Newbie Shell Design Document v2.3

CRITICAL ARCHITECTURAL DECISION: UNIFIED EXECUTION MODEL

Major Update: All command execution in Newbie flows through the `&run` primitive. This creates a clean separation between natural language parsing (what the user wants) and process execution (how it gets done).

Memory 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.

Execution Architecture:

- Modifiers configure the command by setting command fields
- Action commands build executable command strings
- All execution happens via `&run` with the built command string
- Data flows through without storage in intermediate structures

UNIFIED EXECUTION MODEL: Everything is &run

The fundamental architectural insight is that all commands, whether native Newbie operations or external scripts, ultimately become `&run` operations with different command strings.

Command Flow Architecture:

```
Natural Language Input → Command Building → Executable String → &run Execution
```

Examples of the unified model:

```
&copy source/ &to dest/ &preserve
→ builds: "rsync -a source/ dest/"
→ executes: &run rsync -a source/ dest/

&admin &copy files/ &to /system/backup/
→ builds: "sudo rsync -a files/ /system/backup/"
→ executes: &run sudo rsync -a files/ /system/backup/

script.py
→ executes: &run python script.py

&run custom_tool --flag value
→ executes: &run custom_tool --flag value
```

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.

Modern Hardware Reality:

- Your consumer Ryzen 7 has more computational power than entire university computer centers from the Unix era
- Multi-gigabyte RAM is standard
- Multi-core processors can parallelize line processing
- SSDs make streaming I/O fast enough that buffering entire files is often unnecessary
- The bottleneck is human comprehension, not CPU cycles

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

Examples:

```
&find error &in logs.txt  # Command: &in is command modifier
# When processing logs.txt content:
# "Database error &in connection pool" - &in is just literal text

&find user said "I can't connect to the server" &in support_logs.txt
&find SQL: INSERT INTO table VALUES ('O'Brien', "quote") &in database_logs.txt
&find C:\Program Files\App\config.ini not found &in error_logs.txt
```

The universal & prefix system works without conflicts because commands and data exist in completely separate processing contexts.

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. Universal & Prefix System (NO EXCEPTIONS)

CRITICAL CHANGE: All commands now use the & prefix without exception. This eliminates parsing ambiguity discovered during implementation.

Examples:

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

Implementation Benefits:

- Eliminates command/data context collisions

- Simplifies parser state machine
- Enables delimiter-free processing throughout
- Consistent mental model for users

3. Command Building Architecture

Problem Solved: Timing issues with natural language syntax (e.g., `&move file.txt &to newfile.txt`) where `&move` was executing before `&to` could set the destination.

New Architecture:

1. **Command Structure** - Added Command struct to accumulate all command components
2. **Handler Functions** - Changed from immediate execution to command building
3. **Two-Phase Processing** - Parse/build phase, then execute phase
4. **Execution Engine** - execute_command() runs fully constructed commands, always via `&run`

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
- **Execution Commands** (`&run`) execute the built command string

Working Example:

```
&move file.txt &to newfile.txt

Processing Flow:
1. handle_move() sets command.action = "move" and command.source = "file.txt"
2. handle_to() sets command.destination = "newfile.txt"
3. execute_command() translates to appropriate system command and executes via &run
```

4. The &copy Command: rsync Front-End Implementation

The `&copy` command serves as a natural language front-end to rsync rather than implementing file copying functionality directly. This approach leverages rsync's decades of optimization and robust handling of edge cases while providing Newbie's discoverable syntax.

Why rsync as backend:

- Handles complex scenarios: partial transfers, network interruptions, permission preservation

- Optimized delta transfers and compression
- Battle-tested across different filesystems and conditions
- Extensive option set covers virtually all file operation scenarios
- Superior error handling and recovery mechanisms

Natural Language Mapping to rsync flags:

Newbie Modifier	rsync Flag	Purpose
&preserve	-a	Archive mode: permissions, timestamps, ownership
&verify	--checksum	Verify transfers via checksums
&sync	--delete	Mirror mode: remove extra files in destination
&compress	-z	Compress during transfer
&resume	--partial	Resume interrupted transfers
&progress	--progress	Show transfer progress
&bandwidth 100KB	--bwlimit=100	Limit transfer rate
&exclude *.tmp	--exclude='*.tmp'	Exclude patterns
&dry_run	--dry-run	Preview operations
&verbose	-v	Detailed output

Command Examples:

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

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

&copy large_dataset/ &to remote_server/ &compress &progress &bandwidth 1MB
# → &run rsync -az --progress --bwlimit=1000 large_dataset/ remote_server/

&copy project/ &to backup/ &exclude *.log &exclude *.tmp &dry_run
# → &run rsync -a --exclude='*.log' --exclude='*.tmp' --dry-run project/ backup/
```

5. Pattern Language: Left-to-Right Streaming (No Regex)

Design Philosophy: Modern hardware can handle natural language parsing, eliminating the need for users to learn regex syntax and debug backtracking issues.

The Regex Problem: Traditional regex engines suffer from fundamental issues inappropriate for modern shell usage:

- Cryptic, unreadable syntax: `^[A-Z]+[0-9]+\.txt$` is meaningless to most users
- Poor debugging capabilities with unhelpful error messages
- Backtracking complexity that prevents efficient streaming
- Single-threaded performance limitations
- Excessive escaping requirements

Solution: Left-to-Right Streaming Pattern Language

Newbie implements a pattern language designed for efficient left-to-right processing that works naturally with line-based streaming:

Natural Language Pattern Syntax:

Basic Elements:

- `&start` and `&end` - Beginning and end anchors (equivalent to regex ^ and $)
- `&text` - Any characters (equivalent to regex .*)
- `&letters` - Any letters, case-insensitive (equivalent to [A-Za-z]*)
- `&upperletters` - Uppercase letters only (equivalent to [A-Z]*)
- `&lowerletters` - Lowercase letters only (equivalent to [a-z]*)
- `&numbers` - Numeric digits (equivalent to [0-9]*)

Quantified Matching: All character classes support optional numeric quantifiers:

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

Examples:

```
&find &start error &numbers &end &in logs.txt
# Processes character by character as file is read
# Immediately identifies matches without storing entire file in memory
# Scales to arbitrarily large files

# Complex literal text (impossible to escape cleanly in regex):
&find &start he said, "copy the file to C:\hosts". I don't know why &end &in file.txt
```

Mode Detection: The parser automatically detects search mode:

- **Literal mode:** No & pattern keywords present - simple substring search
- **Pattern mode:** & pattern keywords present - full pattern language active

Implementation: Simple state machine with no backtracking:

```rust
enum MatchState {
    LookingForStart,
    FoundStart,
    MatchingNumbers,
    Complete
}
```

Streaming Architecture Advantage: The `&start...&end` format enables incremental matching as data streams in:

- No backtracking required: Patterns are evaluated left-to-right as characters arrive
- Real-time processing: Large files can be processed without buffering entire contents
- Multi-threaded performance: Reader/worker/writer threads process data continuously
- Memory efficient: Only small working buffers needed, not entire file contents

6. File I/O and Compression Architecture

Transparent Compression Support

Design Philosophy: All file operations should work transparently with compressed files without requiring users to specify compression formats or use different commands.

Magic Byte Detection: Automatic detection of compression formats through file headers:

- **gzip:** `[0x1f, 0x8b, ..]`

- **bzip2:** `[0x42, 0x5a, ..]`
- **xz:** `[0xfd, 0x37, 0x7a, 0x58, 0x5a, 0x00]`
- **zstd:** `[0x28, 0xb5, 0x2f, 0xfd, ..]`

Unified Reader Architecture

create_reader() Function: Central abstraction that returns appropriate decompression readers based on file format detection:

```rust
fn create_reader(path: &str) -> Result<Box<dyn BufRead>, Box<dyn Error>>
```

Integration Strategy:

- All file reading operations use create_reader() instead of direct File::open()
- BufReader-based line processing works identically for compressed and uncompressed files
- Streaming architecture maintains constant memory usage regardless of compression
- No changes required to existing command logic - compression is completely transparent

Supported Compression Formats

Rust Crate Integration:

- **flate2** - gzip/deflate support (.gz, .deflate)
- **bzip2** - bzip2 support (.bz2)
- **xz2** - LZMA/XZ support (.xz)
- **zstd** - Zstandard support (.zst)
- **lz4_flex** - LZ4 support (.lz4)

7. Command Architecture

Show Command: Universal Display

Universal display with composable modifiers:

```
&show file.txt                  # Paged display (less equivalent)
&show file.txt &raw             # Raw output (cat equivalent)
&show file.txt &first 20 &lines     # First 20 lines (head equivalent)
&show file.txt &last 20 &lines      # Last 20 lines (tail equivalent)
&show file.txt &numbered            # With line numbers (renumbered 1-N)
&show file.txt &original_numbers    # With original file line numbers
&show file.txt &first 1000 &chars   # First 1000 characters
&show file.txt &last 1000 &chars    # Last 1000 characters
```

Implementation Strategy:

- Don't use external programs for basic display (cat, head, tail) - adds overhead without benefit
- Exception: Interactive paging - implement native paging similar to less
- **Advantage:** Compression support - less doesn't handle compressed files, our implementation does

Memory Constraints:

- Use BufReader for line-by-line processing (already established)
- For `&last N &lines` - use circular buffer to avoid loading entire file
- For `&first N &lines` - can terminate early after N lines
- Compression decoders integrate transparently with BufReader

Find Command: Context-Aware Search

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

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

8. 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

9. Admin Command Architecture

Bounded privilege escalation with automatic cleanup:

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

Implementation Strategy:

- Uses sudo for privilege escalation
- Automatically calls `sudo -k` after command completion to clear cached credentials
- Always positioned leftmost in command structure for clear privilege scope
- Integrates with existing sudo configuration and audit systems

Security Model:

- Each `&admin` command is isolated - no persistent elevated privileges
- Bounded scope prevents privilege leakage to subsequent commands
- Leverages battle-tested sudo mechanisms rather than custom privilege handling
- Full audit trail through sudo logging infrastructure

10. External Script Integration

Bidirectional interoperability for gradual adoption:

From Newbie to External Scripts:

```
&run backup_script.sh
&run python analyze_logs.py /var/log/
&run make target=release
```

From Bash to Newbie:

```
bash

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

Design Principles:

- Clean handoff between shell environments with preserved exit codes
- Arguments passed through seamlessly
- Environment variables inherited from calling shell
- Standard input/output/error streams connected properly
- Script isolation - each `&run` is independent

11. Configuration Philosophy

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

```
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
```

12. Control Flow

Indentation-based structure

Uses whitespace indentation like Python, eliminating brackets and semicolons:

```
&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:

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

13. 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

Error: &admin must be leftmost modifier
Try: &admin &copy file.txt &to /etc/ instead of &copy &admin file.txt &to /etc/
```

14. 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
- **Compression:** Transparent decompression through create_reader() abstraction

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,
    pub run_command: Option<String>, // The final executable command
    pub admin_mode: bool,           // Whether to wrap with sudo
}

// 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: "&copy", handler: handle_copy },
    KeywordEntry { name: "&move", handler: handle_move },
    KeywordEntry { name: "&to", handler: handle_to },
    KeywordEntry { name: "&admin", handler: handle_admin },
    KeywordEntry { name: "&run", handler: handle_run },
];
```

Memory Management Strategy (CRITICAL CONSTRAINT)

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

CRITICAL: Never use Vec when processing data iteratively - it expands memory usage unpredictably. Use streaming approaches with fixed-size buffers, iterators, or circular buffers for line-based processing.

Fixed-Size Buffer Architecture:

```rust
// Core parsing uses fixed buffers
let mut current_args: [Option<&str>; MAX_ARGS_PER_KEYWORD] = [None; MAX_ARGS_PER_KEYWORD];

// Line processing maintains constant memory usage
let lines: Vec<&str> = output.lines().collect(); // OK - finite line count
let selected_lines: Vec<&str> = if let Some(n) = first_lines {
    lines.into_iter().take(n).collect() // OK - bounded by n
} else { lines };
```

Threading Architecture

Adaptive Threading Strategy: Auto-detect CPU cores and allocate max(1, cores - 2) worker threads, reserving cores for reader/writer threads.

Multi-threaded Pipeline Processing:

- **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 Compensation Strategy: Traditional shells process commands sequentially, but newbie can have multiple threads working on different parts of a pipeline simultaneously. For operations like pattern matching across large datasets, this parallelization often compensates for interpreter overhead, especially on multi-core systems.

Implementation Phases

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

- ✓ Static keyword registry with function pointers implemented in Rust prototype
- ✓ Fixed-size buffers (MAX_ARGS_PER_KEYWORD = 8) for memory efficiency
- ✓ Streaming approach with line-based processing boundaries
- ✓ Universal & prefixing enforced for all commands (including &exit)
- ✓ Command building architecture with two-phase processing
- ✓ &move command with &to modifier implemented

Phase 1a - Core Execution Engine:

- &run command implementation (generic process execution)
- Process spawning, argument handling, stream management
- Error handling and exit code propagation

Phase 1b - Admin and Security:

- &admin command implementation (sudo wrapper around &run)
- Privilege escalation with automatic cleanup
- Security model and bounded scope

Phase 1c - Natural Language Front-Ends:

- &copy command (rsync front-end using command building pattern)
- &remove command
- &show command with modifiers (&numbered, &first, &last, &lines, &chars)

Phase 2: Pattern language implementation

- &start...&end structure with line-by-line streaming constraints
- Start with basic character classes (&letters, &numbers, &text)
- Add quantified matching (&numbers4, &letters3) later

Phase 3: Threading Architecture

- Implement adaptive threading: max(1, cores - 2) worker threads
- Reader/worker/writer pattern with line-based work units
- Circular buffer for &last N &lines operations

Phase 4: Variable system

- &v., &system., &global., &process., &network., &config. namespaces
- Transparent resolution: assignment-time for explicit values, query-time for system variables

Phase 5: Integration & Polish

- GNU tool integration via &run
- Pipeline operations (into syntax)
- Configuration system
- Error handling with helpful suggestions

15. Performance Strategy

Two-phase execution model:

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

Left-to-right streaming architecture during execution phase: The natural language format inherently supports streaming processing. Patterns like `&find &start error &numbers &end &in logs.txt` can be matched incrementally as data arrives, without requiring backtracking or look-ahead.

This streaming approach maximizes throughput on modern multi-core systems while maintaining low memory usage, and provides compatibility with existing tool reliability.

16. 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
- **Transparent compression** works with all file operations without user intervention
- **Unified execution model** creates clean, testable, maintainable architecture where everything flows through `&run`

17. Success Metrics

User Experience:

- Reduced learning curve for shell newcomers
- Faster task completion for common operations
- Fewer syntax errors and debugging sessions
- Improved script maintainability

Performance:

- Faster text processing through multi-threading

- Competitive performance with traditional tools through parse-once, execute-fast model
- Efficient resource usage

Adoption:

- Standalone pattern matching tool adoption
- Integration into Linux distributions
- Community contribution and extension

18. Conclusion

Newbie represents a fundamental rethinking of shell design, prioritizing human comprehension and modern computing capabilities over historical constraints. By combining natural language interfaces with threaded performance and reliable tool integration via the unified `&run` execution model, newbie makes shell computing accessible to broader audiences while maintaining the power needed for advanced use cases.

The design eliminates major pain points of traditional shells - cryptic syntax, hostile error messages, poor performance, and maintenance difficulties - while preserving essential functionality through the clean architecture where everything flows through `&run`.

The elimination of regex and string delimiters alone represents a paradigm shift that saves developers countless hours of syntax wrestling, allowing them to focus on solving actual problems rather than battling quote escaping mechanics and backtracking complexity.

Most importantly, the unified execution model creates a clean, testable, and maintainable architecture where natural language parsing, command building, and process execution are cleanly separated, with `&run` serving as the universal execution primitive that everything builds upon.

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