International Journal of Scientific Research in Engineering and Management (IJSREM)
Volume: 09 Issue: 11 | Nov - 2025 SJIF Rating: 8.586 ISSN: 2582-3930

# Cypher - An Operating System

**Prof. Ashwini R [1], Venkateshwaran S [2], Yashwanth Rao H [3], Sanjay G K [4], Manoj S [5]**

[1] *Ashwini R, Asst. Prof, Dept. of ISE, East West Institute of Technology*
[2] *Venkateshwaran S, Dept. of ISE, East West Institute of Technology*
[3] *Yashwanth Rao H, Dept. of ISE, East West Institute of Technology*
[4] *Sanjay G K, Dept. of ISE, East West Institute of Technology*
[5] *Manoj S, Dept. of ISE, East West Institute of Technology*

---

**Abstract -** Minimal Linux-based operating systems provide a practical environment for examining how user-space components interact with the kernel, but mainstream operating systems often obscure these fundamentals behind complex subsystems. Custom shells offer a clearer view of command interpretation by exposing the fundamental stages without inherited complexity. This paper presents Cypher, a Linux-based operating system designed with its own command interpreter called Rayshell, that demonstrates the basic stages of command processing, including lexical analysis, parsing, expansion and execution. The objective is not to reproduce the completeness or complexity of full-featured operating systems, but to demonstrate a clear, instructive design suitable for research, experimentation and educational use.

***Key Words*:** Operating System, Linux, Shell, Command Interpreter

## 1. INTRODUCTION

Linux provides a mature and well-established foundation for operating system development, offering stable kernel facilities for process management, memory handling, device interaction, and system services. Building anything on top of Linux usually means working with layers of tools and components that have been around for decades. Shells like Bash and Zsh are powerful, but they also carry a large amount of historical behavior and internal complexity. For someone trying to understand how a shell actually works at its core, these established tools can feel more like black boxes than learning material.

Cypher was developed as a Linux-based operating system intended to provide a simpler environment that behaves like a normal system but stays simple enough that its pieces can be followed without digging through years of legacy code. Instead of relying on an existing shell, the system includes a custom one named Rayshell. It is not designed to compete with established shells, but to offer a compact and understandable implementation that is easy to follow and extend.

Rayshell is built around the standard stages such as lexical analysis, parsing, expansion and execution. It implements essential features such as command execution, pipelines, redirections, variables, expansion rules, control-flow statements, job control, and basic scripting. The focus is on minimalism and correctness rather than replicating the full feature set of production-grade shells.

Cypher ultimately provides a minimal and functional system that is easy to inspect, modify, and experiment with. By integrating a lightweight graphical interface and a basic window manager, it offers a clear and usable environment.

## 2. LITERATURE REVIEW

Work related to lightweight Linux systems and custom shells mostly falls into three groups: instructional sources that show how to build a system from scratch, research on experimental Linux distributions, and official documentation that defines how Linux components are expected to behave. Together, these references shape the background for Cypher, but none of them directly focus on building a simple, easily-inspectable shell as the core of a small Linux environment.

Linux From Scratch (LFS) [1] is the closest thing to a foundation for projects like Cypher. It explains how to assemble a complete Linux user space using only source code, and it walks through system initialization, dependency handling, and toolchain setup. LFS is practical and detailed, but its goal is to teach system construction, not to analyze design choices behind small shells or minimal environments. Still, it sets the baseline for understanding how a user space can be built cleanly and with full control.

Research projects such as ASH OS [2] and other Linux-based prototypes [3] show how developers build customized distributions for performance, UI changes, or specific use cases. These papers are useful for understanding how a distribution can be shaped around a particular idea, but they lean toward feature integration

rather than simplification. Their focus is on improving the user experience or adding system-wide capabilities, not on exposing the internal workings of the shell.

The Linux Foundation's documentation [4] provides the technical rules that all Linux systems and shells are expected to follow like system-call behavior, process management, POSIX features, and general compatibility guidelines. It gives the authoritative definitions needed to ensure that a custom shell behaves correctly, but it does not explore minimal designs or educational shells.

## 3. SYSTEM DESIGN

Cypher is structured in three layers: a minimal Arch Linux base, the custom Rayshell interpreter, and a lightweight graphical setup built with Hyprland. The system design as shown in Fig 3.1, keeps each layer understandable, and easy to work with, avoiding the heavy complexity of a full Linux distribution.

### 3.1 Base System

The system begins with a very minimal Arch Linux environment. Arch's simplicity and packaging tools make it practical to assemble only what is required: the kernel, core userland programs, filesystem utilities, init setup, and the toolchain needed to build Rayshell.

By avoiding additional services and background components, the base remains predictable and easy to inspect.

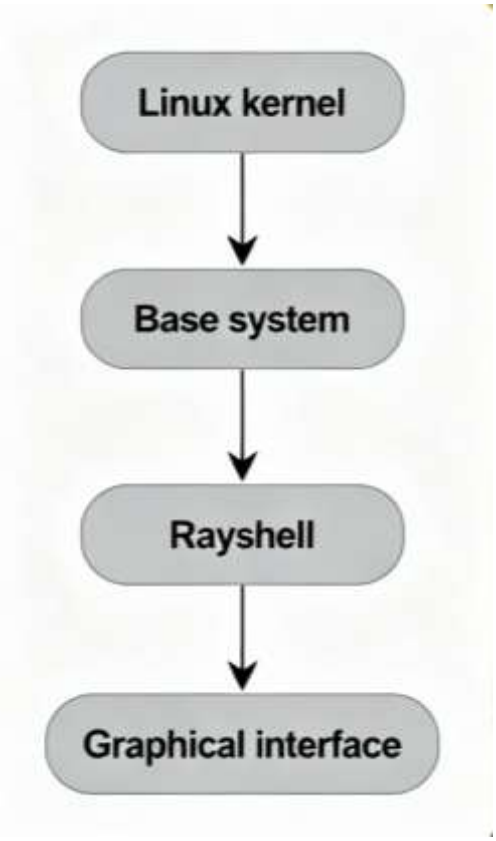


**Fig -3.1**: System Architecture

### 3.2 Rayshell

Rayshell is the main component that defines how a user interacts with Cypher, so its design is built around strict separation of stages as shown in Fig 3.2. The interpreter is structured as a pipeline, where each stage handles one well-defined responsibility before handing control to the next.

1. *Tokenization*: The lexer scans the raw input string and breaks it into tokens: words, operators, redirection symbols, variable references, assignment statements, and control keywords. The lexer handles quoting rules, escape sequences, and token boundaries. Its output is a linear token stream with exact type information.

2. *Parsing*: The parser walks through the token stream and constructs an internal representation of the command, an abstract syntax tree. It encodes pipelines, redirection operations, control structures such as if and while, grouped commands, and shell lists separated by semicolons or newlines. Rayshell uses a grammar specifically written for the project instead of copying the long and inconsistent grammars used by traditional shells. This keeps the parser predictable and reduces hidden behaviors.

3. *Expansion*: After the parse step, Rayshell performs expansions on the parsed structure. This includes variable expansion, command substitution, and pathname expansion (globbing). The expansion stage is intentionally isolated from lexing and parsing to make debugging easier.

4. *Built-ins*: Rayshell has its own built-ins for core operations like *cd, pwd,* variable assignment, and job listing. Built-ins bypass process creation overhead and interact directly with shell state (working directory, environment table, job list). Their behavior is kept minimal and predictable to maintain the instructional purpose of the shell.

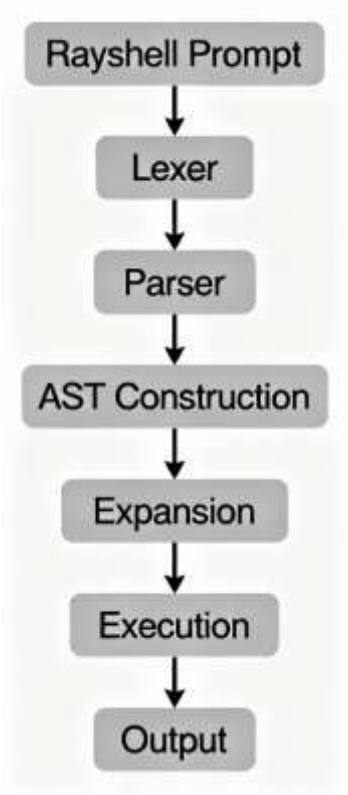


**Fig -3.2** Shell Architecture

5. *Execution*: The executor walks the expanded structure and performs the actual actions like running external commands, evaluating built-ins directly inside the shell process, setting up pipes and file descriptors, handling redirections, managing background jobs and also implements basic job control so that users can suspend, resume, and inspect running jobs.

### 3.3 Graphical design

Cypher adopts a minimal graphical environment built around Hyprland, a lightweight Wayland compositor known for its responsiveness and modular configuration model. The goal is to provide a usable environment that supports interaction with Rayshell and basic system tools without adding unnecessary complexity that comes with traditional minimal systems.

The setup relies on only a few components: Hyprland as the compositor and window manager, a taskbar for basic system information, and Kitty as the primary terminal emulator. Most user interaction still happens inside Rayshell, so the GUI exists mainly to make the system easier to use rather than to replace the shell. Hyprland's configuration files make it straightforward to define keybindings, organize windows, and fine-tune behavior without writing custom window-manager code.

This design keeps the graphical layer minimal and easy to understand. Instead of trying to imitate a full desktop environment, Cypher uses a small, predictable setup that supports experimentation and keeps the user close to the system's underlying behavior. Images of the terminal and desktop are shown in Fig 4.1 and 4.2

## 4. IMPLEMENTATION

**Fig -4.2**: Desktop of Cypher

The development of Cypher was carried out in a pragmatic manner: first building a minimal Linux base, then developing Rayshell, finally putting together a simple graphical environment for usability.

### 4.1 System setup

The development process started with a manual Arch Linux installation. Only the necessary components like bootloader, toolchain, and a minimal window manager were included. Keeping the system small made it easier to see exactly what was running and to avoid unwanted complexities while developing the shell.

Most of the work here was straightforward: partitioning the disk, preparing the filesystem, installing the kernel, setting up networking, and configuring a user account. The goal was to create a clean slate where Rayshell could be added without conflicts or legacy behavior from existing shells.

Once the system reached the point where it would boot reliably and provide a login, the foundation was

**Fig -4.1:** Terminal with Rayshell

considered ready for Rayshell.

### 4.2 Development of Rayshell

Rayshell was developed using Python, to make the development process easier.

The initial concept was to use Python's subprocess module, because that is the usual path when implementing a shell-like interface. The problem is that subprocess behaves like a middleman. Every command becomes a "request," and the module hands it off to the system like an outsourced job. That works for normal utilities, but not for a shell intended to feel native and immediate.

So Rayshell took a more direct approach: Python's *os* module, specifically os.exec*() was used to spawn and replace processes at a much lower level. When Rayshell launched a command, the shell process *became* that command. When the command terminated, control passed back to the parent stub, which then re-initialized Rayshell's prompt. That behavior gave the shell a cleaner, more authentic execution model, closer to how classic shells like bash or dash operate.

Writing the command dispatcher was the next major task. It needed to distinguish between built-ins and external binaries. The fragile part was ensuring RayShell never accidentally exec()'d itself out of existence by replacing its own process while still needing to run. The built-ins like change directory, exit, environment variable manipulation, and a few helper utilities were implemented as plain Python functions. Everything else was delegated to the real system binaries through the execvp family. This design made RayShell light, fast, and predictable.

Error handling had to be handled manually. Invalid commands, missing binaries, broken environment paths, and failed permission checks all surfaced as raw OSError exceptions. These weren't wrapped by Python into friendly messages; RayShell had to intercept them and respond with consistent shell-like output. The goal was to avoid Python's tracebacks entirely, preserving the illusion that RayShell was a native part of the operating system rather than an application sitting on top of it.

The shell's prompt system was intentionally minimal. No fancy glyphs or animations, just 'rayshell'. The intention was to keep the experience grounded and stable, especially since the system was intentionally stripped-down during installation. RayShell ultimately became a small, explicit, transparent shell. No layers of abstraction, no unnecessary complexity, and no dependency bloat. Its behavior is deterministic because everything it does is visible: when it runs a command, it directly hands execution to the OS; when something breaks, the cause is obvious.

This simplicity is what made it fit neatly into the rest of Cypher's system architecture. The shell didn't need to be clever; it only needed to be solid enough to stand as the user's primary interface and flexible enough to coexist with the GUI layer without conflicting with the base system.

### 4.3 Features of Rayshell

Rayshell supports the essential Unix shell workflow: executing binaries, managing working directories, inspecting the environment, and running basic system utilities. Commands such as *cd, exit,* and environment variable manipulation are handled internally so execution does not break the shell's own process context. Everything else is dispatched directly to system binaries through *execvp*, allowing RayShell to behave like a conventional POSIX shell without re-implementing core tools.

Control structures (if and while) were developed with an easier syntax rather than copying bash. The syntax, as shown in Fig. 4.3 and 4.4, is closer to a programming language's syntax, so anyone who has a basic


**Fig -4.3**: Working of If block

understanding of programming languages can use Rayshell without having to deal with complex semantics.

History support was added for usability, mostly out of frustration. Without it, even trivial mistakes required retyping long commands. RayShell loads a history file on


**Fig -4.4**: Working of History command

startup, appends every executed line, and exposes it interactively.

Job control was the most technically demanding feature but also the most important for interactivity. A shell is supposed to let programs run in the background, bring them back, and keep track of what's alive. RayShell manages this using direct calls to fork(), exec*(), and waitpid() through Python's os module. The shell creates process groups, monitors them, and lets users suspend and resume tasks.

The final but crucial feature was signal handling, because without it the shell behaves like a fragile demonstration rather than a real interactive environment. As soon as RayShell started running external programs, the limitations became obvious: pressing Ctrl-C would either kill the entire shell or fail to interrupt the running command, depending on how Python chose to react. Neither outcome is acceptable in anything calling itself a shell. To fix this, Rayshell has explicit handlers for signals such as SIGINT and SIGTSTP and routes them to the correct process group instead of letting Python's default behavior take over. This required tying together RayShell's job-control system with low-level UNIX primitives so the foreground job responds instantly to user interruptions while the shell itself remains unaffected. In practice, this means users can halt a runaway command, stop a long-running job, or resume it later without destabilizing the shell.


**Fig -4.5**: Working of While Block

### 4.4 Graphical Interface

The graphical interface of Cypher was designed to be simple, functional, and lightweight, emphasizing usability without unnecessary complexity. Window management is handled by Hyprland, a modern dynamic tiling compositor for Linux, selected for its speed, minimal resource usage, and straightforward integration with custom components.

A custom taskbar was developed using the PyQt5 framework in Python. This taskbar allows easy access to open workspaces, and other basic features. A complete desktop of Cypher is shown in Fig. 4.2.

Initially, a custom terminal emulator was attempted to unify the interface experience, but achieving stable performance was challenging. Hence, Kitty, a fast and feature-rich terminal emulator, was incorporated as the default terminal. Kitty integrates seamlessly with

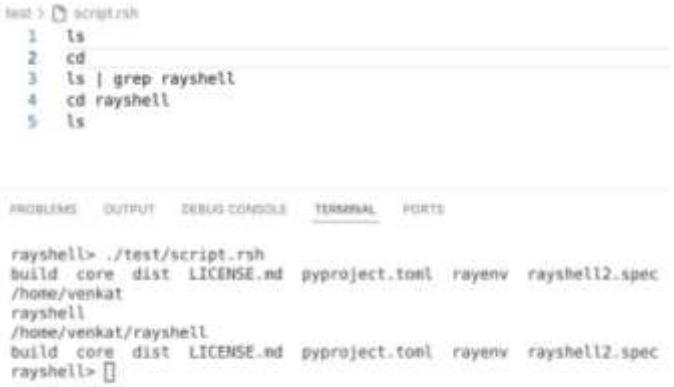


**Fig -4.6**: Working of Script file

Hyprland and the taskbar, offering reliable command-line access while maintaining the visual and functional goals of the interface.

The graphical environment was designed to work hand-in-hand with Rayshell, offering a smooth and intuitive user experience. The goal was to keep it stable, minimal, and flexible, reflecting Cypher's lightweight philosophy, while leaving space for future improvements and additions.

## 5. CONCLUSION

Cypher demonstrates how a Linux-based operating system can be customized around a lightweight, educational shell and a minimal graphical interface. Rayshell showcases a clear and structured approach to command interpretation, with features such as custom control structures, job management, history tracking, and signal handling, all built without relying on external subprocess frameworks. The graphical environment complements this by providing a stable, responsive interface through Hyprland, a PyQt5-based taskbar, and a reliable terminal emulator in Kitty.

Within the wider ecosystem of Linux-based operating systems, Cypher occupies a very different role compared to projects such as EndeavourOS or Omarchy. Both of them prioritize approachability and a complete desktop environment. Cypher, in contrast, integrates its own command interpreter which focuses on experimentation alongside usability. Future work could expand both the shell and interface capabilities, exploring custom terminal features, enhanced GUI elements, or deeper integration with system services, while maintaining the lightweight and modular philosophy that Cypher embodies.

## ACKNOWLEDGMENT

We deeply express our sincere gratitude to our guide, Mrs. Ashwini R, Asst. prof, Department of ISE, EWIT for her guidance throughout the development of this work.

## REFERENCES

[1] G. Beekmans, Linux From Scratch, LFS Community.

[2] ASH OS: A Comprehensive Linux Based Operating System with Optimized User Interface, by Arvind Kumar, M. Iyyappan, S. Priyan, Abhijat, Sourabh Kumar Jha, and Himanshu Gaikward, International Journal of Advanced Computer Science and Applications, vol. 14, no. 2, 2023.

[3] Linux-Based OS Prototypes, by Kumar et al., Journal of Emerging Computing Systems, 2023.

[4] The Linux Community Documentation, by The Linux Foundation.

[5] OSDev Wiki: Operating System Development Community.