STM JOURNALS

Journal of
Advances in Shell Programming

ISSN: 2395-6690
Volume 12, Issue 1, 2025
January–April
DOI (Journal): 10.37591/JoASP

https://journals.stmjournals.com/joasp

Review JoASP

Exploring the Bourne Again Shell (Bash)

V. Basil Hans*

Abstract

The Bourne Again Shell (Bash) is a popular command-line interpreter and scripting language, created as a free software alternative to the original Bourne Shell (sh). Developed as part of the GNU Project, Bash extends the functionality of its predecessor by incorporating features from the C Shell (csh) and KornShell (ksh), making it more versatile and powerful. Bash offers robust command-line editing, job control, and support for functions and variables, enabling users to automate tasks, manage processes, and handle input/output redirection with ease. Its portability across Unix-like operating systems, including Linux and macOS, and its status as the default shell for many distributions, have cemented its position as a cornerstone of modern computing. Key features include advanced scripting capabilities, such as command chaining, loops, conditionals, and support for regular expressions, making Bash ideal for both interactive and non-interactive use. Programming with the Bourne shell resembles programming in traditional languages. If you have written code in languages like C, Pascal, BASIC, or FORTRAN, you will notice many familiar features. For instance, the shell has variables, conditional and looping constructs, functions, and more. Shell programming is also different from conventional programming languages. The shell itself offers limited functionality, so, most tasks rely on calling external programs. Consequently, the shell includes robust features for combining programs in sequence to accomplish tasks. It is an essential tool for system administrators, developers, and advanced users, offering a versatile platform to manage system resources and automate routine tasks. Bash's open-source nature ensures continued development and integration of new features, securing its relevance in evolving computing environments.

Keywords: Bash, UNIX, GNU operating system, file system shell-scripting

INTRODUCTION

Bourne Again Shell (Bash) is a powerful command-line interpreter and scripting language that has become the default shell in many Unix-like operating systems, particularly Linux. Originally developed as a free and open-source replacement for the Bourne Shell (sh) in 1989, Bash is part of the GNU Project and has since grown into one of the most widely used shells globally [1].

Bash is designed to provide users with an interactive environment for executing commands and automating tasks. It incorporates features from its predecessors, such as the Bourne Shell's simplicity, the C Shell's command history and job control, and the KornShell's advanced scripting capabilities [2]. This combination makes Bash both user-friendly for beginners and powerful enough for advanced users and developers.

***Author for Correspondence**
V. Basil Hans
E-mail: vhans2011@gmail.com

Research Professor, Department of Management & Commerce, Srinivas University, Mangaluru, Karnataka, India

Received Date: November 06, 2024
Accepted Date: November 09, 2024
Published Date: March 20, 2025

Citation: V. Basil Hans. Exploring the Bourne Again Shell (Bash). Journal of Advances in Shell Programming. 2025; 12(1): 1–9p.

The versatility of Bash extends beyond simple command execution. With features like input/output redirection, pipe chaining, and support for loops, conditionals, and functions, Bash enables users to write complex scripts to automate routine tasks, manage system resources, and perform system administration efficiently [3]. Its portability across many operating systems, including Linux, macOS,

and even Windows (through WSL), makes Bash a universal tool for system administrators, developers, and power users.

This introduction to Bash sets the stage for understanding its core functionalities, exploring how it has evolved to meet the needs of modern computing environments, and why it remains a key component of many systems today [4].

One of the most observed perceptions about using UNIX and Linux systems is the power that comes from flexibility in their commands and the link between them: the shell. A really powerful Unix and Linux shell has been the Bourne Again Shell, to give the right name. This shell consists of a command interpreter that provides the user terminal support for an effective human-interfaced environment as well as scripting and automation capabilities. Of course, not everyone loves to work on a command line [5]. We know GUIs, which are graphic user interfaces. While experienced Linux users understand the importance, beginners do not understand why we need to step up with the command lines and dig deeper into the operating system. Digging deeper introduces some of the key concepts of configuring your system and tells us more about why you need to learn the UNIX shell.

In this study, this powerful shell would be explored in a historical and technical context known as Bash. Bash is available on nearly every Linux and UNIX server [6]. Learn to configure how your shell behaves, as well as which file it uses in your home directory to store the settings. If you do not know it, Bash is also your login shell: go to the login or home directory by opening a terminal and typing. For an automation process, you are going to create a shell script that is a text file. In other words, Bash is available to everyone who uses UNIX or Linux and is an essential skill if you want to interact effectively with the operating system.

History and Evolution

The Bourne Again Shell, also known as Bash, is the default shell in most Linux distributions today. However, before the creation of Bash, there was the Bourne Shell, which was created at Bell Labs after the Thompson Shell [7]. The Bourne Shell has been developed into a number of historical branches, and Bash has, in turn, influenced the development of the Korn Shell and the Public Domain Korn Shell, from which MKS Korn Shell is derived. The original UNIX shell was nothing more than an interface to the operating system kernel that bundled standard keyboard input into system calls. As a result, developers created more advanced shells, such as the Thompson Shell, the Bourne Shell, the C Shell, and the Korn Shell. Shortly after that, one of the first UNIX releases was created at a software company in Berkeley, California.

The first version of Bash was released in January 1989. It was initially intended as a free replacement for the Bourne Shell. Bash owes much of its command syntax and usage to the Bourne Shell. The goal of the first release was to become the primary shell for the GNU operating system, and in 1993, Bash became the official shell of the Free Software Foundation, after which it quickly became the default shell of Unix-like systems. Since then, it has become the *de facto* standard for system administration and shell programming on Linux, Solaris, Mac OS, and Windows. In November 1979, development of the Korn Shell began at Bell Labs, and it would become the primary shell for Berkeley Systems Distribution. In 2017, a principal engineer in the operating systems group at Oracle conducted a security research aimed at the Bourne Again Shell. In the same year, when documenting a section of IEEE Std 1003.1-2017, it was reported that it was down to a “core” of 30 members, only indicating other than China and the United States [8].

Features and Capabilities

Bash allows users to prepare the shell for the interactive execution of commands. In addition to common command-line editing features, its capabilities include command-line history, job control, and an advanced set of wildcards and globbing. As a scripting language, Bash enables users to quickly and

effectively automate repetitive tasks so they do not have to type a bunch of commands over and over again. While Bash has many built-in commands for manipulating files and managing running processes, built-in commands may have limited capabilities or some missing options that are available with stand-alone utilities on some Unix systems. The dependence on external files means that a script can be transferred to another system with a compatible shell and have a better chance of functioning. Being the shell for Linux and also a primary shell for FreeBSD and macOS, Bash is the default shell on most of the systems [9].

The GNU Bash shell provides extensive support for command-line editing and programmable completion as it gives users the power to speed up their work when interacting with the file system, system commands, and even their shell environment. The basics covered in this study are critical for interacting with the shell. It is the very foundation that all of the practical applications of Bash will be built on in studies to come, so it is very important to get a solid understanding from the start.

BASIC COMMANDS AND SYNTAX

Whether new to the Bash terminal or most of the way through a tutorial, there are few basic commands you need to navigate the file system. Here, we list all of them and show you how to leverage them as you go. For questions of interaction with the file system, new syntax cannot be given without showing useful commands. Commands covered include ls, cd, pwd, and touch. Before we get into the syntax of Bash commands, and especially before we build from anywhere functionality, it is important to go over basic commands you need to navigate the file system on a daily basis in the terminal [10].

1. *ls (list directory contents):* This command outputs a list of names in the directory that is given to it (by default, the directory is your current directory).
2. *cd (change the shell working directory):* This command changes the directory you are in (by default, the new directory is a subdirectory of the current directory).
3. *pwd (print name of current working directory):* This command echoes the pathname of the current working directory, from the root back to the current directory. It is most useful as a mechanism for checking your location.
4. *touch:* The touch command is used to create files. For example, if I am in the directory home and I write touch thisisnothome or touch/thisisnothome, the file thisisnothome will be made, so it is found at, or very close to, the root directory.

Navigating the File System

Navigating the file system effectively is an essential skill to possess in any Unix environment. Before moving ahead, understanding the basic hierarchy of files and directories can be quite helpful. Whether you refer to it as "/", the root directory, the top-level directory, or something else, it all refers to the primary directory that houses every file and directory on the system. When you navigate through your graphical file system, your view of the file system is analogous to viewing it through binoculars. You do not see all possible directories and files; you only see the specific portion of the file system starting at your current location. The Bourne Again Shell is similar to the Command Prompt under Microsoft Windows and is a handy means of viewing the file system. You can use Bash to perform many of the same functions as the graphical file system viewer.

One of the basic ways to understand your place in the file system is through the "Print Working Directory" or pwd command. This command will display the entire path from the root directory to your current location. In the case of the User 1, this is the directory /Users/User 1/Desktop. The "/" indicates that you are in the root directory. Another useful command is the ls command, which displays the files and subdirectories within your current directory. The cd command allows you to move between different directories and especially to change your working directory to a different location. The change from one location in the file system to another location in the file system is considered a "navigation". When discussing the change in location, the term "path" is especially helpful. In your file system, all files and directories can be thought of as entries in the file system itself. To refer to a specific file or

directory, you can use either an absolute path or a relative path. Intermediate and advanced users like using the Bash command line because these commands can help automate long procedures and allow the user to access more subdirectories and files on the entire file system.

Working with Files and Directories

Exploring the Bourne Again Shell (Bash)

There are a few basic actions that you will need to perform more than others within the Bash environment. Among these are the abilities to move a file from one location to another, copy files, make directories, and remove files. Each of these abilities comes with its own set of predefined commands:

- cp: This command is used to create a copy of an existing file, that is to replicate the file in a different location, giving the new copy a different filename or different location.
- mv: This command is used to move a file from one location in your directory structure to another location, possibly onto another filesystem.
- rm: This command is used to permanently remove a file or directory from your system.
- mkdir: This command is used to make or create a new directory (or directories).
- rmdir: This command is used to remove or delete an existing directory from your system.

Because files, like everything else in Unix-based systems, are just files, managing them includes changing or restricting who can execute them. At its simplest, as long as you have read permission for a program file, you can execute it. If you do not have read permission, you cannot execute it. We can change this state of affairs; users and processes will not be able to interact with a file unless they have both permission and the appropriate level of access. To see the specific permissions for a file, take a look at the output of ls: ls-l filename. When you do so, there will be a string in the first line that contains several characters that show what the permission settings are. The letters will usually look something like -rwxr-xr-x. Each of these sections is a set of permissions that is associated with a position on a file. Thus, it can be divided as follows: activity-owner-group-world. Each position represents a bit of information: d is the directory indicator and - is the plain file indicator. The three activities simply represent read, write, and execute permissions. Writing a letter in the position is not of any use, but running chmod this way is certainly effective as chmod a-w filename or chmod a+r filename, for example. Rather than using the numeric method, a file can be set to executable with the command chmod u+x filename. The other permissions can be changed in the same way, and more than one permission can be set at once. For example, the command ugo+x filename would make the aforementioned file executable by the file's owner and all the members of the group it belongs to. With these commands at your disposal, you are well equipped to manage the files and directories in the contents of your files. To help reinforce your understanding, the section contains a series of examples that illustrate how to use each command. Additionally, some suggested use cases for each command are included, which might give you a better idea of when you might want to use these commands, either to practice on your own or as a starting point for a larger shell script. The next section will look at the different ways that you can interact with your Bash shell, and how you can adjust settings to make it more comfortable to work in. Remember, learning how to use your Bash shell can greatly enhance your ability to use your system, as well as contribute efficiency and speed to any type of scripting and automation you are doing.

SHELL SCRIPTING

As operating systems continue to evolve to provide cutting-edge tools and applications, the need to simplify repetitive tasks for the user prompted the creation of shell scripting. Shell scripting can assist you with written automation for your day-to-day tasks on not only a UNIX-like system but also on Linux-based systems. Shell scripting consists of various components, such as the shebang, assigning values to variables, and comments. Shebang is the first line of a shell script that determines which program to execute a script with and mentions that it is a Bash script. A new variable can be created by assigning a value to it either as a string or an integer. Comments must always start with a symbol and allow a user to present explanations or notes within the script.

After creating a new file, and once the shebang, variables, and subsequent comments have been entered, the completed script may then be saved. The file must then be made executable, which may be accomplished by running a command in a terminal. After marking the script as executable, the script may then be run. If one were to have several scripts in one directory, one may run a specific script by providing its full path. A shell script will conserve user resources and time by automating day-to-day operations. Day-to-day batch job systems can include tasks such as backing up data, generating reports, and additionally checking for unknown systems and users, all of which may be scripted. Proper structuring is imperative to allow for scripts to be interpreted by the system in a predictable manner. The improved readability of a structure may also allow for quick troubleshooting and serve as future documentation for users. To adhere to coding standards, one should strive to enforce proper syntax in following guidelines and finally make systematic and structured comments. To debug a shell script, a few methods may be employed. Integration simplifies any script that contains a syntax error or provides unexpected results by routing a script through the debug. If a single script contains errors, a standard method for manual debugging is to include a command in order to see the status or to view various variables.

Writing and Executing Scripts

The Bourne Again Shell (sh) is a script interpreter that allows users to type the commands that the kernel of their computer's UNIX-type operating system executes. Shells have a special mode of operation, called script. In this mode, the shell processes the commands given as input until either the end of file or an exit command is encountered. In script mode, the shell reads input from a file and transmits the output to another file. When a user types at the command prompt, the typed characters are executed. The command shell is an application that enables users to interact with their operating systems. Users can enter simple instructions and commands as sequences of text and let the shell execute these instructions and commands.

Today, most systems use the Bourne Again shell. By being able to write and execute scripts, you will then be able to explore automation in the Bourne Again shell and beyond. When a shebang line entered into the very first line of a text file, a script will be formatted as such, meaning it will be executed by the Bourne Again Shell. To create a simple script, open a terminal and use a text editor. Write the script, save, and close. According to best practices in Bash scripting, it is often best to begin any script with a shebang line. Beginning your script with this line will allow scripts to be run successfully when executed in the Bash environment. Some conventions with Bash scripts are relatively universal but may or may not be strictly followed. For example, Bash is not whitespace or case-sensitive, although it is often best to adhere to some of these guidelines, particularly when making a script public. Try running a script with and without the shebang to compare results. Start light with simple scripts to get the hang of simple coding patterns. Once you have written a script, you can try a few different ways to execute it. For a given directory, you will want to make sure you are in that directory in order to execute a file. If you are not, you can use the command to navigate there. Sometimes when you first start writing scripts, error messages will pour out of your terminal. Here are some common simple mistakes that may occur. Often, a helpful message will appear.

Control Structures and Loops

Having the ability to make decisions in your system's script is crucial. With that in mind, we want to take a little detour and explore some fundamental programming constructs with Bash to make your scripting experience more rewarding. Before you move on to abstraction, you will need to know about control flow constructs. These control structures come in a variety of types such as loops and conditionals. Some kinds of conditionals include if, case, and another obscure one called select. With these, you can set the script on an entirely different path based on some condition. When it comes to loops, you can perform actions until a certain condition is false with the until loop, keep a loop running while a condition is true with the while loop, or loop a set number of times with the for loop. This helps you write a script with less code, making that particular piece of automation more valuable. Additionally,

while some languages can be exhausting and awkward to add decision-making in, Bash has been designed to create these kinds of structures with ease. The ability to run the same piece of code multiple times and tailor a script to react differently based on certain conditions is essential to writing a dynamic and scalable automation effort. With only simple case constructs, these branching operations can themselves be nested, and loops can contain decisions. That makes sense especially when you are trying to solve a complex problem where the logic might otherwise overwhelm a simple script. With the exception of the select construct, the topic of nested loops and nested decisions with multi-level if constructs and case constructs extends the basic paradigm outlined.

CUSTOMIZING THE BASH ENVIRONMENT

There are a number of ways to customize the Bash environment. Here are a few concepts that will help us accomplish that goal. Aliases are used for customization purposes and convenience. By defining an alias, you effectively create a shortcut to a longer command.

Functionality that has been combined into a single script is difficult, if not impossible, to use with a different tool or in a different environment. By encasing a particular sequence of commands into a function, the commands are provided a restricted, encapsulated, and easily useful setting for reusability.

Bash continues to be a shell in use by many Unix-like operating systems. Each user can and may configure their login environment using dotfiles like .bashrc, .bash_profile, .profile, and .login. At login, .bash_profile is read. It then looks for .bashrc and sources it. This can be done immediately by starting a new terminal or by using source. An example shell environment file like .bash_profile follows.

The configuration of the Bash environment is user activity aimed at improving the overall experience of working in the shell. Using aliases and functions can prove useful in making work more efficient, thus saving time. The necessary customization, beyond what the Bash shell provides by default, will depend on how you work and interact with the system. User settings are determined by the contents of one or more personal login files such as ~/.bashrc, ~/.profile, ~/.bash_login, or ~/.bash_profile. This section introduces some of the available customization strategies. Look for the examples that follow to help you develop a feel for the functionality in today's Bash shell.

Aliases and Functions

Aliasing is a critical shell feature that helps improve efficiency in the Bash environment. Without aliases, to execute the command: ls -F -l, the user must type the long command string each time. With aliases, the time and typing are reduced. The command below is a user-defined alias: alias ll='ls -F -l'. For example, when a user types ll as a command and presses Enter, the shell replaces the ll command with the command ls -F -l before executing it. To display a full list of all aliases currently in effect, use the alias command by itself with no arguments. Unlike a built-in command synonym, with which an alias and its substitution may coexist, a user-defined alias and its positional equivalent cannot coexist. Used as part of a Bash script, the alias command creates an alias that is only available to the script and will not work in the user's or system's global environment after the script terminates.

In addition to aliases, advanced operational grouping in the form of functions can be carried out in the Bash environment. A function groups a set of commands into a single entity that can be called with a single word. At the programmatic level, functions are an excellent mechanism for organizing multi-step activities that need to be repeated at various times. Like variables and aliases, functions do not require a separate mechanism to declare them. To create a user-defined Bash function, the following simple procedure is used: FUNCTIONNAME() {COMMANDS;}. For this example, the command is: testfunction() {echo "This is a sample function";}. The commands do not run when entered since only the function has been declared. To run the testfunction function, type its name as a command and press Enter.

Configuration Files

File Breakdown Bash is a powerful tool for interacting with the operating system using textual commands. Since the terminal can be accessed by multiple users, this section presents information on how to customize your user experience within the Bash environment. When customizing Bash configurations, there are two files that users will typically work with: .bashrc and .bash_profile. The .bashrc file is automatically sourced each time a new session of Bash is started. The .bash_profile file is similarly sourced each time Bash is started; however, it is only sourced during the login command, which occurs after successful authentication of a user. Essentially, the .bash_profile file is used to set up the user's environment for all login sessions.

Configurations that usually go in the .bash_profile file perform actions like setting the prompt, modifying the PATH variable, creating environment variables, and configuring ulimit settings. The .bashrc file currently only sources the .bash_aliases file responsible for defining frequently used custom commands. Common customizations made to these files involve modifying the prompt or defining environment variables. Customizing the prompt allows a user to see details about their current directory, the system, the base of the home directory, and more! By setting an environment variable in these configuration files, users can maintain scrolls, fields, and limits of the output options. When one of these files is edited to make these customizations, a user can simply open up a new session in the terminal to test it out. This ensures safety when making these commands, as opening the session in the terminal allows a user to revert any undesired changes. It is recommended that you open a separate terminal window and log in to ensure that you have maintained access in the event that you accidentally lock yourself out!

Troubleshooting: If you applied any settings in the .bashrc file to modify or change the behaviour of your terminal and your terminals are not starting properly, follow these steps to troubleshoot:

1. Open a terminal using a different profile.
    a. Click Terminal at the top and select New Profile.
    b. Go to the Profile Preferences section, click on the Command tab, and select Run a custom command instead of my shell.
    c. Enter the full path to the bash shell executable.
2. Open the terminal as root and get rid of the issues from the file causing the problem: $ su $ vi /home/username/.bashrc
3. To lengthen your terminal's history: Append the following commands at the end of .bash_profile: export HISTTIMEFORMAT="%Y-%m-%d %H:%M:%S" shopt -s histappend HISTFILESIZE=100000 HISTSIZE=30720
4. To make customizations: Append all your customizations to a file with your name within your home directory and insert the line in your .bashrc to load this file.

ADVANCED BASH CONCEPTS

These concepts are provided as a means of showing the depth and versatility of the shell for skilled and knowledgeable users. New scripters may not see the point of learning such seemingly complex details. After all, you can already get quite a lot done with what you have learned. However, there does come a time, as your skill and experience increase, when the forms of command and pattern matching used up to now increasingly fail to impress or meet your scripting needs, especially when it comes to line-oriented text processing. Understanding and using basic and advanced shells and commands will provide additional benefits and functions that you use more and more often as you develop them.

Most programming languages include pattern matching. Regular expressions originated in the mathematical concept of set theory and have been used in various ways in computers and operating systems for decades. A product of the Unix operating system created in the 1960s, had gained some popularity by the early 1970s. A regular expression is a special text string used to express patterns. You can use regular expressions in searching, searching and replacing, and data validation. They are simply

a concise way of specifying multiple, alternative string names of text for any particular character, similar to how the command line itself uses three special "character" codes. There are two major uses of regular expressions: string matching and substring replacement.

The shell passes parameters to a function in exactly the same manner as any other command. As noted earlier, you can use the command line tools to manipulate a file path. Such commands are well suited to command line use, where you type each new command name, carefully call new commands with their output, and eventually receive the result you need. Unfortunately, they do not work so well in a scripting environment, ideally using a design based on modularity wherein scripts are written as simple operations that break a complex problem into simple subparts. Each part is constructed to perform a single function that is best described by its name.

Regular Expressions

A regular expression is a sequence of characters that describes a search pattern. Regular expressions are used by a variety of programming languages and are available in many tools for advanced text processing. Bash, the default shell on most UNIX-based systems, uses regular expressions to support advanced functionality for scripting. For this reason, a basic understanding of regular expressions is essential for anyone writing Bash scripts. This section introduces regular expressions, covers their basic syntax, and provides examples of their use. You can use regular expressions to describe a search pattern. As a result, regular expressions are used for a variety of text processing tasks, such as input validation or transforming text data. Regular expressions are supported by a variety of commands, such as grep and sed, as well as many programming languages.

Combine regular expressions with these commands and programming language constructs to effectively search for and transform text. Regular expressions describe patterns and therefore do not necessarily search for whole words. Regular expressions can be highly sophisticated. For users new to regular expressions, this study offers a concise summary of the most basic features. One of the limitations of this study is that it discusses regular expressions in the context of searching and text processing rather than in a general scripting context. Regular expressions can be similarly useful for tasks such as input validation when writing Bash scripts. It is recommended that users read the entire section before attempting the exercises. Mastering regular expressions to have an efficient workflow in data analysis and system administration. Use grep with regular expressions to search for patterns. Hierarchical choices are greedy and performance impairing, such as alternation; hence, carefully consider when using them.

Process Management

Process management is an essential aspect of using shell sessions, and Bash has specific constructs to manage running commands and applications. It is capable of managing a large number of system processes. The two main groups of processes are those running in the foreground and those running in the background. By running a command in the background, we are not prevented from continuing to perform other tasks in that shell. Job control offers a way to deal with commands or jobs running in the background, in the foreground of the given Bash session, or suspended, by using appropriate Bash commands. For computers running multiple tasks, it is important to be able to manage processes. Allowing some to run in the background, and others to receive input and display output helps to effectively utilize system resources.

Bash assigns every job a job ID in combination with the PID. The PID is not static over time, as every time the process is terminated, its PID is freed and can be reused by another. At the system process level, the command is assigned a process ID, or PID. Every process launched also has an identification number called zero, or in other words, it is considered the parent of all future processes. In many cases, part of these discussions will involve running multiple instances of a function or program simultaneously. A concept called process substitution is able to help with running multiple commands

at one time. It is identified by the use of open and close parentheses. When a command is running in the background, it does not mean that it is invisible; it just means that it will not get its input from or give its output to the display monitor. The jobs command is used to list all the jobs that are running in the background. Each job also has an identification string referred to as a 'job' ID.

CONCLUSION

Bourne Again Shell (Bash) has proven itself as an essential tool in the landscape of modern computing, offering a robust, flexible, and accessible command-line interface and scripting environment. As the primary shell in most Unix-like operating systems, including Linux, it is essential for system administrators, developers, and advanced users.

Bash's blend of simplicity, inherited from the original Bourne Shell, and advanced features borrowed from other shells, like C Shell and KornShell, makes it versatile for both interactive use and complex scripting tasks. Its portability across platforms and extensive support for automation, process management, and error handling ensures that Bash remains a reliable tool in both personal and professional environments.

Through continued development and its open-source nature, Bash maintains its relevance as computing needs to evolve, enabling users to automate workflows, manage systems efficiently, and enhance productivity. Ultimately, Bash is not only a shell for command execution but a powerful tool for solving real-world computing challenges, making it a vital component in the toolkit of anyone working with Unix-based systems.

REFERENCES

1. Kidwai A, Arya C, Singh P, Diwakar M, Singh S, Sharma K, Kumar N. A comparative study on shells in Linux: A review. Mater Today: Proc. 2021 Jan 1; 37: 2612–6.
2. Campesato O. Bash Command Line and Shell Scripts Pocket Primer. USA: Mercury Learning and Information; 2020 May 28.
3. Sampo Rapeli. Understanding the role of Unix shell in software development and developer experience. Thesis. Finland: Aalto University School of Science; 2024. Available from https://aaltodoc.aalto.fi/server/api/core/bitstreams/98c0f503-5e25-4257-8bd5-3f4a7072f31f/content
4. Bauraitė A, Brilingaitė A, Bukauskas L. Designing Trainee Performance Assessment System for Hands-On Exercises. 32nd International Conference on Information Systems Development (ISD2024 Gdansk, Poland). 2024. Available form https://aisel.aisnet.org/cgi/viewcontent.cgi?article=1577&context=isd2014
5. Kappelmann-Fenzl M. Introduction to Command Line (Linux/Unix). In Next Generation Sequencing and Data Analysis. Cham: Springer International Publishing; 2021 May 5; 71–78.
6. Singh SK. Linux Yourself: Concept and Programming. Chapman & Hall/CRC; Florida, USA; 2021.
7. Pfaff B. (1999 Aug 1). Bourne Shell Programming in One Hour. [Online]. Available from http://pjwstk.wafel.com/sop/shell.pdf
8. Spinellis D, Avgeriou P. Evolution of the Unix system architecture: an exploratory case study. IEEE Trans Softw Eng. 2019 May 2; 47(6): 1134–63.
9. Fox R. Linux with Operating System Concepts. Chapman & Hall/CRC; Florida, USA; 2021. DOI: 10.1201/9781003203322.
10. Olushile P. Essential Linux Commands, 100. Linux Commands Every System Administrator Should Know. Packt Publishing Ltd; Birmingham, United Kingdom; 2023 Nov 30.

