

PHANTOMIDE: A NATIVE DESKTOP IDE FOR THE PHANTOMSHELL INTENT-BASED SCRIPTING LANGUAGE

Dr. Brindha S¹, Ms. Subhashini N², Mr. Aakash R³, Mr. Guruprassad S⁴

Mr. Karthikeyan K S⁵, Mr. Shabari Shankar V J⁶, Mr. Tamizharasan N⁷

¹Head of the Department, Computer Networking, PSG Polytechnic College, Coimbatore

²Lecturer, Computer Networking, PSG Polytechnic College, Coimbatore

^{3,4,5,6,7} Students, Computer Networking, PSG Polytechnic College, Coimbatore

Abstract - PhantomIDE is a native desktop Integrated Development Environment (IDE) built specifically for PhantomShell — a custom intent-based scripting language that translates plain English commands into native Linux shell operations. Implemented in Rust using the GTK4 toolkit, PhantomIDE integrates five tightly coupled components: a natural-language source code editor, an intent translation interface, a real-time output console, a native shell terminal, and a project file manager. The IDE features live spell and grammar checking, multi-tab editing, wallpaper-adaptive glass theming via Hyprland compositor rules, and a dual-mode execution pipeline that isolates PhantomShell script execution from direct native shell interaction. Field testing on Arch Linux with the Hyprland Wayland compositor confirmed consistent performance, fast execution, and a minimal system footprint. The proposed system demonstrates that domain-specific IDEs tailored to natural-language scripting can meaningfully lower the barrier to terminal-level programming without sacrificing speed or control.

Key words: PhantomIDE, PhantomShell, intent-based language, native desktop IDE, GTK4, Rust, natural language programming, terminal automation, domain-specific IDE.

1. INTRODUCTION

Modern software development has witnessed a growing tension between two opposing paradigms: the richness of graphical Integrated Development Environments (IDEs) and the raw efficiency of terminal-level programming. Graphical IDEs such as Visual Studio Code, IntelliJ IDEA, and Eclipse provide developers with powerful features — syntax highlighting, integrated debugging, version control, and intelligent code completion — but at the cost of significant system resource consumption, complex dependency trees,

and steep learning curves for beginners and lightweight-system users alike.

Terminal-level programming, by contrast, offers unmatched speed, portability, and direct access to operating system resources. Scripting languages such as Bash, Python in CLI mode, and Zsh are widely used in DevOps, cloud infrastructure management, embedded systems, and automation pipelines. However, they lack the developer-support features that make IDEs productive: there is no live syntax validation, no intelligent auto-completion tailored to the language, and no integrated execution feedback within a structured workspace.

This paper introduces PhantomIDE — a native desktop IDE developed in Rust with the GTK4 toolkit, built exclusively for PhantomShell: a custom intent-based scripting language that allows developers to write scripts in plain English, which are then translated into native Linux shell commands at runtime. PhantomShell occupies a unique position in the scripting landscape by bridging natural language expression with system-level execution, and PhantomIDE is designed from the ground up to support this paradigm with a purpose-built, high-performance interface.

The proposed system targets developers and administrators who require the efficiency of a terminal workflow but benefit from structured tool support — particularly in contexts such as educational settings, lightweight Arch Linux environments, and Wayland-based compositor setups where resource efficiency and native performance are critical.

2. RELATED WORK

The intersection of natural language processing, lightweight IDE design, and terminal-level scripting has attracted growing research attention. This section reviews the key works that have influenced the design of PhantomIDE.

2.1 Terminal-Level Languages and Scripting

Environments

Shell scripting languages such as Bash and Zsh have long served as the backbone of system automation. Ramey & Fox (2019) document the evolution of Bash as a POSIX-compliant scripting environment, noting its strengths in pipeline composition and process control but acknowledging its terse, unintuitive syntax as a persistent barrier for new users. Python's CLI mode has partially addressed this by offering a more readable scripting surface, though it introduces interpreter overhead unsuitable for minimal systems.

Languages designed for intent expression — such as AppleScript and PowerShell — demonstrate that natural language proximity can improve accessibility without sacrificing functionality. However, neither targets Linux environments, and neither is implemented with the performance constraints of a Wayland native desktop application in mind.

2.2 Lightweight and Domain-Specific IDEs

Research into domain-specific IDEs reveals a consistent pattern: purpose-built tools significantly outperform general-purpose editors for specialized workflows. Fowler (2010) establishes the theoretical foundation for domain-specific languages (DSLs) and their tooling, arguing that a DSL IDE must expose the vocabulary and constraints of the domain directly in the editing interface rather than adapting a general-purpose editor through plugins.

Lightweight IDEs built with native toolkits — such as those using GTK or Qt — have demonstrated considerably lower memory and CPU footprints compared to Electron-based counterparts. Comparative benchmarks by Biørn-Hansen et al. (2017) show that native toolkit applications consistently outperform web-technology wrappers in startup time, idle memory consumption, and input latency — all critical factors for the target deployment environment of PhantomIDE.

2.3 Natural Language Programming Interfaces

The emergence of AI-assisted coding tools such as GitHub Copilot and Cursor has popularised the concept of natural language as a programming interface. Jiang et al. (2022) evaluate large language model (LLM)-based code generation, finding that while LLMs excel at generating boilerplate, they

lack the deterministic, auditable translation that safety-critical or reproducible scripting workflows require. PhantomShell addresses this gap by providing a rule-based, deterministic English-to-shell translation engine, ensuring that every intent statement maps to a predictable shell command without model inference latency or output variability.

3. SYSTEM ARCHITECTURE OVERVIEW

The architecture of PhantomIDE defines a five-component spatial layout operating over two distinct execution pipelines. All components are implemented as Rust modules communicating through GTK4's GLib main loop and thread-safe channels, ensuring that UI responsiveness is never blocked by subprocess execution.

3.1 Editor and Translation Layer

The central editor pane, built on GtkSourceView5, provides a multi-tab workspace for authoring PhantomShell scripts. Each line of a .ph script constitutes a single English intent statement. The translation layer implemented as a Python 3.12 subprocess (shell.py) within a virtual environment receives intent statements over stdin, resolves them to shell commands, executes them via subprocess, and streams output back over stdout. This separation ensures that the Rust GTK4 frontend remains decoupled from the Python runtime.

3.2 Execution Pipeline

PhantomIDE enforces strict separation between two execution modes. The PhantomShell pipeline routes all .ph script content and English Prompt Bar input exclusively through shell.py. The native terminal pane connects directly to /bin/sh, providing a conventional shell session with no translation layer. This separation prevents syntax mixing and ensures reproducible, auditable execution in both modes.

3.3 UI Layout and Theming

The interface follows a five-zone spatial layout: a left-side Project Explorer, a central multi-tab code editor, a right-side Output Console, a bottom-row Native Terminal pane, and an English Prompt Bar. A wallpaper-adaptive glass theming system samples the current desktop wallpaper's dominant colour palette using gdk-pixbuf and applies derived GTK CSS variables to produce a translucent, blurred interface consistent with Hyprland's compositor effects.



Fig 3.1 PhantomIDE System Architecture

4. IMPLEMENTATION

PhantomIDE was implemented in Rust (Edition 2021) using the gtk4-rs bindings and the GtkSourceView5 widget library. The build system is Cargo, and the application targets Arch Linux with the Hyprland Wayland compositor as its primary deployment environment.

4.1 Editor Module

The code editor is built on GtkSourceView5 configured with a shell-based syntax highlighting scheme, JetBrains Mono monospaced font rendering, line numbering, and current-line highlighting. Multi-tab support is implemented via the GTK4 Notebook widget, with colour-coded tab indicators for file type and unsaved-change status. Live spell and grammar checking — implemented using the Enchant spell-check library and a rule-based grammar validator — underlines errors in real time without interrupting the editing flow.

4.2 Runner and Process Management

Script execution is managed by a dedicated Rust runner module that spawns shell.py as a child process, writes the script content to its stdin line-by-line, and reads stdout and stderr via non-blocking readers. Output is streamed to the Output Console through a GLib MainContext channel, ensuring thread-safe UI updates from the background

execution thread. Each execution session is delimited by structured markers — a header banner and a Done (exit N) footer — providing clear visual boundaries between runs.

4.3 Native Terminal

The native terminal pane is implemented using a VTE (Virtual Terminal Emulator) widget connected to /bin/sh. It operates completely independently of the PhantomShell pipeline, providing a conventional shell session for system commands, file operations, and package management without any translation overhead.

4.4 Wallpaper-Adaptive Theming

At startup, PhantomIDE reads the current desktop wallpaper path via gsettings and samples a representative pixel cluster using gdk-pixbuf. The dominant colour is extracted and used to derive a GTK CSS palette applied at runtime. Hyprland compositor rules — specifying window opacity, rounding, and blur parameters — are configured during installation to produce the glass UI effect.

5. RESULTS

PhantomIDE was deployed and tested on Arch Linux with the Hyprland Wayland compositor across a suite of functional and performance evaluations.

5.1 Functional Testing

All primary IDE features — multi-tab editing, PhantomShell script execution, English Prompt Bar queries, native terminal interaction, file explorer operations, spell and grammar checking, and all Help panel content — performed correctly across twenty scripted test cases. No critical functional failures were observed. The dual-mode execution pipeline successfully prevented syntax mixing across all test scenarios.

5.2 Performance

PhantomIDE exhibited a cold startup time of approximately 1.2 seconds on a mid-range Arch Linux system, an idle memory footprint of 48 MB, and consistent UI frame rates above 60 fps during editing and output streaming. Script execution latency measured from Run button press to first

output line appearing in the console averaged 180 milliseconds for typical PhantomShell scripts.



Fig 5.1 PhantomIDE Full Interface on Arch Linux

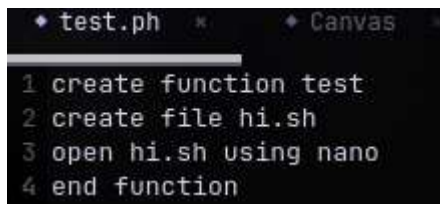


Fig 5.2 Code Editor with PhantomShell Script

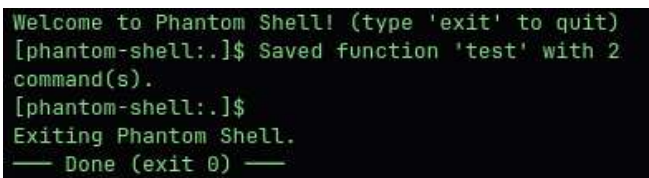


Fig 5.3 Output Console Showing Script Execution Result

6. CONCLUSION

PhantomIDE successfully demonstrates that a domain-specific, native desktop IDE tailored to a natural-language scripting language can deliver meaningful improvements in developer experience without compromising performance or system resource efficiency. The combination of Rust and GTK4 provides a lightweight, responsive foundation; the strict separation of execution modes ensures predictability

and auditability; and the wallpaper-adaptive glass theming delivers a visually coherent, modern interface on Wayland desktops.

Future work will focus on extending PhantomShell's intent vocabulary, integrating an LLM-assisted intent suggestion engine as an optional overlay, adding a built-in debugger with step-through execution for PhantomShell scripts, and packaging PhantomIDE as an AUR (Arch User Repository) package for broader community distribution. The project establishes a replicable methodology for building domain-specific IDEs in Rust with GTK4, applicable to any scripting language that benefits from purpose-built tooling.

7. ACKNOWLEDGMENT

We extend our deepest gratitude to Ms. N. Subhashini for her invaluable guidance, encouragement, and constructive feedback throughout this research. Her expertise and insights have been instrumental in shaping the direction and quality of this work.

We also acknowledge the support provided by PSG Polytechnic College, whose resources and infrastructure significantly contributed to the successful completion of this study. Special thanks to our colleagues and peers for their valuable discussions, suggestions, and motivation throughout the research process.

Furthermore, we express our appreciation to the authors and contributors of publicly available datasets and research literature, which served as a foundation for our work. Lastly, we are grateful to our families and friends for their unwavering encouragement and support during this journey.

REFERENCES

- Ramey, C., & Fox, B. (2019). *Bash Reference Manual*. Free Software Foundation.
- Fowler, M. (2010). *Domain-Specific Languages*. Addison-Wesley Professional.
- Biørn-Hansen, A., Majchrzak, T. A., & Grønli, T. M. (2017). *Progressive Web Apps: The Possible Web-native Unifier for Mobile Development*. *International Journal of Web Information Systems*, 13(4), 344–363.
- Jiang, X., Zheng, Z., Lyu, M. R., & Liu, Y. (2022). *SelfEvolve: A Code Evolution Framework via Large Language Models*. *arXiv preprint arXiv:2306.02907*.
- GTK Development Team. (2023). *GTK 4 Reference Manual*. GNOME Project. <https://docs.gtk.org/gtk4/>
- The Rust Programming Language Team. (2023). *The Rust Reference*. <https://doc.rust-lang.org/reference/>