Newbie Shell Design Document v2.2

1. Design Philosophy

Core Mission

Newbie is a modern, user-friendly Linux shell interpreter designed to complement traditional shells with natural language commands and predictable behavior. Built in Rust with a threaded architecture, newbie runs alongside existing shells (bash, zsh) rather than replacing them, allowing users to gradually adopt natural language syntax while preserving existing workflows and infrastructure compatibility.

Trading CPU Cycles for Cognitive Load Reduction

Newbie intentionally trades raw execution performance for dramatic improvements in user experience. The design recognizes that modern computing bottlenecks are cognitive rather than computational - users spend far more time debugging syntax errors, looking up command flags, and wrestling with escaping rules than waiting for commands to execute.

Traditional Approach:

- Minimize CPU usage above all else
- Cryptic syntax to reduce keystrokes
- Minimal error messages to save processing
- User debugging time considered "free"

Newbie Approach:

- Minimize user mental overhead
- Readable syntax even if more verbose
- Comprehensive error messages with suggestions
- Accept interpreter overhead for usability gains

Justification: If a newbie command takes 50ms instead of 10ms but eliminates 5 minutes of documentation lookup and debugging, that's a 600x net performance improvement from the user's perspective.

Natural Language Over Cryptic Abbreviations

- Commands use readable English words: &find, &show, &copy, &remove
- Syntax follows natural language patterns: &find error &in logs.txt

- No arbitrary abbreviations requiring memorization
- Consistent verb-object-modifier pattern across all commands

Predictable Behavior

- Commands do exactly what they say, nothing more or less
- No "smart" behavior that changes based on context
- Consistent output formatting across all commands
- Same input always produces same output

No Escaping Required

The String Delimiter Problem: String delimiters are the primary source of shell scripting pain, causing quote escaping nightmares, variable quoting bugs, nested delimiter hell, and multiple escaping layers.

The Solution: Newbie eliminates escaping entirely through complete separation of command and data contexts:

- Command context: User input lines and .ns script files where &keywords have special meaning
- Data context: File content being processed where all text is literal
- No mixing of contexts eliminates collision scenarios entirely

Users can process any content without escaping concerns: JSON files with complex quoting, source code with various syntaxes, log files with special characters, database dumps with embedded quotes.

Deployment Model

Newbie operates as a separate interpreter:

- `newbie` - Interactive shell session
- `newbie script.ns` - Execute newbie script files
- Traditional bash scripts (.sh) continue working unchanged

This ensures zero disruption to existing infrastructure while enabling gradual adoption based on user preference and task appropriateness.

2. Design

Universal & Prefix System

Critical Decision: All commands use the & prefix without exception. This eliminates parsing ambiguity and provides:

- Eliminates command/data context collisions
- Simplifies parser state machine
- Enables delimiter-free processing throughout
- Consistent mental model for users

```
newbie

&exit                    # No longer just 'exit'
&show file.txt               # All commands use &prefix
&admin &copy files/ &to backup/
&call external_script.sh
```

Command Building Architecture

Problem Solved: Timing issues with natural language syntax where &move was executing before &to could set the destination.

Solution: Two-phase processing:

1. **Command Structure** - Command struct accumulates all command components
2. **Handler Functions** - Build commands instead of immediate execution
3. **Parse/Build Phase** - Collect all modifiers and arguments
4. **Execution Phase** - execute_command() runs fully constructed commands

Command Handler Pattern:

- Context Modifiers (&first, &last, &numbered) update Command and return Continue
- Action Commands (&show, &copy, &move) set command.action and return Stop
- Target Modifiers (&to) set destination and return Continue

Memory Management Strategy

Critical Constraint: Command structure must never store content data to avoid Vec memory explosion on large datasets. Command stores only configuration state (flags, limits, modes, source/destination paths) that guide streaming operations.

Pattern:

- Modifiers configure the command by setting command fields
- Action commands execute streaming operations using command as processing guidance
- Data flows through without storage in intermediate structures

- Fixed-size buffers (MAX_ARGS_PER_KEYWORD = 8) prevent memory expansion

Line-at-a-time Processing:

- Natural boundaries for pattern matching state machine
- Each line becomes discrete unit for complete pattern evaluation
- Streaming with fixed-size buffers prevents memory expansion
- Circular buffer for &last N &lines operations - keep only N lines in memory

Parsing Architecture

Whole-Line Parsing Strategy: The parser analyzes entire command lines as complete units to avoid backtracking issues during parsing, then uses line-based streaming for execution.

Parsing Process:

1. Complete line analysis - identify all components before execution
2. Context resolution - command context determines parsing rules
3. Variable resolution timing - determine when variables resolve
4. Execution preparation - prepare parsed structure for streaming

Static Keyword Registry: Function pointer approach with O(1) command lookup and compile-time verification of command handlers.

Threading Compensation Strategy

Multi-threaded architecture with adaptive scaling:

- **Reader thread:** Stream input data line by line
- **Worker threads:** Process operations incrementally using left-to-right matching (adaptive count)
- **Writer thread:** Format and output results as they become available

Threading Strategy: Auto-detect CPU cores and allocate max(1, cores - 2) worker threads, reserving cores for reader/writer threads. Traditional shells process sequentially, but newbie can have multiple threads working on pipeline parts simultaneously, often compensating for interpreter overhead through parallelization.

Pattern Language: Readable Alternative to Regular Expressions

Problems with Regular Expressions:

- Cryptic, unreadable syntax

- Poor debugging capabilities
- Single-threaded performance limitations
- Backtracking complexity prevents efficient streaming
- Excessive escaping requirements

Newbie's Solution: Left-to-right streaming pattern language designed for incremental matching:

Basic Elements:

- &start and &end - Beginning and end anchors
- &text - Any characters
- &letters - Any letters, case-insensitive
- &upperletters, &lowerletters - Case-specific letters
- &numbers - Numeric digits

Quantified Matching:

- &text5 - Exactly 5 text characters
- &letters3 - Exactly 3 letters
- &numbers4 - Exactly 4 numbers

Streaming Advantages:

- No backtracking required - patterns evaluated left-to-right
- Real-time processing without buffering entire contents
- Multi-threaded performance with reader/worker/writer threads
- Memory efficient - only small working buffers needed

Command Architecture

Unified Parsing: Verb + Object + Modifiers pattern maintains natural English flow:

```
[VERB] [OBJECT] [MODIFIERS]
&show file.txt &numbered
&find error &in logs.txt
&copy file.txt &to backup/
&remove directory/ &subdirs
```

Context-Aware Action Determination:

- Object type detection via trailing slash convention
- Positional parsing enables context-aware modifier validation
- Unified modifier system with consistent & prefix rules
- Parser knows valid modifiers for each object type

Traditional Command Unification

Find Command: Replaces ls, grep, and find with context-aware behavior:

```
newbie

&find *.txt                  # File listing
&find error &in logs.txt         # Content search (literal mode)
&find &start error &numbers &end   # Pattern mode
```

Show Command: Universal display with composable modifiers:

```
newbie

&show file.txt               # Paged display (less)
&show file.txt &raw              # Raw output (cat)
&show file.txt &first 20 &lines    # First 20 lines (head)
&show file.txt &numbered         # With line numbers
```

Copy Command Architecture: Natural language front-end to rsync:

- &preserve → -a (archive mode)
- &verify → --checksum (verify transfers)
- &sync → --delete (mirror mode)
- &compress → -z (compress during transfer)
- &resume → --partial (resume interrupted transfers)

```
newbie

&copy source/ &to destination/ &preserve
# → rsync -a source/ destination/


&copy files/ &to backup/ &sync &verify
# → rsync -a --delete --checksum files/ backup/
```

Variable System

Transparent Resolution Model: Variables resolve when the interpreter has sufficient data, eliminating complex timing categories:

- Assignment-time resolution: Variables resolve immediately when assigned explicit values
- Query-time resolution: System and process variables resolve when referenced
- No namespace-based timing: All namespaces use same resolution logic

Namespace Organization:

- &v. - User-defined variables
- &system. - System state and environment
- &process. - Process information
- &network. - Network state
- &global. - Cross-session configuration
- &config. - Application configuration

Admin Command Architecture

Bounded privilege escalation with automatic cleanup:

```
newbie

&admin &copy sensitive.conf &to /etc/
&admin &show /var/log/secure &last 50 &lines
```

Security Model:

- Each &admin command isolated - no persistent elevated privileges
- Uses sudo for privilege escalation
- Automatically calls sudo -k after command completion
- Always positioned leftmost for clear privilege scope

External Script Integration

Bidirectional interoperability for gradual adoption:

From Newbie to External Scripts:

```
newbie
```

```
&call backup_script.sh
&call python analyze_logs.py &with /var/log/
```

From Bash to Newbie:

```
bash

newbie daily_maintenance.ns
newbie script.ns | grep error
```

Configuration Philosophy

Replace bash's cryptic configuration with human-readable alternatives:

```
newbie

prompt:
  user: green bold
  host: green bold
  path: blue bold
  symbol: default

shortcuts:
  ll = &find all &with details &with types
  la = &find all hidden

&if &system.path not contains ~/.local/bin: &+ ~/bin: &then
  &system.path = ~/.local/bin: &+ ~/bin: &+ &system.path
&end
```

Control Flow

Indentation-based structure: Uses whitespace indentation like Python, eliminating brackets and semicolons:

```
newbie
```

```
&if backup_needed &then
   Check available disk space before starting backup
   &show &system.disk.free

   Start the backup process
   &copy important_files/ &to backup/ &preserve &verify
&end

&for &file &in &*.txt
   &if &file matches &start &upperletters &numbers .txt &end &then
      &move &file &to processed_ &+ &file
   &end
&end
```

Comment System

Lines without & as the first non-whitespace character are automatically comments:

```
newbie

This is a comment
&show file.txt &numbered
   This indented comment describes the above command
&find error &in logs.txt
```

Error Handling Philosophy

Leverage Rust's Result types for comprehensive error messages:

- Context-aware error descriptions
- Specific suggestions for common mistakes
- Clear indication of where parsing failed
- Educational guidance rather than cryptic codes

```
Error: Pattern incomplete - missing &end after &start
Command: &find &start error &numbers &in logs.txt
Try: &find &start error &numbers &end &in logs.txt
```

3. Implementation

Technology Stack

- **Language:** Rust for memory safety and performance
- **Threading:** Reader/worker/writer pattern across all components
- **Tool Integration:** FFI bindings to GNU utilities; rsync front-end for &copy
- **Parsing:** Static keyword registry with function pointers for O(1) lookup

Core Implementation Structure

```rust
```

```rust
// Command structure that handlers build up
#[derive(Debug, Clone)]
pub struct Command {
    pub action: Option<String>,     // move, copy, show, etc.
    pub source: Option<String>,
    pub destination: Option<String>,
    pub first_n: Option<usize>,
    pub last_n: Option<usize>,
    pub current_unit: LineOrChar,
    pub numbered: bool,
    pub original_numbers: bool,
    pub raw_mode: bool,
}

// Function pointer type for command handlers
type CommandHandler = fn(&[&str], &mut Command) -> Result<ExecutionResult, Box<dyn Error>>;

// Static keyword registry - stays in RAM
static KEYWORDS: &[KeywordEntry] = &[
    KeywordEntry { name: "&exit", handler: handle_exit },
    KeywordEntry { name: "&show", handler: handle_show },
    KeywordEntry { name: "&find", handler: handle_find },
    KeywordEntry { name: "&move", handler: handle_move },
    KeywordEntry { name: "&to", handler: handle_to },
    KeywordEntry { name: "&admin", handler: handle_admin },
    KeywordEntry { name: "&call", handler: handle_call },
    // ... additional commands
];

// Fixed-size buffers prevent memory expansion
const MAX_ARGS_PER_KEYWORD: usize = 8;
```

Core Parsing Implementation

Critical Memory Constraint: Never use Vec when processing data iteratively - it expands memory usage unpredictably. Use streaming approaches with fixed-size buffers, iterators, or circular buffers.

```rust
```

```rust
pub fn parse_and_execute_line(input: &str) -> Result<(), Box<dyn Error>> {
    // EXCEPTION: This Vec is OK - only splitting command line, not processing file data
    let tokens: Vec<&str> = input.split_whitespace().collect();

    let mut command = Command::new();

    // Fixed-size storage prevents memory expansion on large datasets
    let mut current_keyword: Option<&str> = None;
    let mut current_args: [Option<&str>; MAX_ARGS_PER_KEYWORD] = [None; MAX_ARGS_PER_KEYWORD];
    let mut arg_count = 0;

    // Parse tokens building command structure
    for token in tokens {
        if token.starts_with('&') {
            // Process previous keyword, start new one
        } else {
            // Add argument to current keyword
        }
    }

    // Execute the complete command
    execute_command(&command)
}
```

Command Handler Implementation

rust

```rust
fn handle_move(args: &[&str], command: &mut Command) -> Result<ExecutionResult, Box<dyn Error>> {
    if args.len() != 1 {
        return Err(NewbieError::new("&move requires exactly one source argument"));
    }
    command.action = Some("move".to_string());
    command.source = Some(args[0].to_string());
    Ok(ExecutionResult::Stop)
}

fn handle_to(args: &[&str], command: &mut Command) -> Result<ExecutionResult, Box<dyn Error>> {
    if args.len() != 1 {
        return Err(NewbieError::new("&to requires exactly one destination argument"));
    }
    command.destination = Some(args[0].to_string());
    Ok(ExecutionResult::Continue)
}
```

Command Execution Implementation

rust

```rust
fn execute_command(command: &Command) -> Result<(), Box<dyn Error>> {
    match command.action.as_deref() {
        Some("move") => {
            let source_path = command.source.as_ref()
                .ok_or_else(|| NewbieError::new("&move requires source"))?;
            let dest_path = command.destination.as_ref()
                .ok_or_else(|| NewbieError::new("&move requires &to destination"))?;

            // Expand tilde paths, validate existence, handle directory vs file logic
            let expanded_source = expand_tilde(source_path);
            let expanded_dest = expand_tilde(dest_path);

            // Directory vs file logic based on trailing slash convention
            if expanded_dest.ends_with('/') {
                // Handle as directory destination
            } else {
                // Direct rename/move
                fs::rename(&expanded_source, &expanded_dest)?;
            }
        },
        // Other command implementations...
    }
    Ok(())
}
```

Main Interactive Loop

```rust
```

```
fn main() -> Result<(), Box<dyn Error>> {
    println!("Newbie Shell v0.2.0 - Command Building Architecture");
    println!("Type '&exit' to quit");
    println!("Try: &move file.txt &to newfile.txt");

    let stdin = io::stdin();
    loop {
        print!("newbie> ");
        io::Write::flush(&mut io::stdout())?;

        let mut line = String::new();
        match stdin.read_line(&mut line) {
            Ok(0) => break, // EOF
            Ok(_) => {
                let trimmed = line.trim();
                if trimmed.is_empty() {
                    continue;
                }
                if let Err(e) = parse_and_execute_line(trimmed) {
                    println!("Error: {}", e);
                }
            }
            Err(e) => {
                println!("Error reading input: {}", e);
                break;
            }
        }
    }
    Ok(())
}
```

Implementation Phases

Phase 1 (COMPLETED): Core parsing engine with command building architecture

- ✓ Static keyword registry with function pointers
- ✓ Fixed-size buffers for memory efficiency
- ✓ Universal & prefixing for all commands
- ✓ Two-phase processing (parse/build then execute)
- ✓ &move command with &to modifier

Phase 2: Pattern language implementation

- &start...&end structure with streaming constraints
- Basic character classes (&letters, &numbers, &text)
- Quantified matching (&numbers4, &letters3)

Phase 3: Additional commands

- &copy command using rsync front-end pattern
- &remove, &admin, &call with natural language syntax
- &show command with composable modifiers

Phase 4: Control flow structures

- &if, &for, &while with indentation-based parsing
- Pattern matching in conditionals

Phase 5: Variable system

- &v., &system., &global. namespaces
- Transparent resolution model
- Arithmetic operations with & prefix

Phase 6: Integration and polish

- GNU tool integration via FFI
- Pipeline operations (into syntax)
- Configuration system
- Comprehensive error handling

Architecture Advantages

- **Function pointer dispatch** eliminates runtime string matching overhead
- **Fixed-size argument buffers** prevent memory bloat
- **Static registry** enables compile-time verification of command handlers
- **Clear separation** between parsing and command building phases
- **Two-phase processing** resolves natural language timing issues
- **Streaming execution** maintains constant memory usage
- **Threading compensation** often overcomes interpreter overhead

Performance Strategy

Two-phase execution model:

1. **Parse phase:** Natural language commands parsed once into efficient Rust operations
2. **Runtime phase:** Compiled operations execute at native speed with streaming

Left-to-right streaming: Pattern matching occurs incrementally as data arrives, without backtracking or look-ahead, enabling real-time processing without buffering entire files.

Current Implementation Status

The prototype demonstrates core design principles in working code:

- Command building architecture prevents timing issues
- Universal & prefix system eliminates parsing ambiguity
- Fixed-size buffers support streaming architecture
- Static keyword registry provides O(1) lookup
- Natural language command structure replaces cryptic flags

This serves as a working proof-of-concept validating core design principles while providing foundation for the complete Newbie shell system.

This document reflects the architectural decisions and implementation status as of Phase 1 core engine completion with command building architecture. The spec-driven approach ensures design consistency as complexity increases during development.