems or data is a key part of writing secure, production ready automation (Pradhan 2018). Controlling who can execute administrative scripts and how they do so is a cornerstone of secure scripting. Misuse of powerful scripts can lead to privilege escalation or system compromise. Always restrict execution to only trusted users or groups using Unix permissions (chmod, chown) and directory access controls. When scripts require root privileges, use sudo in combination with visudo to define fine grained access policies (Viega and Voas 2000). This allows specific users to run scripts with elevated privileges without granting full root access. For example, visudo rules can restrict a user to running /usr/local/bin/backup.sh only. Never run scripts as root unless absolutely necessary. Additionally, include checks within the script to detect if it's being executed by the correct user, and abort otherwise. Protecting execution environments and avoiding writable paths in PATH can also help mitigate the risk of script hijacking or privilege abuse (Santana 2025).

VII. CASE STUDIES AND REAL WORLD SCENARIOS

In modern Unix administration, real world scripting use cases demonstrate the power and necessity of automation. One key scenario is automating daily health checks. Administrators often use scripts to gather system metrics such as CPU load, memory usage, disk space, and active processes then compile this data into reports or dashboards. These scripts can be scheduled via cron or systemd timers and set to alert administrators via email or Slack when thresholds are breached (Wang 2010). Another critical use case is provisioning new servers, where scripts handle tasks such as setting hostnames, configuring SSH keys, installing required software, setting up firewall rules, and applying baseline configurations. By using templated scripts or tools like Ansible or cloud init, new servers can be brought online consistently and quickly. Additionally, in disaster recovery and backup planning, scripting plays a central role. Scripts can automate the creation of system snapshots, database dumps, and file backups, then move those securely to offsite or cloud storage (Mehra 2025). They can also validate backups and clean up outdated archives to save space. These scripted processes reduce recovery time, improve reliability, and ensure compliance with data protection policies. Together, these scenarios underscore how foundational scripting techniques contribute to system stability, efficiency, and resilience in real world operations (Michael 2013).

VIII. CHALLENGES AND FUTURE TRENDS

While shell scripting remains a foundational skill in Unix administration, it comes with several limitations. Scripts can quickly become hard to maintain as they grow in size or complexity, especially without clear modularization or documentation. Debugging can be cumbersome due to limited tooling and unstructured error reporting. Moreover, shell scripts may not handle edge cases or input errors gracefully, increasing the risk of unintended consequences in production environments (Liargkovas et al., 2023). In response to these challenges, the industry has embraced configuration management tools like Ansible, Puppet, and Chef. These tools enable administrators to define infrastructure and application states declaratively, promoting consistency, scalability, and repeatability. They reduce manual intervention and integrate well with CI/CD pipelines, making them ideal for modern DevOps practices (Rahman et al., 2021). Looking further ahead, AI and intelligent automation are beginning to shape the future of Unix administration. Tools powered by machine learning can now perform tasks like anomaly detection, predictive maintenance, and automated remediation. Early implementations include AI driven alerting systems that adjust thresholds based on usage patterns or bots that recommend or even execute corrective actions. While still evolving, these technologies point toward a future where routine administrative work is not just automated but optimized and adaptive in real time (Kaur et al., 2023).

IX. CONCLUSION

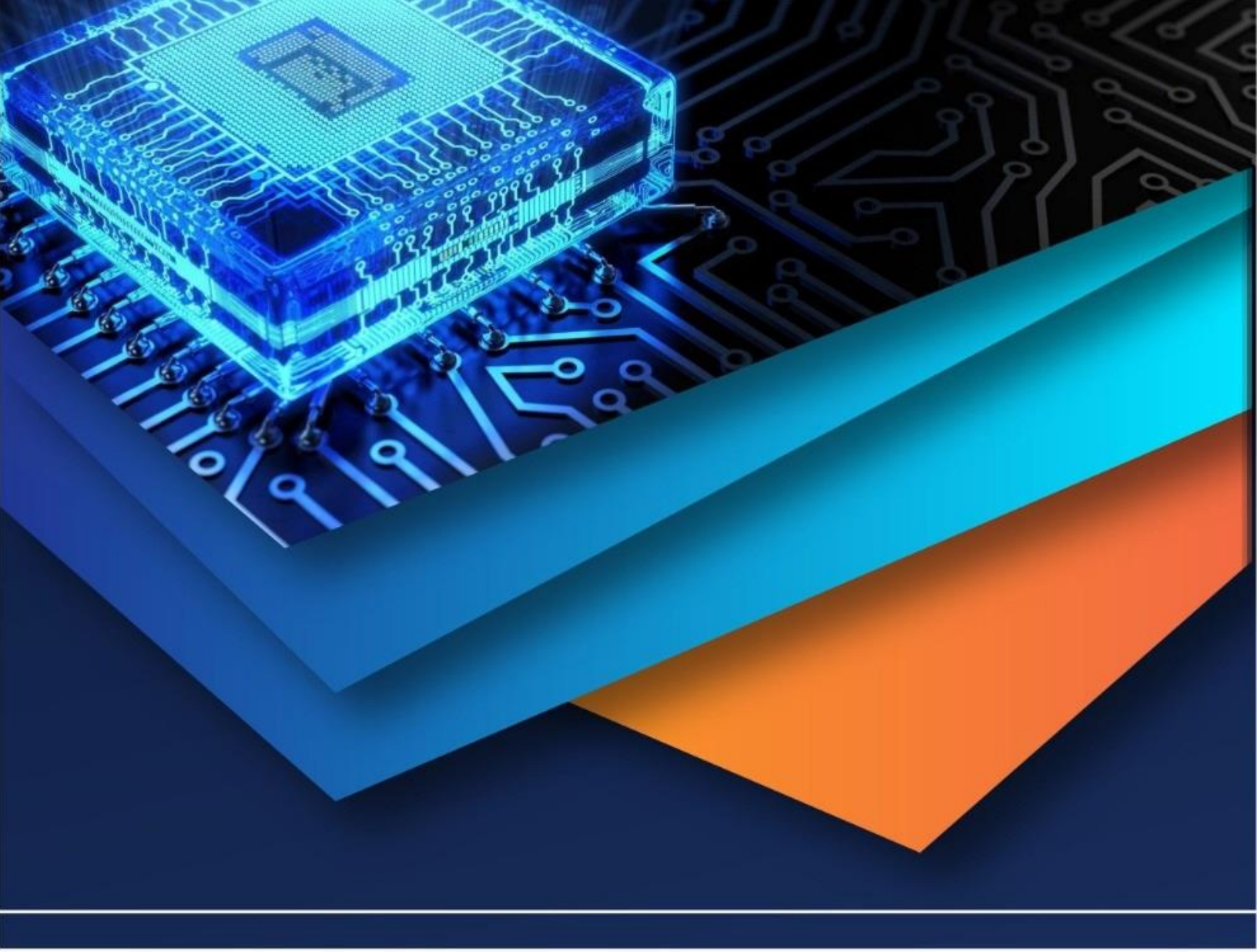
In conclusion, scripting remains a vital pillar of Unix system administration, offering administrators the power to automate, standardize, and simplify complex workflows. This review has highlighted key scripting techniques from shell basics like loops, variables, and conditionals to advanced concepts such as modular functions, input parsing, logging, and error handling. These foundational skills are essential for tackling real world tasks like user management, software provisioning, system monitoring, and backups.

In modern sysadmin workflows, scripting is not just a productivity tool it's a necessity for managing scalable infrastructure, reducing human error, and ensuring consistent operations across servers and environments. As infrastructure grows in size and complexity, the role of scripting expands into orchestration, CI/CD pipelines, and hybrid cloud automation. To thrive in this evolving landscape, system administrators should build fluency in both shell scripting and complementary languages like Python, while also gaining familiarity with tools like Ansible, cron, systemd timers, and log managers. Hands on practice through lab environments, open source contributions, or real world problem solving is the most effective way to master these skills. As the field moves toward intelligent automation and predictive systems, a strong scripting foundation will remain a key asset for any IT professional seeking to stay relevant and efficient.

REFERENCES

[1] Abu Al Haija, Q. (2023). Cost effective detection system of cross site scripting attacks using hybrid learning approach. Results in Engineering, 19, 101266. https://doi.org/10.1016/j.rineng.2023.101266
[2] Ahmad, T., Ma, C., Al Ars, Z., & Hofstee, H. P. (2022a). Communication efficient cluster scalable genomics data processing using Apache Arrow Flight. 2022 21st International Symposium on Parallel and Distributed Computing (ISPDC), 138–146. https://doi.org/10.1109/ispdc55340.2022.00028
[3] Ahmad, T., Ma, C., Al Ars, Z., & Hofstee, H. P. (2022b). Communication efficient cluster scalable genomics data processing using Apache Arrow Flight. 2022 21st International Symposium on Parallel and Distributed Computing (ISPDC), 138–146. https://doi.org/10.1109/ispdc55340.2022.00028
[4] Akin, O., Baykan, C., & Rao, D. R. (1987). Structure of a directory space: A case study with a unix operating system. International Journal of Man Machine Studies, 26(3), 361–382. https://doi.org/10.1016/s0020 7373(87)80069 x
[5] Andelkovic, A., Hausknecht, K., & Sirovatka, G. (2020). Linux forensic triage: Overview of process and Tools. 2020 43rd International Convention on Information, Communication and Electronic Technology (MIPRO), 1230–1235. https://doi.org/10.23919/mipro48935.2020.9245304
[6] Beuchelt, G. (2017). Unix and linux security. Computer and Information Security Handbook, 197–216. https://doi.org/10.1016/b978 0 443 13223 0.00011 4
[7] Derrouazin, A., Aillerie, M., Mekkakia Maaza, N., & Charles, J. P. (2017). Multi input output fuzzy logic smart controller for a residential hybrid solar wind storage energy system. Energy Conversion and Management, 148, 238–250. https://doi.org/10.1016/j.enconman.2017.05.046
[8] Fadhilah, F., & Adrian, R. (2023). Implementasi Modul Otomatisasi penetration testing Menggunakan Bourne again shell scripting Pada website aplikasi stream pt. Intikom Berlian Mustika Berbasis Kali Linux. Jurnal Sistem Dan Teknologi Informasi (JustIN), 11(3), 554. https://doi.org/10.26418/justin.v11i3.67468
[9] Frisch, A. (2002). Essential System Administration Tools and techniques for linux and unix administration Aeleen Frisch. Ed.: Michael Loukides. O'Reilly.
[10] Fry, C., & Potter, S. (n.d.). Interdisciplinary Embedded Systems Design: integrating hardware oriented embedded systems design with software oriented embedded systems development. 2018 ASEE Annual Conference & Exposition Proceedings. https://doi.org/10.18260/1 2 30700
[11] Gift, N., & Jones, J. M. (2008). Python for unix and linux system administration efficient problem solving with python. O'Reilly.
[12] Gill, S. S., Wu, H., Patros, P., Ottaviani, C., Arora, P., Pujol, V. C., Haunschild, D., Parlikad, A. K., Cetinkaya, O., Lutfiyya, H., Stankovski, V., Li, R., Ding, Y., Qadir, J., Abraham, A., Ghosh, S. K., Song, H. H., Sakellariou, R., Rana, O., … Buyya, R. (2024). Modern computing: Vision and challenges. Telematics and Informatics Reports, 13, 100116. https://doi.org/10.1016/j.teler.2024.100116
[13] Glass, M. (2004). Beginning PHP, Apache, MySQL Web Development. Wiley.
[14] Hofmann, M., Albaugh, L., Wang, T., Mankoff, J., & Hudson, S. E. (2023a). KnitScript: A domain specific scripting language for advanced machine knitting. Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology, 1–21. https://doi.org/10.1145/3586183.3606789
[15] Hofmann, M., Albaugh, L., Wang, T., Mankoff, J., & Hudson, S. E. (2023b). KnitScript: A domain specific scripting language for advanced machine knitting. Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology, 1–21. https://doi.org/10.1145/3586183.3606789
[16] Islavath, N. (2020). The Power of Docker: Containerization for efficient software development and deployment. International Journal of Science and Research (IJSR), 9(11), 1748–1751. https://doi.org/10.21275/sr201226085354
[17] Jain, M. (2018). Advanced techniques in Shell scripting. Beginning Modern Unix, 283–312. https://doi.org/10.1007/978 1 4842 3528 7_10
[18] Jose, J., & Shenoy, G. S. (2024). An efficient framework for integrating devops practices in Network configuration and monitoring. 2024 3rd International Conference for Innovation in Technology (INOCON), 1–6. https://doi.org/10.1109/inocon60754.2024.10512008
[19] Joy, M., Jarvis, S., & Luck, M. (2002). Advanced shell programming. Introducing UNIX and Linux, 173–193. https://doi.org/10.1007/978 0 230 80245 2_9
[20] Kan, V., Lnu, M. P., Berhe, S., El Kari, C., Maynard, M., & Khomh, F. (2025a). Automated UML visualization of software ecosystems: Tracking versions, Dependencies, and security updates. Procedia Computer Science, 257, 834–841. https://doi.org/10.1016/j.procs.2025.03.107
[21] Kan, V., Lnu, M. P., Berhe, S., El Kari, C., Maynard, M., & Khomh, F. (2025b). Automated UML visualization of software ecosystems: Tracking versions, Dependencies, and security updates. Procedia Computer Science, 257, 834–841. https://doi.org/10.1016/j.procs.2025.03.107
[22] Kandogan, E., Maglio, P. P., Haber, E. M., & Bailey, J. H. (2009). Scripting practices in complex systems management. Proceedings of the Symposium on Computer Human Interaction for the Management of Information Technology, 9–18. https://doi.org/10.1145/1641587.1641589
[23] Kaur, R., Gabrijelčič, D., & Klobučar, T. (2023). Artificial Intelligence for cybersecurity: Literature review and future research directions. Information Fusion, 97, 101804. https://doi.org/10.1016/j.inffus.2023.101804
[24] Kidwai, A., Arya, C., Singh, P., Diwakar, M., Singh, S., Sharma, K., & Kumar, N. (2021). A comparative study on shells in linux: A Review. Materials Today: Proceedings, 37, 2612–2616. https://doi.org/10.1016/j.matpr.2020.08.508
[25] Leitch, R., & Stefanini, A. (1989). Task dependent tools for intelligent automation. Artificial Intelligence in Engineering, 4(3), 126–143. https://doi.org/10.1016/0954 1810(89)90009 5
[26] Liargkovas, G., Kallas, K., Greenberg, M., & Vasilakis, N. (2023). Executing shell scripts in the wrong order, correctly. Proceedings of the 19th Workshop on Hot Topics in Operating Systems, 103–109. https://doi.org/10.1145/3593856.3595891

[27] Liu, W., Islamaj Do an, R., Kwon, D., Marques, H., Rinaldi, F., Wilbur, W. J., & Comeau, D. C. (2014). BIOC implementations in go, Perl, python and ruby. Database, 2014(0). https://doi.org/10.1093/database/bau059
[28] Liu, X., Jiang, Y., Wu, L., & Wu, D. (2016). Natural shell. International Journal of People Oriented Programming, 5(1), 1–18. https://doi.org/10.4018/ijpop.2016010101
[29] Liu, Y., Yue, Y., & Guo, L. (2011). Unix shell introduction. UNIX Operating System, 229–243. https://doi.org/10.1007/978 3 642 20432 6_8
[30] Lochan Pradhan, P. (2018). Role of scripting language on unix operating system for risk assessment. International Journal of Computer Network and Information Security, 10(9), 47–59. https://doi.org/10.5815/ijcnis.2018.09.05
[31] Mehra, T. (2025). Linux administration for managing large infrastructure: A practical approach to real time deployment and research publication. INTERANTIONAL JOURNAL OF SCIENTIFIC RESEARCH IN ENGINEERING AND MANAGEMENT, 09(02), 1–9. https://doi.org/10.55041/ijsrem41156
[32] Michael, R. K. (2013a). Mastering unix shell scripting: Bash, Bourne, and Korn shell scripting for programmers, system administrators, and unix gurus. Wiley.
[33] Michael, R. K. (2013b). Mastering unix shell scripting: Bash, Bourne, and Korn shell scripting for programmers, system administrators, and unix gurus. Wiley.
[34] Nascenzi, T., Cuatt, T., & McDonald, R. A. (2025). Advanced scripting development and application of openvsp. AIAA SCITECH 2025 Forum. https://doi.org/10.2514/6.2025 0372
[35] Offutt, J. (2011). A mutation Carol: Past, present and future. Information and Software Technology, 53(10), 1098–1107. https://doi.org/10.1016/j.infsof.2011.03.007
[36] Pakin, S. (2014). MPI Bash: Parallel Scripting Right from the Bourne Again Shell (Bash). https://doi.org/10.2172/1131016
[37] Platt, D. (2020). Shell scripting basics. Tweak Your Mac Terminal, 305–339. https://doi.org/10.1007/978 1 4842 6171 2_5
[38] Rahman, A., Rahman, M. R., Parnin, C., & Williams, L. (2021). Security smells in Ansible and Chef Scripts. ACM Transactions on Software Engineering and Methodology, 30(1), 1–31. https://doi.org/10.1145/3408897
[39] Ranjan, M. Kr., Saxena, S., Gupta, R., Singh, A., Anarthe, A. S., & Anand, A. (2024). Optimizing System Management: Innovative approaches to Shell scripting for automation in large scale systems. Journal of Advances in Shell Programming. https://doi.org/10.37591/joasp.v11i03.180756
[40] Samad, T., & Cofer, D. (2001). Autonomy in automation: Trends, technologies, Tools. Computer Aided Chemical Engineering, 1–13. https://doi.org/10.1016/s1570 7946(01)80002 x
[41] Santana, M. D. (2025). Eliminating the security weakness of linux and unix operating systems. Computer and Information Security Handbook, 217–233. https://doi.org/10.1016/b978 0 443 13223 0.00012 6
[42] St.Amant, K., & Giordano, W. (2023). Expanding communication expectations: Examining audience understanding of scripts through fold and swap strategies. Journal of Technical Writing and Communication, 55(1), 58–78. https://doi.org/10.1177/00472816231216911
[43] Tansley, D. (2011). Linux and unix shell programming. Addison Wesley.
[44] Thomas, D., Fowler, C., & Hunt, A. (2009). Programming ruby. Pragmatic.
[45] Viega, J., & Voas, J. (2000). The Pros and cons of unix and windows security policies. IT Professional, 2(5), 40–47. https://doi.org/10.1109/6294.877496
[46] Wali, A., Mahamad, S., & Sulaiman, S. (2023). Task Automation Intelligent Agents: A Review. Future Internet, 15(6), 196. https://doi.org/10.3390/fi15060196
[47] WANG, K. C. (2019). Systems programming in unix. SPRINGER.
[48] Wang, P. S. (2010). Mastering Linux. https://doi.org/10.1201/9781439894750
[49] Wu, M. H., Hsu, F. H., Hunag, J. H., Wang, K., Liu, Y. Y., Chen, J. X., Wang, H. J., & Yang, H. T. (2024). MPSD: A robust defense mechanism against malicious PowerShell scripts in Windows Systems. Electronics, 13(18), 3717. https://doi.org/10.3390/electronics13183717
[50] Yuranda, R., & Negara, E. S. (2024). Application of deep learning algorithm for web shell detection in web application security system. Jurnal Sisfokom (Sistem Informasi Dan Komputer), 13(3), 330–336. https://doi.org/10.32736/sisfokom.v13i3.2234
[51] Zhang, S., Dong, Y., Liu, B., & Gu, J. (2013). Content management system of website group based on Zsh Frame. Proceedings of the International Conference on Computer, Networks and Communication Engineering (ICCNCE 2013). https://doi.org/10.2991/iccnce.2013.62
[52] Zhong, R., Li, Y., Kuang, J., Gu, W., Huo, Y., & Lyu, M. R. (2025). Logupdater: Automated detection and repair of specific defects in logging statements. ACM Transactions on Software Engineering and Methodology. https://doi.org/10.1145/3731754
[53] Zhou, G. D., Xie, M. X., Yi, T. H., & Li, H. N. (2018a). Optimal Wireless Sensor Network Configuration for structural monitoring using automatic learning Firefly algorithm. Advances in Structural Engineering, 22(4), 907–918. https://doi.org/10.1177/1369433218797074
[54] Zhou, G. D., Xie, M. X., Yi, T. H., & Li, H. N. (2018b). Optimal Wireless Sensor Network Configuration for structural monitoring using automatic learning Firefly algorithm. Advances in Structural Engineering, 22(4), 907–918. https://doi.org/10.1177/1369433218797074



10.22214/IJRASET

45.98

IMPACT FACTOR: 7.129

IMPACT FACTOR: 7.429

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY