# Bash Mastery 2026

## Advanced Shell Scripting, Automation, and Production Practices

# Preface

## Why This Book Exists

Bash is everywhere. It powers the startup scripts of your servers, orchestrates your CI/CD pipelines, glues together the tools that keep production systems running, and remains the default interactive shell on virtually every Linux and macOS machine in the world. And yet, most people who write Bash every day have never truly learned it.

They copy snippets from Stack Overflow. They wrap fragile one-liners in cron jobs and hope for the best. They treat Bash as a "quick and dirty" tool—then wonder why their scripts break at 3 a.m. on a Saturday.

**Bash Mastery 2026** was written to change that.

This book is for developers, DevOps engineers, SREs, and system administrators who are ready to stop writing throwaway Bash and start writing Bash that is **structured, reliable, secure, and production-worthy**. It is not an introduction to the command line. It assumes you already know your way around a terminal and have written at least a handful of shell scripts. What it offers is the path from competence to mastery.

# What You'll Find Inside

The book is organized into sixteen chapters and five appendices, moving deliberately from foundational thinking to advanced production practices.

We begin by reshaping how you *think* about Bash—not as a scripting afterthought, but as a legitimate programming environment with its own execution model, scoping rules, and architectural patterns (**Chapters 1–2**). From there, we dive deep into the constructs that form the backbone of every serious Bash program: variables, arrays, parameter expansion, and control flow (**Chapters 3–4**).

**Chapters 5–8** represent the heart of the book's philosophy: that Bash code deserves the same engineering discipline we apply to any other language. You'll learn to write modular Bash programs with clean project layouts, implement error handling that *actually works* in practice, and debug complex scripts with the confidence of an expert.

The second half of the book tackles the domains where Bash truly shines—and the boundaries where it doesn't. **Chapters 9–12** cover advanced text processing pipelines, structured data handling with modern techniques, system interaction, process control, and automation at scale. **Chapters 13–15** address the concerns that separate hobbyist scripts from production infrastructure: security hardening, deployment practices, performance optimization, and knowing when Bash has reached its limits. **Chapter 16** looks ahead, exploring how Bash continues to evolve and where it fits in the modern toolchain beyond 2026.

The **appendices** are designed as daily references—strict mode patterns, anti-pattern catalogs, production-ready templates, a debugging and logging cookbook, and a structured learning path to guide your continued growth.

# How to Read This Book

You can read it cover to cover for a comprehensive journey, or jump directly to the chapters that address your most pressing challenges. Every chapter is designed to stand on its own while contributing to a larger, cohesive understanding of Bash mastery.

Throughout the text, you'll find real-world examples, opinionated best practices, and honest assessments of where Bash excels and where you should reach for something else. This is not a book that pretends Bash is the right tool for every job. It *is* a book that will make you exceptionally good at using Bash where it belongs.

# Acknowledgments

This book owes a debt to the generations of shell programmers, open-source contributors, and the maintainers of GNU Bash itself—particularly Chet Ramey, whose decades of stewardship have kept Bash robust and relevant. Thanks also to the countless engineers whose production war stories, blog posts, and hard-won lessons informed the patterns and practices in these pages.

Most of all, thank you—the reader—for deciding that the tool you use every day is worth mastering.

Let's write better Bash.

---

*Miles Everhart, 2026*

# Table of Contents

# Chapter 1: Thinking in Bash Like a Professional

## Introduction

The difference between someone who uses Bash and someone who thinks in Bash is profound. It is the difference between a person who memorizes a handful of commands and a practitioner who understands the philosophy, the architecture, and the reasoning behind every keystroke. This chapter is designed to transform the way you approach the Bash shell. We are not going to start with syntax. We are going to start with mindset. By the end of this chapter, you will understand how professional engineers think when they sit down at a terminal, how they decompose problems, how they reason about data flow, and how they leverage the full power of Bash as a programming environment rather than a simple command prompt.

## 1.1 The Philosophy of the Unix Shell

To think in Bash like a professional, you must first understand the philosophical foundation upon which Bash was built. Bash, the Bourne Again Shell, did not emerge in a vacuum. It was born from the Unix tradition, a tradition that dates back to the early 1970s at Bell Labs. The Unix philosophy, articulated most clearly by

Doug McIlroy, Ken Thompson, and others, can be distilled into several core principles that directly shape how professionals use Bash today.

The first principle is that each program should do one thing and do it well. In Bash, this means that when you write a script or construct a pipeline, you should not attempt to build monolithic solutions. Instead, you compose small, focused tools together. The `grep` command searches text. The `sort` command sorts lines. The `uniq` command removes duplicates. Each does one thing. The professional Bash user understands that the power is not in any single command but in the composition of many commands.

The second principle is that programs should work together. This is the principle of interoperability. In Bash, the primary mechanism for this is the pipe operator. When you write `cat access.log | grep "404" | awk '{print $1}' | sort | uniq -c | sort -rn`, you are not writing one program. You are orchestrating five programs, each performing its specialized task, connected through a stream of text.

The third principle is that text is the universal interface. Unlike other programming environments where data might be passed as objects, structures, or binary formats, the Unix tradition and Bash in particular treat plain text as the lingua franca. Every command reads text from standard input and writes text to standard output. This simplicity is what makes Bash so remarkably composable.

| Principle | Description | Bash Manifestation |
|---|---|---|
| Do one thing well | Each tool has a focused purpose | Individual commands like `grep`, `sed`, `awk` |
| Programs should work together | Tools connect through standard interfaces | The pipe operator and redirection |
| Text as universal interface | Plain text is the common data format | Standard input, standard output, standard error |

| | | |
|---|---|---|
| Favor composability over features | Build complex behavior from simple parts | Pipelines, command substitution, process substitution |
| Prototype rapidly | Get something working quickly | Interactive shell, one-liners, quick scripts |

Understanding these principles is not academic. It is practical. When a professional encounters a problem, they do not immediately think about writing a 200-line script. They think about which existing tools can be composed together to solve the problem in a single pipeline. This is what it means to think in Bash.

# 1.2 How Bash Processes Commands: The Mental Model

A professional Bash user carries a mental model of how the shell processes every command. Without this model, you are guessing. With it, you can predict the behavior of complex expressions before you execute them. Let us walk through this model in detail.

When you type a command and press Enter, Bash does not simply execute what you typed. It performs a series of transformations on your input, in a specific order, before any command is actually run. Understanding this order is critical.

**Step 1: Tokenization.** Bash first breaks your input into tokens, which are words and operators. It uses spaces, tabs, and special characters to determine where one token ends and another begins. This is why quoting matters so much in Bash. A space inside double quotes is treated differently than a space outside them.

**Step 2: Command identification.** Bash determines what type of command you have entered. It could be a simple command, a pipeline, a list of commands

connected by `&&` or `||`, a compound command like a loop or conditional, or a function call.

**Step 3: Expansions.** This is where the real complexity lives, and where professionals distinguish themselves from beginners. Bash performs expansions in a specific order:

| Expansion Order | Expansion Type | Example Input | Result |
|---|---|---|---|
| 1 | Brace expansion | `file{1,2,3}.txt` | `file1.txt` `file2.txt` `file3.txt` |
| 2 | Tilde expansion | `~/documents` | `/home/user/documents` |
| 3 | Parameter and variable expansion | `$HOME` or `${var}` | The value of the variable |
| 4 | Command substitution | `$(date +%Y)` | The output of the `date` command |
| 5 | Arithmetic expansion | `$((5 + 3))` | `8` |
| 6 | Process substitution | `<(ls /tmp)` | A file descriptor containing the output |
| 7 | Word splitting | Result of unquoted expansions | Split on characters in `$IFS` |
| 8 | Filename expansion (globbing) | `*.txt` | All matching filenames |
| 9 | Quote removal | `"hello"` | `hello` (quotes stripped) |

Consider this example:

```
echo "Today is $(date +%A) and there are $(ls ~/projects/*.sh |
wc -l) scripts"
```

Bash processes this by first recognizing the double-quoted string, then performing command substitution for both `$(date +%A)` and `$(ls ~/projects/*.sh | wc -l)`, then performing tilde expansion on `~`, then performing filename expansion on `*.sh` within the subshell, and finally assembling the result. The professional understands each of these steps and can predict the output without running the command.

**Step 4: Redirection.** After expansions, Bash processes any redirection operators like `>`, `>>`, `<`, `2>&1`, and so forth. These are set up before the command executes.

**Step 5: Command execution.** Finally, Bash looks up the command. It checks in this order: aliases, functions, built-in commands, and then external commands found via the `$PATH` variable. The command is then executed with the processed arguments.

```
# Demonstrating the lookup order
type echo        # echo is a shell builtin
type ls          # ls is /usr/bin/ls (or aliased)
type my_function # my_function is a function (if defined)
```

This mental model is what allows professionals to debug problems that mystify beginners. When a variable expansion does not produce the expected result, the professional knows exactly which step in the pipeline to examine.

# 1.3 Streams, File Descriptors, and the Data Flow Paradigm

Every process in a Unix system, and therefore every command you run in Bash, has access to three standard streams. These streams are the arteries through which data flows, and understanding them is fundamental to professional Bash usage.

| File Descriptor | Name | Default Destination | Purpose |
|---|---|---|---|
| 0 | Standard Input (stdin) | Keyboard | Data fed into a command |
| 1 | Standard Output (stdout) | Terminal screen | Normal output from a command |
| 2 | Standard Error (stderr) | Terminal screen | Error messages and diagnostics |

The reason these are called file descriptors is that in Unix, everything is treated as a file. A stream is simply a file that has been opened for reading or writing. File descriptor 0 is opened for reading, while file descriptors 1 and 2 are opened for writing.

Professionals use redirection to control where these streams go:

```
# Redirect stdout to a file
ls /etc > listing.txt

# Redirect stderr to a file
ls /nonexistent 2> errors.txt

# Redirect both stdout and stderr to the same file
ls /etc /nonexistent > all_output.txt 2>&1

# The modern Bash shorthand for the above
ls /etc /nonexistent &> all_output.txt

# Redirect stdin from a file
sort < unsorted_data.txt

# Append stdout to a file instead of overwriting
echo "new entry" >> logfile.txt
```

But the professional goes further. Bash allows you to open additional file descriptors beyond the standard three. This is a technique that separates intermediate users from advanced practitioners:

```
# Open file descriptor 3 for writing to a log file
exec 3> /var/log/my_script.log

# Write to file descriptor 3
echo "Script started at $(date)" >&3
echo "Processing data..." >&3

# Normal output still goes to stdout
echo "This appears on screen"

# Close file descriptor 3
exec 3>&-
```

You can also use file descriptors for reading:

```
# Open file descriptor 4 for reading from a configuration file
exec 4< /etc/my_app.conf

# Read from file descriptor 4
while read -r line <&4; do
    echo "Config line: $line"
done

# Close file descriptor 4
exec 4<&-
```

The data flow paradigm in Bash means that you should think of every command as a transformer. Data flows in through stdin, gets transformed, and flows out through stdout. Errors flow out through stderr. When you chain commands with pipes, you are connecting the stdout of one command to the stdin of the next. This is the fundamental model of computation in Bash, and it is remarkably powerful once you internalize it.

```
# A professional pipeline demonstrating data flow
find /var/log -name "*.log" -mtime -7 -type f 2>/dev/null |
    xargs grep -l "ERROR" 2>/dev/null |
    while read -r logfile; do
        echo "=== $logfile ==="
        grep -c "ERROR" "$logfile"
```

```
done |
sort -t'=' -k2 -rn
```

In this example, notice how stderr is explicitly redirected to `/dev/null` at two points. The professional knows that `find` may encounter permission errors and that `grep` may encounter binary files, and they handle these cases explicitly rather than letting error messages pollute the output stream.

# 1.4 Thinking in Pipelines: Decomposing Problems

The hallmark of professional Bash thinking is the ability to decompose a complex problem into a sequence of simple transformations. This is pipeline thinking, and it requires practice to develop.

Let us work through a real-world example. Suppose you are asked to find the top 10 IP addresses making requests to a web server, but only for requests that resulted in a 500 status code, and only for the last 24 hours of logs.

A beginner might try to write a Python script or a complex awk program. A professional Bash user decomposes the problem:

1. Filter the log for the last 24 hours
2. Filter for 500 status codes
3. Extract the IP address field
4. Count occurrences of each IP
5. Sort by count in descending order
6. Take the top 10

Each step becomes one command in a pipeline:

```
# Assuming Apache combined log format
```

```
# Step 1: Get lines from the last 24 hours
awk -v cutoff="$(date -d '24 hours ago' '+%d/%b/%Y:%H:%M:%S')" \
    '$4 > "["cutoff' /var/log/apache2/access.log |
# Step 2: Filter for 500 status codes
awk '$9 == 500' |
# Step 3: Extract the IP address (first field)
awk '{print $1}' |
# Step 4: Sort and count
sort |
uniq -c |
# Step 5: Sort by count descending
sort -rn |
# Step 6: Take top 10
head -10
```

Notice how each step in the pipeline corresponds exactly to one step in the problem decomposition. This is not a coincidence. This is the methodology. The professional first thinks about the problem in terms of data transformations and then maps each transformation to a command.

Here is another example. Suppose you need to find all duplicate files in a directory tree based on their content, not their names:

```
find /path/to/directory -type f -exec md5sum {} + |
    sort |
    awk '{print $1}' |
    uniq -d |
    while read -r hash; do
        echo "Duplicate group (hash: $hash):"
        grep "^$hash" <(find /path/to/directory -type f -exec
md5sum {} +)
        echo ""
    done
```

The pipeline thinking approach means you can build up solutions incrementally. You start with the first command, verify its output, add the next command, verify again, and continue until the pipeline is complete. This is how professionals develop pipelines interactively at the terminal before committing them to scripts.

# 1.5 The Professional's Toolkit: Essential Concepts

Beyond philosophy and mental models, there are several concrete concepts that professionals keep at the forefront of their thinking when working in Bash.

**Quoting discipline.** The professional quotes every variable expansion by default. The command `rm $file` is dangerous. The command `rm "$file"` is safe. The difference is that without quotes, word splitting and globbing are performed on the expanded value. If `$file` contains spaces or glob characters, the unquoted version will produce unexpected and potentially destructive results.

```
# Dangerous: word splitting and globbing on the variable
file="my important file.txt"
rm $file    # This tries to remove "my", "important", and
"file.txt"

# Safe: the variable is treated as a single word
rm "$file"  # This correctly removes "my important file.txt"
```

**Exit status awareness.** Every command in Bash returns an exit status, an integer between 0 and 255. Zero means success. Any non-zero value means failure. The professional checks exit statuses and uses them for control flow:

```
if grep -q "pattern" file.txt; then
    echo "Pattern found"
else
    echo "Pattern not found"
fi

# Using exit status with && and ||
mkdir -p /tmp/workdir && cd /tmp/workdir || { echo "Failed to
create workdir"; exit 1; }
```

| Exit Status | Meaning | Common Usage |
|---|---|---|
| 0 | Success | Command completed without errors |
| 1 | General error | Catchall for miscellaneous errors |
| 2 | Misuse of shell command | Syntax errors, invalid options |
| 126 | Command cannot execute | Permission problem or not executable |
| 127 | Command not found | Typo in command name or not in PATH |
| 128+N | Fatal error signal N | Process killed by signal N |
| 130 | Terminated by Ctrl+C | Script interrupted by user (signal 2) |
| 255 | Exit status out of range | Exit value exceeded 255 |

**Defensive scripting.** The professional begins scripts with safety settings:

```
#!/usr/bin/env bash
set -euo pipefail
IFS=$'\n\t'
```

Let us break down what each of these settings does:

| Setting | Effect | Why It Matters |
|---|---|---|
| `set -e` | Exit immediately if a command exits with non-zero status | Prevents scripts from continuing after failures |
| `set -u` | Treat unset variables as an error | Catches typos in variable names |
| `set -o pipefail` | A pipeline returns the exit status of the last command to fail | Prevents silent failures in pipelines |

| | | |
|---|---|---|
| `IFS=$'\n\t'` | Set the Internal Field Separator to newline and tab only | Prevents word splitting on spaces, which is a common source of bugs |

**Note:** The `set -e` option has nuances that professionals must understand. Commands in conditionals (like `if` statements), commands followed by `&&` or `||`, and commands in subshells have different behavior under `set -e`. We will explore these nuances in detail in later chapters.

# 1.6 The Difference Between Interactive and Scripted Bash

A professional understands that Bash operates in two distinct modes, and the conventions for each are different.

In interactive mode, when you are typing commands at the terminal, brevity and speed are valued. You use aliases, you rely on tab completion, you use history expansion, and you write terse one-liners. The goal is to accomplish tasks quickly.

In scripted mode, when you are writing a script that will be executed repeatedly, possibly by other people or by automated systems, clarity, robustness, and maintainability are valued. You use full option names where possible, you add comments, you handle errors explicitly, and you validate inputs.

```
# Interactive style (acceptable at the terminal)
for f in *.log; do wc -l $f; done | sort -rn | head

# Script style (appropriate for a script file)
```

```
#!/usr/bin/env bash
set -euo pipefail

# Count lines in each log file and display the top results
readonly LOG_DIR="${1:?Usage: $0 <log_directory>}"
readonly TOP_N="${2:-10}"

if [[ ! -d "$LOG_DIR" ]]; then
    echo "Error: Directory '$LOG_DIR' does not exist" >&2
    exit 1
fi

find "$LOG_DIR" -maxdepth 1 -name "*.log" -type f -print0 |
    xargs -0 wc -l |
    sort -rn |
    head -n "$TOP_N"
```

Notice the differences in the scripted version: the shebang line, the safety settings, the use of `readonly` for constants, input validation, meaningful error messages directed to stderr, the use of `find` with `-print0` and `xargs -0` to handle filenames with special characters, and the parameterization of values that might change.

# 1.7 Practical Exercise: Building Professional Habits

Let us put these concepts into practice with a comprehensive exercise. You will build a small script that demonstrates professional Bash thinking.

**Exercise: System Health Reporter**

Create a script called `health_report.sh` that generates a brief system health report. The script should demonstrate pipeline thinking, proper quoting, exit status handling, and defensive scripting practices.

```
#!/usr/bin/env bash
set -euo pipefail
```

```
IFS=$'\n\t'

# health_report.sh - Generate a system health summary
# Usage: health_report.sh [output_file]
# If no output file is specified, output goes to stdout

readonly SCRIPT_NAME="$(basename "$0")"
readonly OUTPUT_FILE="${1:-/dev/stdout}"
readonly TIMESTAMP="$(date '+%Y-%m-%d %H:%M:%S')"

# Function to log errors to stderr
log_error() {
    echo "[ERROR] ${SCRIPT_NAME}: $1" >&2
}

# Function to generate a section header
section_header() {
    local title="$1"
    printf '\n=== %s ===\n\n' "$title"
}

# Verify we can write to the output file
if [[ "$OUTPUT_FILE" != "/dev/stdout" ]]; then
    if ! touch "$OUTPUT_FILE" 2>/dev/null; then
        log_error "Cannot write to '$OUTPUT_FILE'"
        exit 1
    fi
fi

# Begin generating the report
{
    echo "System Health Report"
    echo "Generated: $TIMESTAMP"
    echo "Hostname: $(hostname)"

    section_header "Disk Usage (Top 5 Filesystems)"
    df -h --output=target,pcent,size,used,avail 2>/dev/null |
        head -n 6 ||
        log_error "Could not retrieve disk usage"

    section_header "Memory Usage"
    if command -v free > /dev/null 2>&1; then
```

```
        free -h
    else
        log_error "free command not available"
    fi

    section_header "Top 5 CPU-Consuming Processes"
    ps aux --sort=-%cpu |
        awk 'NR<=6 {printf "%-10s %5s %5s %s\n", $1, $3, $4,
$11}' ||
        log_error "Could not retrieve process information"

    section_header "Load Average"
    uptime

    section_header "Recent Failed Login Attempts"
    if [[ -r /var/log/auth.log ]]; then
        grep -i "failed" /var/log/auth.log 2>/dev/null |
            tail -5 ||
            echo "No failed login attempts found"
    else
        echo "Auth log not readable (may require elevated
privileges)"
    fi

} > "$OUTPUT_FILE"

echo "${SCRIPT_NAME}: Report generated successfully" >&2
```

**What to observe in this exercise:**

| Technique | Where It Appears | Why It Matters |
|---|---|---|
| Defensive settings | `set -euo pipefail` | Catches errors early |
| Readonly variables | `readonly SCRIPT_NAME=...` | Prevents accidental modification |
| Default parameter values | `${1:-/dev/stdout}` | Makes the script flexible |
| Error function writing to stderr | `log_error()` | Keeps error messages separate from output |

| | | |
|---|---|---|
| Command existence check | `command -v free` | Portable way to check if a command is available |
| File readability check | `[[ -r /var/log/auth.log ]]` | Prevents errors before they happen |
| Grouped output redirection | `{ ... } > "$OUTPUT_FILE"` | Efficient single redirection for multiple commands |
| Quoted variables everywhere | `"$OUTPUT_FILE"`, `"$TIMESTAMP"` | Prevents word splitting bugs |

# 1.8 Summary and Key Takeaways

Thinking in Bash like a professional is not about memorizing more commands. It is about internalizing a set of principles and mental models that guide your decisions at every level, from how you quote a variable to how you architect a pipeline.

The Unix philosophy teaches you to compose small, focused tools. The command processing model teaches you to predict how Bash will interpret your input. The data flow paradigm teaches you to think in terms of streams and transformations. Pipeline thinking teaches you to decompose problems into sequential steps. And defensive scripting practices teach you to write code that fails safely and communicates clearly.

As you progress through the remaining chapters of this book, every technique and pattern we discuss will be built upon this foundation. The syntax will become more advanced, the patterns more sophisticated, and the applications more complex. But the thinking will remain the same: clear, deliberate, and grounded in the principles we have established here.

The terminal is not just a place to type commands. For the professional, it is an environment for thought, a place where complex problems are decomposed,

transformed, and solved with elegance and precision. That is what it means to think in Bash like a professional.

**Note:** Before proceeding to Chapter 2, take time to practice the concepts introduced here. Open a terminal and build pipelines interactively. Start with a single command, verify its output, add another command with a pipe, verify again, and continue building. This incremental, exploratory approach is how professionals develop their solutions, and it is how you will develop your intuition for Bash.

# Chapter 2: Bash Execution Model Deep Dive

## Understanding How Bash Processes Your Commands From Start to Finish

When you type a command into a Bash terminal and press Enter, a remarkably complex sequence of events unfolds behind the scenes. Most users, and even many experienced scripters, treat Bash as a simple command interpreter: you type something, it runs, and you get output. But the reality is far more nuanced. Bash follows a precise, multi-stage execution model that determines how your input is parsed, expanded, evaluated, and ultimately executed. Understanding this model is not merely academic. It is the difference between writing scripts that work by accident and writing scripts that work by design.

This chapter takes you deep inside the Bash execution engine. We will walk through every stage of command processing, examine how Bash creates and manages processes, explore the lifecycle of subshells and child processes, and dissect the mechanisms of job control. By the end of this chapter, you will have a mental model of Bash internals that allows you to predict behavior, debug subtle issues, and write scripts with confidence.

# 2.1 The Bash Command Processing Pipeline

Every line you feed to Bash, whether interactively or from a script, passes through a well-defined pipeline of processing stages. These stages happen in a specific order, and understanding that order is critical for predicting how Bash will interpret your input.

## The Stages of Command Processing

The following table outlines each stage in the order Bash processes them:

| Stage Number | Stage Name | Description |
|---|---|---|
| 1 | Tokenization (Lexical Analysis) | Bash reads the input line and breaks it into tokens: words and operators. Quoting rules are applied here to determine which characters are literal and which are special. |
| 2 | Command Identification | Bash identifies the type of command: simple command, pipeline, list, compound command, or function definition. |
| 3 | Brace Expansion | Expressions like `{a,b,c}` or `{1..10}` are expanded into multiple words. This is the first expansion stage. |
| 4 | Tilde Expansion | Leading tildes in words are replaced with home directory paths. For example, `~` becomes `/home/username`. |

| | | |
|---|---|---|
| 5 | Parameter and Variable Expansion | Variables like `$VAR`, `${VAR}`, and special parameters like `$1`, `$@`, `$?` are replaced with their values. |
| 6 | Command Substitution | Expressions enclosed in `$(...)` or backticks are executed, and their standard output replaces the expression. |
| 7 | Arithmetic Expansion | Expressions inside `$((...))` are evaluated as arithmetic, and the result replaces the expression. |
| 8 | Word Splitting | The results of unquoted parameter expansion, command substitution, and arithmetic expansion are split into separate words based on the value of `IFS`. |
| 9 | Pathname Expansion (Globbing) | Words containing `*`, `?`, or `[...]` are expanded to match filenames in the filesystem. |
| 10 | Quote Removal | Remaining quote characters that were not produced by expansion are removed from the final words. |
| 11 | Redirection Processing | Redirections like >, <, >>, and `2>&1` are set up before the command executes. |
| 12 | Command Execution | Bash determines whether the command is a builtin, a function, or an external program, and executes it accordingly. |

**Note:** The order of these stages matters enormously. Brace expansion happens before variable expansion, which means you cannot use a variable inside brace expansion and expect it to work. For example: