Newbie Shell Design Document v49

PERFORMANCE BENCHMARK TARGET

Test File: `~/Archive/latest-truthy.nt.bz2` (41.7GB compressed Wikidata N-Triples)

- **Purpose**: Performance comparison between Newbie Shell, pure Rust implementation, and Unix pipeline commands
- **Format**: N-Triples RDF data with predictable line structure (well under 4KB line limit)
- **Availability**: Publicly downloadable from Wikidata for reproducible benchmarks
- **Scale**: Real production data size for validating streaming architecture and threading compensation theory

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:

```bash
&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
```

NEW: BASH Command Execution with Silent Default

&run BASH Syntax: Resolves quote parsing complexity by treating everything after BASH as a shell command string.

Silent Execution by Default: Commands execute without output unless explicitly prefixed with `&show`.

&show as Both Command and Modifier: Natural dual functionality for file display and output enabling.

Examples:

```bash
&run BASH echo 'Hello World'      # Silent execution
&show &run BASH echo 'Hello World' # Display output
&run BASH ls -la | grep error     # Complex pipeline, silent
&show &run BASH find /var -name "*.log" # Display results
```

NEW: Magic Byte Detection for Transparent Decompression

Eliminates User Cognitive Load: All file operations work transparently with compressed files without requiring users to specify compression formats.

Implementation: Automatic detection through file headers enables threaded performance benefits while maintaining streaming architecture.

Supported Formats:

- **gzip:** [0x1f, 0x8b, ..]
- **bzip2:** [0x42, 0x5a, ..]
- **xz:** [0xfd, 0x37, 0x7a, 0x58, 0x5a, 0x00]
- **zstd:** [0x28, 0xb5, 0x2f, 0xfd, ..]

NEW: .ns Script Abstraction

Complex Pipelines as Simple Recipes: .ns scripts abstract complex data transformation pipelines into readable, natural language commands while maintaining competitive performance through the underlying streaming architecture.

Example Use Case: The Wikidata processing pipeline demonstrates the target - complex data transformation expressed in natural language while maintaining performance through streaming.

NEW: &lookup Command - In-Memory Reference Data

Hybrid Architecture Approach: While maintaining streaming constraints for primary data processing, the `&lookup` command loads reference files into memory for fast key-value lookups during data transformation operations.

Memory Allocation: Uses up to 40% of available system RAM for lookup tables, providing substantial capacity (e.g., 12.8GB on a 32GB system) while preserving memory for OS operations and streaming buffers.

Use Case: Common ETL pattern where large datasets are enriched with reference data lookups without requiring database joins or disk-based operations.

Core Functionality

- **Single Lookup File:** Only one lookup file can be loaded at a time to maintain memory predictability
- **Key-Value Format:** Simple text format with one key-value pair per line: `key=value`
- **Variable Integration:** Lookup values accessible through `&lookup.keyname` namespace
- **Memory Boundary:** Fails fast if lookup file exceeds 40% RAM limit rather than attempting partial loads

Syntax and Usage

```bash
```

```
&lookup reference.txt              # Load lookup file into memory
&find &lookup.product_id &in data.csv   # Use lookup values in processing
&show &lookup.category             # Display specific lookup value
```

Error Conditions

- **File too large:** Clear error message with file size and RAM limit
- **Invalid format:** Line-by-line format validation with error reporting
- **Memory allocation failure:** Graceful handling with cleanup
- **File not found:** Standard file existence checking

Implementation Constraints

- **HashMap Storage:** Internal use of HashMap for O(1) lookup performance
- **Single Instance:** Only one `&lookup` namespace active at a time
- **Memory Tracking:** Runtime calculation of 40% RAM limit based on system detection
- **Format Validation:** Strict key=value parsing with error reporting for malformed lines

ENHANCED: Threading Architecture Strategy

Current Implementation: Single-threaded with fixed-size buffers for memory efficiency.

Proposed Extension: Simple three-thread architecture per newbie instance:

- **Reader thread:** Handle file I/O and decompression for the input file
- **Worker thread:** Pattern matching and filtering on the decompressed stream
- **Writer thread:** Output formatting, compression, and writing results

Scaling Strategy: Run multiple newbie instances for different processing patterns rather than coordinating multiple outputs within a single instance. Each instance processes independently with its own three-thread pipeline.

Benefits:

- Simple implementation: no coordination between patterns or shared state
- Natural OS-level scheduling across available cores
- Fault isolation: one pattern failure doesn't affect others
- Flexible scaling: add more instances as processing power allows

Example multi-pattern processing:

```bash
newbie extract_labels.ns input.bz2 &   # Instance 1: 3 threads
newbie extract_entities.ns input.bz2 & # Instance 2: 3 threads
newbie filter_properties.ns input.bz2 & # Instance 3: 3 threads
```

Benefits:

- **Scalability:** Better utilization of multiple cores on large datasets
- **Responsiveness:** &run thread stays available while work happens in parallel
- **Isolation:** External commands don't block internal processing

Threading Compensation Strategy Confirmation: The approach works particularly well with compressed input files, where decompression can happen in parallel with processing.

NEW: Pattern Language Implementation (&start...&end)

Based on the current implementation, here's the comprehensive pattern language specification:

Pattern Structure

```bash
&find &start=pattern &end=pattern &in file
&find &start pattern elements &end &in file
```

Assignment vs Block Mode

- `&start=text` for fast prefix matching
- `&start...&end` for complex patterns

Whitespace Control

- `&space N` for explicit spaces
- `&tab N` for explicit tabs

Example: `&find &start=Error &space 3 &numbers 4 &end=@en . &in logs.txt`

Left-to-Right Pattern Matching Algorithm

The implementation follows this streaming algorithm:

1. **Parse left anchor:** if `&start=` then leftpattern$ = characters to next &keyword

2. **Parse right anchor:** if `&end=` then rightpattern$ = characters to next &keyword
3. **Early rejection:** if leftpattern$ doesn't match line start, discard immediately
4. **Parse middle:** build mid_elements[] array with {type, count} for &numbers, &letters, etc.
5. **Scan character-by-character** left-to-right with state machine
6. **No regex, no backtracking** - pure streaming scan

Memory Constraints (CRITICAL)

- **NO Vec ANYWHERE** in data processing - causes memory explosion on large files
- **NO regex** - performance killer on large datasets
- **4KB line buffer limit** - static allocation, predictable memory usage
- **Fixed-size arrays only** - prevent dynamic allocation during processing
- **Error on lines >4KB** - "possible binary data or malformed input"

Threading Architecture

- **Thread per file:** &in triggers thread creation with isolated 4KB buffer
- **cores-1 threading:** Ryzen 7 = 15 concurrent files max
- **Static memory per thread:** 15 files × 4KB = 60KB total
- **No coordination overhead:** each thread processes independently

File Processing Integration

- **&in keyword triggers:** thread spawn, file open, decompression setup, pattern initialization
- **Transparent decompression:** magic byte detection handles .bz2, .gz automatically
- **No explicit open/close:** &in and &to manage file lifecycle
- **Example:** `&find pattern &in compressed.bz2` - decompression automatic

UTF-8 and Internationalization

- **UTF-8 native:** modern standard, no encoding detection needed
- **&letters includes:** accented characters, international text
- **&numbers includes:** Unicode digit characters from all scripts

Implementation Fixes Made

- **Fixed Vec violations in main.rs:** replaced with fixed-size arrays and slices
- **Eliminated .collect() calls:** use direct iteration instead

- **Fixed array initialization:** use `[const { None }; 64]` for non-Copy types

Performance Philosophy

- **Trade CPU for cognitive load:** 50ms vs 10ms execution acceptable if saves 5 minutes debugging
- **Streaming first:** line-by-line processing, no buffering entire files
- **Threading compensation:** utilize multiple cores to offset interpreter overhead
- **Predictable performance:** avoid algorithms with exponential behavior

Syntax Consistency

- **&end as universal terminator:** same pattern for &if...&end, &for...&end, &start...&end
- **Assignment forms:** &start=text, &end=text for precise anchoring
- **Block forms:** &start pattern &end for complex matching
- **&end requires no code:** opening keywords contain all parsing logic

Real-World Application Target

- **Wikidata processing:** multi-GB compressed files → 1 line pattern vs 300 lines Rust
- **Log analysis:** server logs, error filtering, timestamp matching
- **Text preprocessing:** the domain has enormous datasets requiring efficient processing

NEW: &in Keyword - File Input Processing

The `&in` keyword is a fundamental file processing primitive that serves as the bridge between pattern matching operations and file data sources.

Core Functionality

- **File Input Specification:** `&in filename` tells Newbie to read data from the specified file for processing by the preceding command.
- **Threading Trigger:** Each `&in` keyword spawns a dedicated processing thread with its own 4KB buffer, enabling parallel file processing.
- **Transparent Decompression:** Automatically detects compression formats via magic byte detection and handles decompression without user intervention.
- **Memory Management:** Maintains strict streaming architecture - no file content is stored in memory beyond the 4KB line buffer.

Syntax and Usage

```bash
# Basic pattern matching in files
&find error &in logs.txt
&find &start Error &space 3 &numbers 4 &end &in server.log

# Works transparently with compressed files
&find pattern &in compressed.bz2
&find &start timestamp &tab 2 &numbers &end &in logs.gz

# Multiple file processing (separate threads)
&find error &in logs1.txt
&find error &in logs2.txt
&find error &in logs3.txt.gz
```

Implementation Architecture

Thread Lifecycle:

1. **Thread Creation:** `&in filename` triggers immediate thread spawn
2. **File Opening:** Thread opens file with create_reader() for automatic decompression
3. **Buffer Allocation:** Thread gets isolated 4KB line buffer
4. **Stream Processing:** Line-by-line processing with pattern matching state machine
5. **Thread Termination:** Automatic cleanup when file processing completes

Memory Constraints:

- **4KB Line Buffer Limit:** Lines exceeding 4KB trigger "possible binary data" error
- **No File Caching:** Each line is processed and discarded immediately
- **Static Memory Allocation:** Total memory usage = (active_threads × 4KB)
- **Thread Limit:** Maximum cores-1 concurrent &in operations (e.g., 15 threads on Ryzen 7)

Error Handling:

```bash
```

```
# File not found
&find pattern &in nonexistent.txt
# Error: File not found: nonexistent.txt

# Binary data detection
&find pattern &in large_binary.exe
# Error: Line exceeds 4KB limit in large_binary.exe - possible binary data or malformed input

# Compression format unsupported
&find pattern &in file.rar
# Error: Unsupported compression format: .rar
```

Integration with Pattern Language

The `&in` keyword seamlessly integrates with the pattern matching system:

- **Left-to-Right Processing:** Pattern elements are evaluated as each character streams from the file specified by `&in`.
- **State Machine Integration:** The pattern matching state machine operates directly on the character stream from `&in`, enabling real-time matching without backtracking.
- **Early Termination:** Pattern matches can trigger early termination of file processing for efficiency.

Performance Characteristics

- **Threading Compensation:** Multiple `&in` operations run in parallel, often compensating for interpreter overhead through CPU parallelization.
- **Streaming Efficiency:** No memory bloat regardless of file size - 100MB file uses same 4KB as 1KB file.
- **Compression Performance:** Decompression happens in parallel with pattern matching, maximizing throughput.

Example Performance:

bash

```
# Single-threaded equivalent would process sequentially
&find error &in log1.txt.gz  # Thread 1: Decompress + match in parallel
&find error &in log2.txt.bz2  # Thread 2: Decompress + match in parallel
&find error &in log3.txt.xz  # Thread 3: Decompress + match in parallel
# All three files processed simultaneously, limited only by available cores
```

File Format Support

Uncompressed Files: Direct BufReader access for maximum performance

- .txt, .log, .csv, .json, .xml, .ns, etc.

Compressed Files: Automatic detection and decompression

- **.gz (gzip)** - magic bytes [0x1f, 0x8b]
- **.bz2 (bzip2)** - magic bytes [0x42, 0x5a]
- **.xz (LZMA/XZ)** - magic bytes [0xfd, 0x37, 0x7a, 0x58, 0x5a, 0x00]
- **.zst (zstandard)** - magic bytes [0x28, 0xb5, 0x2f, 0xfd]

Integration with Other Commands

Show Command Integration:

```bash
&show &in config.txt        # Display file contents
&show &in logs.gz &last 50    # Show last 50 lines of compressed file
```

Copy Command Integration:

```bash
# Future enhancement: &in as source specifier
&copy &in source.txt &to dest.txt
```

Variable Assignment:

```bash
# Future enhancement: capture file content to variables
&v.content = &show &in config.txt
```

.ns Script Integration

In .ns scripts, `&in` enables powerful data processing pipelines:

```bash
```

```
# Process multiple log files for error patterns
&find &start ERROR &space &numbers &space &letters &end &in /var/log/app1.log
&find &start ERROR &space &numbers &space &letters &end &in /var/log/app2.log.gz
&find &start ERROR &space &numbers &space &letters &end &in /var/log/app3.log.bz2

# Archive processing
&find &start User &space Login &space Failed &end &in /var/log/auth.log
```

This establishes `&in` as a critical primitive that enables Newbie's streaming architecture while maintaining the performance and memory efficiency needed for large-scale data processing.

---

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

- 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, &delete
- 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:

```bash
&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
```

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:

```bash
&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, &delete) set command.action and return Stop
- **Target Modifiers** (&to) set destination and return Continue
- **Execution Commands** (&run) execute the built command string

Working Example:

```
bash

&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:

```bash
```

```
&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. NEW: &delete Command Implementation

COMPLETE IMPLEMENTATION: The &delete command provides safe file and directory removal with admin support and comprehensive error handling.

Functionality

- **Single path argument:** `&delete path`
- **File and directory support:** Automatically detects and handles both
- **Admin integration:** `&admin &delete` uses sudo for privileged operations
- **Path expansion:** Supports tilde (~) expansion for home directory
- **Silent by default:** Use `&show &delete` for confirmation output

Implementation Details

rust

```rust
fn handle_delete(args: &[&str], command: &mut Command) -> Result<ExecutionResult, Box<dyn Error>> {
    if args.len() != 1 {
        return Err(NewbieError::new("&delete requires exactly one path argument"));
    }

    command.action = Some("delete".to_string());
    command.source = Some(args[0].to_string());
    Ok(ExecutionResult::Stop)
}

fn execute_delete_command(file_path: &str, command: &Command) -> Result<(), Box<dyn Error>> {
    let expanded_path = expand_tilde(file_path);
    let path = Path::new(&expanded_path);

    if !path.exists() {
        return Err(NewbieError::new(&format!("Path not found: {}", expanded_path)));
    }

    if command.admin_mode {
        // Use sudo rm -rf for admin operations
        let mut sudo_cmd = StdCommand::new("sudo");
        sudo_cmd.arg("rm").arg("-rf").arg(&expanded_path);
        // Execute and handle errors...
    } else {
        // Use native Rust filesystem operations
        if path.is_dir() {
            fs::remove_dir_all(&expanded_path)?;
        } else {
            fs::remove_file(&expanded_path)?;
        }
    }

    Ok(())
}
```

Examples

```bash
```

```
&delete old_file.txt              # Remove a file
&delete temp_directory/           # Remove a directory
&admin &delete /system/old_logs/  # Remove with elevated privileges
&show &delete backup.tar.gz       # Delete with confirmation output
```

Error Handling

- **Path validation:** Checks if path exists before attempting deletion
- **Permission handling:** Clear error messages for permission denied
- **Admin cleanup:** Automatically calls `sudo -k` after admin operations
- **Path expansion:** Handles tilde expansion with fallback

---

6. 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]*)

NEW: Whitespace Control Operators

&space Operator: Explicit space control with numeric arguments

- `&space` - Single space character
- `&space 2` - Exactly 2 space characters
- `&space N` - Exactly N space characters

&tab Operator: Tab character control with numeric arguments

- `&tab` - Single tab character
- `&tab 3` - Exactly 3 tab characters
- `&tab N` - Exactly N tab characters

Parsing Logic:

- Single spaces between keywords on command line = token separators for parser
- Double spaces on command line = one literal space in search pattern
- `&space N` directive = N literal spaces in search pattern
- Default behavior ignores single spaces in target text unless explicitly specified

Examples:

```bash
# Command parsing vs pattern matching distinction:
&start=Error &space 3 &numbers 5 &end
# Parser sees: [&start=Error] [&space] [3] [&numbers] [5] [&end]
# Pattern matches: "Error   22222" (Error + 3 spaces + 5 digits)

&find &start timestamp &tab 2 &numbers 4 &end &in logs.txt
# Matches lines with: timestamp<tab><tab>1234
```

Quantified Matching

All character classes support optional numeric quantifiers:

- `&text 5` - Exactly 5 text characters
- `&letters 3` - Exactly 3 letters
- `&numbers 4` - Exactly 4 numbers

Pattern Examples

```bash
&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

No backtracking required - simple state machine with left-to-right processing:

```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

Architecture Integration

The &start...&end parsing structure integrates cleanly with the broader Newbie architecture:

- **Consistent with control flow:** &if...&end, &for...&end use the same terminator pattern
- **&end requires no code:** Opening keywords (&start, &if, &for) contain all parsing logic
- **Modular design:** Each opening keyword handles its own syntax until finding &end
- **No conflicts:** Context determines which parser handles each &start...&end block

---

7. 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
```

```rust
fn create_reader(path: &str) -> Result<Box<dyn BufRead>, Box<dyn Error>> {
    let mut file = File::open(path)?;
    let mut header = [0u8; 6];
    file.read_exact(&mut header)?;
    file.seek(SeekFrom::Start(0))?; // Reset to beginning

    match &header[..] {
        [0x1f, 0x8b, ..] => {
            // gzip format detected
            Ok(Box::new(BufReader::new(GzDecoder::new(file))))
        },
        [0x42, 0x5a, ..] => {
            // bzip2 format detected
            Ok(Box::new(BufReader::new(BzDecoder::new(file))))
        },
        [0xfd, 0x37, 0x7a, 0x58, 0x5a, 0x00] => {
            // xz format detected
            Ok(Box::new(BufReader::new(XzDecoder::new(file))))
        },
        [0x28, 0xb5, 0x2f, 0xfd, ..] => {
            // zstd format detected
            Ok(Box::new(BufReader::new(ZstdDecoder::new(file))))
        },
        _ => {
            // No compression detected, use plain file
            Ok(Box::new(BufReader::new(file)))
        }
    }
}
```

Transparent 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
- **Complete transparency:** Operations like `&last N &lines`, `&first N &lines`, and pattern matching work identically on compressed and uncompressed files without any code changes
- **Circular buffer compatibility:** The `&last N` circular buffer implementation receives a stream of lines through the BufRead interface, completely unaware of underlying compression

- **Threading integration:** Each thread spawned by `&in` creates its own reader with automatic decompression

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)

---

8. Command Architecture

Show Command: Universal Display

Universal display with composable modifiers:

```bash
&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:

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

---

9. Variable System

Enhanced Implementation: Context-Aware Variable Detection

The variable system has been significantly enhanced beyond the original design with sophisticated auto-detection capabilities.

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

Auto-Detection of Variable Operations

Revolutionary Feature: The system automatically detects variable assignment vs. retrieval without requiring explicit `&set` / `&get` keywords.

Context-Aware Detection Logic:

```rust
```

```rust
fn detect_set_context(tokens: &[Option<&str>], token_count: usize, var_index: usize) -> bool {
    // Pattern 1: &v.varname = value (assignment with equals)
    if var_index + 2 < token_count {
        if tokens[var_index + 1] == Some("=") && tokens[var_index + 2].is_some() {
            return true;
        }
    }

    // Pattern 2: &v.varname value (assignment without equals)
    if var_index + 1 < token_count {
        if let Some(next_token) = tokens[var_index + 1] {
            if !next_token.starts_with('&') && next_token != "=" {
                return true; // Assignment detected
            }
        }
    }

    // Pattern 3: Variable as argument to command (GET)
    if var_index > 0 {
        if let Some(prev_token) = tokens[var_index - 1] {
            match prev_token {
                "&show" | "&find" | "&copy" | "&move" | "&run" => return false,
                "=" => return false, // Variable after equals is being read
                _ => {}
            }
        }
    }

    // Default: assume GET if no clear assignment pattern
    false
}
```

Auto-Prefix Insertion:

- `&v.username = john_doe` automatically becomes `&set &v.username john_doe`
- `&v.username` automatically becomes `&get &v.username`
- `&show &v.config` becomes `&show &get &v.config`

Namespace Organization

- **&v.** - User-defined variables
- **&system.** - System state and environment

- **&process.** - Process information
- **&network.** - Network state
- **&global.** - Cross-session configuration
- **&config.** - Application configuration

Advanced Variable Features

Variable Reference Resolution: Variables can be used as arguments to other commands:

```bash
&v.logfile = /var/log/app.log       # Auto-assignment
&find error &in &v.logfile          # Variable resolved in command
&copy &v.source &to &v.destination  # Multiple variable resolution
```

System Variable Examples:

```bash
&system.home       # User home directory
&system.pwd        # Current working directory
&process.pid       # Current process ID
&network.connected # Network connectivity status
```

Implementation in Command Struct:

```rust
pub struct Command {
    // ... other fields ...
    pub capture_output: bool,    // For variable assignment context
    pub display_output: bool,    // For &show prefix (silent by default)
    // ... rest of fields ...
}
```

---

10. Admin Command Architecture

Bounded privilege escalation with automatic cleanup:

```bash
```

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

---

11. External Script Integration

Bidirectional interoperability for gradual adoption:

From Newbie to External Scripts:

```bash
&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

---

12. Configuration Philosophy

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

```yaml
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
```

---

13. Control Flow

Indentation-based structure

Uses whitespace indentation like Python, eliminating brackets and semicolons:

```bash
```

```
&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:

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

---

14. 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

Examples:

```
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/
```

---

15. 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, run, 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 admin_mode: bool,               // For &admin prefix
    pub capture_output: bool,           // For variable assignment context
    pub display_output: bool,           // For &show prefix (silent by default)
    pub bash_command: Option<String>,      // For &run BASH commands

    // Pattern language fields
    pub pattern_start: Option<String>,     // For &start pattern
    pub pattern_end: Option<String>,       // For &end pattern
    pub input_file: Option<String>,        // For &in filename
    pub pattern_elements: Vec<PatternElement>, // For complex patterns
    pub is_assignment_form: bool,          // Track if &= was used
}

// 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: "&delete", handler: handle_delete },
    KeywordEntry { name: "&run", handler: handle_run },
    KeywordEntry { name: "&to", handler: handle_to },
    KeywordEntry { name: "&admin", handler: handle_admin },

    // Pattern language keywords
    KeywordEntry { name: "&start", handler: handle_start },
    KeywordEntry { name: "&end", handler: handle_end },
```

```
    KeywordEntry { name: "&in", handler: handle_in },
    KeywordEntry { name: "&space", handler: handle_space },
    KeywordEntry { name: "&tab", handler: handle_tab },
    KeywordEntry { name: "&numbers", handler: handle_numbers },
    KeywordEntry { name: "&letters", handler: handle_letters },
    KeywordEntry { name: "&text", handler: handle_text },
    KeywordEntry { name: "&=", handler: handle_assignment },

    // Variable system commands
    KeywordEntry { name: "&set", handler: handle_set },
    KeywordEntry { name: "&get", handler: handle_get },
    KeywordEntry { name: "&vars", handler: handle_vars },
];
```

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 - FIXED IMPLEMENTATION
const MAX_LAST_LINES: usize = 1000;
let mut line_buffer: [Option<String>; MAX_LAST_LINES] = [const { None }; MAX_LAST_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.

---

16. 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 = 32) 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**
✅ **&delete command fully implemented with admin support**
✅ **Variable system with multiple namespaces and context-aware auto-detection**
✅ **BASH command execution with &run BASH syntax**
✅ **Silent execution by default with &show modifier**
✅ **Pattern language with complete &start...&end implementation**
✅ **Magic byte detection for transparent decompression**

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) ✅
- &delete command (complete implementation) ✅
- &show command with modifiers (&numbered, &first, &last, &lines, &chars) ✅

Phase 1d - Pattern Language ✅

- &start...&end structure with line-by-line streaming constraints ✅
- &space and &tab operators with numeric arguments ✅
- All pattern elements (&letters, &numbers, &text) ✅
- &in file processing with threading architecture ✅
- Left-to-right streaming algorithm (no regex, no backtracking) ✅

Phase 1e - Variable System ✅

- Six namespaces (&v., &system., &process., &network., &global., &config.) ✅
- Context-aware variable detection ✅
- Auto-prefix insertion for assignments and retrievals ✅
- Variable resolution in command arguments ✅

Phase 2: Threading Architecture (FUTURE)

- 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
- Performance optimization and benchmarking

Phase 3: Advanced Features (FUTURE)

- Enhanced variable resolution and assignment syntax
- Cross-session persistence for &global. and &config. namespaces
- Pipeline operations (into syntax)
- Interactive debugging and profiling tools

Phase 4: Integration & Polish (FUTURE)

- GNU tool integration enhancements
- Configuration system improvements

- Error handling with enhanced suggestions
- Linux distribution packaging

---

17. 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.

---

18. 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
- Pattern language integrates cleanly with existing &end terminator system

---

19. 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

---

20. Conclusion

Newbie v0.4.1 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.

Key architectural achievements documented here:

- Magic byte detection for transparent decompression
- &run BASH syntax for quote-free command execution
- Silent execution by default with &show modifiers

- Complete pattern language with &space and &tab operators
- &delete command with full admin integration
- Context-aware variable system with auto-detection
- .ns script abstraction for complex pipelines

The unified execution model validation confirms that everything flowing through &run is architecturally sound in practice, enabling both simple command execution and complex pipeline operations through a consistent interface. The pattern language's integration with the existing &end terminator system demonstrates the architectural consistency that makes Newbie both powerful and learnable.

Implementation Status Summary:

- **Phase 1: COMPLETE** - Core architecture, all basic commands, pattern language, variable system
- **Phase 2: PLANNED** - Threading architecture and performance optimization
- **Phase 3: PLANNED** - Advanced features and cross-session persistence
- **Phase 4: PLANNED** - Integration, polish, and distribution

This document reflects the architectural decisions and implementation status as of **Phase 1 completion** with BASH support, silent execution, variable system, pattern language implementation, transparent compression integration, and complete file operations suite. The threading architecture extensions and enhanced .ns script system provide the roadmap for scaling to handle complex data processing pipelines efficiently.

The current implementation successfully validates the core design principles while providing a solid foundation for future enhancements. The system trades CPU cycles for cognitive load reduction while maintaining competitive performance through intelligent architecture decisions and streaming processing approaches.

---

21. Current Version Examples

As implemented in main.rs v0.4.1:

```bash
```

```
newbie> &run BASH echo 'Hello World'    # Execute bash command
newbie> &show &run BASH ls -la          # Execute and display output
newbie> &show src/main.rs &first 10     # Show first 10 lines
newbie> &find error &in logs.txt        # Simple text search
newbie> &find &start Error &space 3 &numbers 4 &end &in server.log # Pattern search
newbie> &delete old_backup.tar.gz       # Remove file
newbie> &admin &delete /tmp/system_cache/ # Remove with privileges
newbie> &v.logpath = /var/log/app.log   # Auto-variable assignment
newbie> &find timeout &in &v.logpath    # Use variable in command
```

This represents the complete, working implementation that exceeds the original design goals in several key areas.