Chapter 1

Working on the Command Line

THE AUTOOPS+ EXAM TOPICS COVERED IN THIS CHAPTER MAY INCLUDE, BUT ARE NOT LIMITED TO, THE FOLLOWING:

✔ **1.0: Automation Coding Concepts**

- 1.1. Given a scenario, use code to support automation.
- 1.4. Given a scenario, troubleshoot common issues with the code life cycle.

✔ **2.0: System Configuration**

- 2.2. Compare and contrast basic approaches to automation.

How do you start to automate your world? Just open the command line. Whether you're on a Windows machine, macOS, Linux, or some variant, you can leave behind the mouse clicks and drag-and-drop of the user interface and take up your keyboard instead. Working from the *command line*, or *terminal*, you're able to perform most, if not all, of the tasks you'd do with the UI through text-based input—input that can be saved, corrected, rewritten, shared, and even version controlled. Working effectively on the terminal and learning its unique ecosystem of commands is the first and most important step to automating systems management and operations.

Navigating Command-Line Variants

In this book, we'll primarily focus on automating deployment, provisioning, and configuration in Unix-like environments or on Windows. This section uses examples and describes tools that will work in any POSIX-compliant operating systems. *POSIX* stands for Portable Operating System Interface (plus an X to make it feel extra Unix-y) and is a standard that defines how Unix and its many variants work under the covers. Ubuntu, CentOS, Amazon Linux, and even your macOS command line are examples of current-day POSIX-compliant systems. Still, even with POSIX compliance making these environments work very similarly, they each developed over many years along different branches of the POSIX family tree, so keep in mind that there are some variations in seemingly identical commands across them. When in doubt, append `-h` or `--help` to learn more about the supported functions of the command you're using in your particular environment.

Even if you're on Windows, it's possible to get a Unix-style terminal experience; however, Windows' journey to Unix compatibility takes some explaining.

A Brief History of the Windows Command Line

At one time, the command line on Windows was something of an afterthought. Once the visual experience of Windows came along in the '90s to provide a user interface over Disk Operating System (DOS), Microsoft placed their emphasis there, focusing on improving

that interface and relegating the terminal to a second-class citizen with limited support for keyboard shortcuts and command-line tooling. This created a challenge for those who wanted to be able to easily script and automate tasks in Windows.

The good news is that the Windows terminal experience is no longer stuck in the era of dial-up and Encarta. Microsoft has made substantial investments to bring the Windows command-line interface into the modern age, providing powerful options for both native scripting and cross-platform automation.

Improvements came in a few different ways. The year 2006 saw the first release of *PowerShell*. PowerShell is both a scripting language and a *shell* (aka command-line terminal) unto itself, providing robust commands for automating common Windows tasks. It is the definitive tool for automating Windows administration and orchestrating configuration within the Microsoft ecosystem, and we'll get into it more in the next chapter. It was the beginning of a transformed Windows command-line experience, but it didn't provide the Linux-style environment that some administrators sought. Microsoft wouldn't deliver a native Linux experience in Windows until a decade later.

Soon, other options came along, like Cygwin, which offers a POSIX-compatible terminal support as a thin layer over the traditional Windows OS, and Git Bash, first bundled with Git for Windows in 2010. This shell gave Windows developers a lightweight Bash emulation layer that included many of the most-used Unix tools, and we were happy, for a time.

The real breakthrough, though, was the introduction of the *Windows Subsystem for Linux (WSL)* in 2016. WSL allows users to run a genuine Linux environment natively on Windows, without the overhead of a virtual machine. The first version, WSL 1, provided a compatibility layer that translated Linux system calls into Windows calls, making it possible to run Linux binaries without modification. In 2019, WSL version 2 went even further, using a lightweight virtual machine with a full Linux kernel for near-native performance and compatibility.

WSL made it possible to install and run popular Linux distributions (like Ubuntu, Debian, Fedora, and more) directly from the Microsoft Store. Suddenly, Windows users could run `apt-get`, `systemd`, `bash`, and `cron` natively, without emulation or the need to dual-boot.

All of this means that Windows users now have real options. You can write PowerShell scripts to manage your system the Windows way, or you can spin up WSL and manage everything like you're on native Linux.

Mac Unix Compatibility

The story is a bit more straightforward on Macintosh operating systems. Today's macOS has its roots in BSD Unix and is fully POSIX-compliant. Most Unix or Linux tools and commands work natively on a Mac with no modification necessary. This compatibility was complicated somewhat by Apple's shift to their own ARM64-based microprocessor architecture, called Apple Silicon, but the change was largely invisible to users, as the terminal now either runs binaries compiled natively for ARM64 or runs them through

a compatibility layer called Rosetta. In either case, we can pretty universally run Linux commands on a Mac terminal in its `zsh` shell.

So now that we know how to get that Linux feeling on all the major operating systems, let's get to automating. We'll start with some important basics.

Environment Variables

Variables are the key to making any automation reusable. It's a classic programming truism that you should never hard-code any values that may change over time or from environment to environment. To see why, imagine a step in a Linux shell script that invokes an API over HTTP using `curl`:

```
curl -H "Authorization: Bearer my_api_key_123"
https://api.example.com/api/some_endpoint
```

Two elements make this script hard to reuse: the hard-coded API key and the API URL. What if you have a development or testing environment for the app? Both the API key and URL are likely to be different. Let's parameterize this script so it can work in multiple contexts. In Linux, you'd define the variables, then refer to them by name in the command:

```
export API_KEY=my_api_key_123
export API_BASE_URL=https://api.example.com/api
curl -H "Authorization: Bearer $API_KEY" $API_BASE_URL/some_endpoint
```

With this modification, those two variables can be defined outside the script, which now becomes generic and reusable. Note that *export* is needed to ensure that these variables are accessible in child shells. Without it, invoking something like `bash -c 'echo $API_BASE_URL'` wouldn't work because the value would not be in scope for the child session spawned by `bash -c`. In PowerShell, this command would look like this:

```
$env:API_KEY = "my_api_key_123"
$env:API_BASE_URL = "https://api.example.com/api"

curl -Headers @{ Authorization = "Bearer $env:API_KEY"
} "$env:API_BASE_URL/some_endpoint"
```

Fun fact: In PowerShell, `curl` is an alias for the `Invoke-WebRequest` cmdlet and not the Linux executable of the same name. They behave similarly, but this explains why the parameters are passed slightly differently from the Linux version.

Externalize Configuration

In all your automations, you should strive to externalize as much configuration as possible. This separation of configuration from code has several key benefits:

- It makes scripts easier to share with others.
- It allows you to more easily update configuration without needing to update your code.
- It often allows configuration updates without any redeployment necessary.
- It creates good habits that help avoid security concerns, like pushing code with secrets into version control.

There are many good ways of managing external configuration (in other words, of securely maintaining and setting environment variables), which we explore more in later chapters. For now, just remember the importance of using variables for any values that may change.

Slice and Dice Text Like a Pro

One of the most common and essential command-line skills for a system administrator, DevOps professional, or app developer is the ability to quickly and efficiently manipulate text. After all, everything on the command line is text, whether it's a log file, a list of processes, or an error message.

You'll want to process large volumes of log files, hunting for specific error codes, warnings, application output, or stack traces. You'll want to search through source code to find relevant lines to troubleshoot a deployment problem or debug a configuration file. You'll write scripts where you need to extract just the domain name from a URL or count how many times a specific user tried to log in (and failed).

In short, if you plan to script or automate anything, you need to master the art of slicing and dicing text.

Unix-compatible environments treat text as the lingua franca of the command line. When commands are chained together with *pipes* (`|`), it's text that flows through the pipeline. Knowing how to speak, read, and dissect that text is what turns a shell user into a shell wizard.

Regular Expressions

Regular expressions (commonly abbreviated *regex*) are a special syntax for describing patterns in text. Regex lets you describe patterns that would be tedious or impossible to match manually. Want to find every log line that contains a date? Or extract all email addresses from a file? Or match every line that starts with `ERROR` but not `OK`? It's time to break out a regex.

However, regular expression syntax can be cryptic. They're famously terse, occasionally unreadable, and fairly challenging to master. Every developer or admin knows the pain of carefully crafting a long or detailed regex with nested quotation marks or extensive character escaping. Still, learning a handful of common patterns gives you 90 percent of what you need to use tools like `grep`, `sed`, and `awk` with confidence, and to carry those skills into scripts written in any other language that supports regex.

We'll start with the basics, talking first about pattern matching in a general sense; later, you'll see how to use tools like `grep` to apply these patterns in practice. Regular expressions are often bracketed by forward slashes, so we'll use that syntax going forward.

Literal Characters

Most characters match themselves. The regex pattern `/dog/` matches the string `"dog"` and only that.

Pattern Matches

In regular expressions, certain special characters let you match longer strings, wildcards, or express where a searched-for string should appear when matching. Here are a few examples, followed by what they match or indicate:

`.` Any single character

`*` Zero or more of the previous character

`+` One or more of the previous character

`?` Zero or one of the previous character

`[]` Any one character inside the brackets

`^` Start of the line

`$` End of the line

When used in a regular expression, these elements are called *metacharacters*, and they allow us to do more than match single words:

`/^ERROR/` matches any line that starts with `ERROR`.

`/404$/` matches any line that ends in `404`.

`/[Uu]ser/` matches `User` or `user`.

`/n*/` matches the letter `n` when it occurs zero or more times. It would match `lunch`, with one `n`, `dinner` with two consecutive, and even `breakfast` with zero.

`/n+/` matches `n` when it appears one or more times, so `lunch` and `dinner`, but not `breakfast`.

We can be specific with character counts by using curly braces:

`/n{1}/` matches `lunch` only, while `/n{2}/` would match `dinner` only.

Escaping Characters

To match one of these metacharacters literally, you escape it with a backslash (`\`). When would you use this? Say you're hunting for the domain name `example.com`. Here's how it can go wrong: The expression `/example.com/` matches `example.com`, but also `examplexcom`, `example9com`, and more, because the unescaped period matches any single character. To make it match the actual period character, change your regex to `/example\.com/`.

To match `[hello]` literally, you'd escape both brackets: `/\[hello\]/`. As you can see, escaping special characters can quickly increase the complexity of regular expressions. It's important to build them carefully so your automation scripts don't end up missing important matches or over-matching strings that shouldn't be flagged.

Character Classes

Character classes let you match any one character from a set. You'll put them in square brackets:

- `[aeiou]` matches any lowercase vowel. You can put any list of characters you like to create a set.
- `[0123456789]` matches any single numeric character, but ranges, shown below, are easier.
  - `[0-9]` matches any digit.
  - `[A-Za-z]` matches any letter.
  - `[^a-z]` negates the set (matches anything but a lowercase letter).

When used inside square brackets, the `^` (caret) character has a different meaning. Alone, it indicates the beginning of a line; in conjunction with a character class, it represents negation.

There are a few shortcuts to these character classes supported by different systems. These, beginning with backslashes, are called "Perl-compatible" expressions, since they originated in that programming language.

`\d` is equivalent to `[0-9]`. Its inverse, `\D`, matches any non-digit.

`\w` matches any "word character" and is equivalent to `[a-zA-Z0-9_]`. Its inverse is `\W`.

`\s` matches whitespace characters such as tabs, spaces, and newlines. You can do the opposite with `\S`.

In addition to Perl, some tools support these out of box.

There are also POSIX character classes, which achieve many of the same shortcuts in a more verbose, but often more readable format:

`[[:digit:]]`	Digits (0–9)
`[[:alpha:]]`	Alphabetic characters
`[[:alnum:]]`	Alphanumeric (letters or digits)
`[[:space:]]`	Any whitespace (like spaces, tabs, newlines)
`[[:punct:]]`	Punctuation characters
`[[:upper:]]`	Uppercase letters
`[[:lower:]]`	Lowercase letters
`[[:xdigit:]]`	Hex digits (0–9, a–f, A–F)

Want to find email addresses in a log file? You could search with a regex pattern like this:

```
/[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-z]{2,}/
```

Let's break that down. `[a-zA-Z0-9._%+-]+` matches everything in an email that comes before the @ sign. Notice that it defines one character class (between square brackets), allowing any lowercase letter, uppercase letter, or digit, period, underscore, percent sign, plus sign, or minus sign.

The Text Manipulation Toolbox

Now that you understand regular expressions (at least enough to be dangerous!), let's look at the power trio of text manipulation: `grep`, `sed`, and `awk`.

grep

The *grep* command is ubiquitous for command-line pros: as common as a flathead screwdriver and just as useful. It's so often used that, until this moment, I didn't know how it got its name. I just thought of it as "grep," that useful command. It turns out it stands for "global regular expression print," which is itself a reference to a commonly used command in the old editor tool `ed: g/re/p --` literally, globally apply this regular expression, then print the result. Unix lore is just full of delightful historical details like this.

In any case, `grep` is one of your best friends on the command line. Let's see how it works. Here's the basic format of a `grep` command:

```
grep [options] 'pattern' filename
```

Let's look at some examples. First, to find all lines containing the word `ERROR` in a log file, you'd use:

```
grep "ERROR" app.log
```

Or to use the `-r` flag to search a directory of source code for the string `"TODO"`, enter:

```
grep -r "TODO" src/
```

You can use the `-c` flag to count matches:

```
grep -c "failed" auth.log
```

You can also use the aforementioned pipes to send text input into `grep`. The `cat` command (a binary with its own lore; it stands for "concatenate") prints out files. So let's pipe an Apache web server log into `grep` and search for "401 unauthorized" errors:

```
cat apache.log | grep '401'
```

I use `grep` all the time when checking for running processes on a Linux machine. For instance, to see if Apache (the `httpd` process) is running, use:

```
ps -ef | grep httpd
```

The first command lists all processes, with the `-ef` flag telling it to list all processes in a fully formatted list. `grep` then prints out the lines that match `httpd`. Either I get a result and know Apache is online, or I get nothing, and realize it's gone down—again. Note that the quotes are not strictly necessary to `grep`, but you'll want them whenever quotes appear in your search string and you need to remove ambiguity.

Add `-i` for case-insensitive searches and `-v` to invert the match (i.e., show lines not matching your pattern).

Using *grep* with Regex

Now, let's use `grep` to try out some regular expressions and see how `grep` supports the various styles you learned earlier. I'm using the `echo` command to output the word `dinner` (maybe I'm hungry!) and piping it as input to `grep`. Let's see what works and what doesn't.

```
> echo "dinner" | grep n{2}
# no result means no match

> echo "dinner" | grep -e 'n\{2\}'
dinner

# let's try extended mode to avoid escaping curly braces
> echo "dinner" | grep -E 'n{2}'
dinner
```

What do you learn from this? You see that `grep` needs a flag, `-e`, to use regular expressions, and that by default, every special character needs escaping. The `-E` is `grep`'s "extended" regular expression support mode, allowing you to get by without those extra backslashes.

However, I'm running this on a Mac. If you try these exercises on a native Linux machine or Windows running WSL, the supported parameters may vary. Much like our favorite comic book heroes, `grep` and its counterparts have variants.

Into the *grep*-verse

Not all `grep` commands are the same. While the name stays consistent, the behavior can shift slightly depending on your environment:

- macOS ships with BSD `grep`, which lacks support for `-P` (Perl-compatible regex) and behaves a bit differently around extended regex.
- Most Linux distros use GNU `grep`, which supports `-E`, `-P`, and other niceties. Most online examples will be in a GNU-compatible syntax.
- WSL (Windows Subsystem for Linux) runs genuine Linux binaries, so you'll get GNU `grep` by default.
- Cygwin tries to mimic GNU behavior, but compatibility isn't always perfect.
- Git Bash for Windows includes a pared-down `grep`—usually GNU-based, but sometimes with missing or different features or command parameters.

Basically, if you try a command you found online and it doesn't work on your machine, check which version of `grep` you're using with `grep --version` and check its help pages for specific parameter support.

sed

The *sed* command stands for "stream editor," which is refreshingly straightforward after `grep`. It's designed for editing text on the fly, substituting, deleting, inserting, or transforming lines as they flow from one place to another. In practicality, `sed` and `grep` can often be used to achieve similar results, but `sed` excels in a few areas that differ from `grep`.

While `grep` is a searchlight looking for patterns in text, `sed` is designed to change that text as it goes by. You feed `sed` input, describe a text transformation, and it spits out

the modified version. That's especially handy when you want to do simple replacements without firing up a text editor. Basically, `sed` is what the name implies: a stream editor that reads input line-by-line, applies a set of editing rules (which you provide), and sends the result to standard output.

The most common use case is substitution—the `s/` command—and it's one you'll see in a million shell scripts:

```
sed 's/foo/bar/' filename
```

That tells `sed` to look for the first instance of `foo` on each line and replace it with `bar`. If you want to replace all instances of `foo` on each line, add the `g` (for "global") at the end:

```
sed 's/foo/bar/g' filename
```

Substitution with `sed` comes in handy when you're trying to clean up log files, config files, or even the output of other commands. Suppose a config file uses `http://` but you're migrating everything to `https://`. Ignoring the fact that you should not still be operating unencrypted HTTP servers in this day and age (at least you're fixing it, right?), here's a `sed` one-liner to the rescue:

```
sed 's|http://|https://|g' config.yaml
```

Note the use of `|` as a delimiter—you're not stuck with slashes. This helps avoid escaping those slashes when your patterns include them, keeping your `sed` commands much more readable. Essentially, whatever character you use after the "s" will become the delimiter between sections, so choose wisely.

Also (and I digress) it's not always a crime to run plain text HTTP. Maybe your web server is safely behind a load balancer that only accepts HTTPS connections and terminates TLS before passing unencrypted traffic to a server protected inside your data center. That's fine. Just be careful out there!

The previous examples spit the modified text to *standard output* (`STDOUT`; basically, the command-line output). If you want to modify the file in place, use `-i`:

```
sed -i 's/debug=true/debug=false/' .env
```

Here's a good one for git pre-commit hooks (a topic for later in this book): Strip out trailing whitespace.

```
sed 's/[[:space:]]*$//' file.txt
```

More *sed* Examples

Here are a few quick one-liners you'll find yourself reaching for again and again.

Tabs to Spaces

```
sed 's/\t/    /g' file.txt
```

Use this code to change all tabs in `file.txt` to a series of four spaces. Just make sure you have the reverse command ready to go when your developers revolt (as if they could ever come to a consensus on this topic).

Get Rid of Blank Lines

```
sed '/^$/d' file.txt
```

This one deletes blank lines in file.txt. Note that `^$`, with nothing in between, designates the beginning of a line (`^`) followed immediately by the end of a line (`$`). The `/d` at the end tells sed to delete the matching character strings from the input stream.

Piping

```
cat app.log | sed 's/error/ERROR/g'
```

As with `grep`, you can use `cat` to output a file, then pipe it into `sed` with a vertical pipe character. This one outputs everything in `app.log`, but with every instance of `error` replaced with `ERROR`. Now, your logs can really shout at your developers when a bug needs fixing.

Don't Get Sedentary

Now that you're comfortable using sed to do simple substitutions, let's kick things up a notch.

Target Specific Lines

```
sed '1s/DEBUG/INFO/' app.log
```

In this example, you prefix the substitution command `s` with the number `1`, narrowing `sed`'s scope to just the first line. The command itself replaces any instance of `DEBUG` with `INFO` on line 1.

Target a Range of Lines

```
sed '3,6s/error/ERROR/' errors.log
```

Here, you specify only to make changes on lines 3–6, inclusive, casting `error` to uppercase.

Referencing the End of the Doc/Line

```
sed '$s/$/ Ka-chow!/' file.txt
```

This one uses the dollar sign in two distinct ways. First, when it precedes the `s` substitution command, it indicates that `sed` should target the last line of the stream. Then,

when the last line is encountered, the dollar sign works like the one in `grep`, indicating the end of the line. As a result, this `sed` command makes sure that the last word in the transformed stream is an enthusiastic `Ka-chow`!

Run Multiple Commands

```
sed '
  s/DEBUG/INFO/
  s/http:/https:/
  s/localhost/127\.0\.0\.1/
' config.yaml
```

If you use single quotes, you can give multiple commands to `sed` when you delimit them with newlines. This lets you run multiple commands (like substitutions or line deletions) at once without sacrificing legibility. This snippet runs three different substitutions, which will be executed in sequence against the incoming stream.

Inserting Lines and a More Complicated Example

You can do more than just search and replace. As with the `/d` command, you can use `/i` and `/a` to insert and append new text, respectively. Let's assume you have a web server configuration file `nginx.conf` that contains the following:

```
http {
    include       mime.types;
    default_type  application/octet-stream;
    ...(more)...
}
```

Here, we are writing a script where we want to insert a new config line into our nginx configurations (potentially something we'll copy out to many servers). We want to add a line like this:

```
log_format  main  '$remote_addr - $remote_user [$time_local] "$request"';
```

Let's use `sed` to perform this change. We'll use the `-i` that overwrites the original file, rather than just outputting modified text.

```
sed -i '/^http {/i\
log_format  main  '"'"'$remote_addr - $remote_user
[$time_local] "$request"'"'"';
' nginx.conf
```

Okay, that's a bit complicated. Let's break it down.

- `^http {` means we're finding the line that *begins* with `http {`.
- `/i` means we'll insert whatever comes next *before* the matched line.

This means this command results in the following:

```
log_format  main  '$remote_addr - $remote_user
[$time_local] "$request"';
http {
    include       mime.types;
    default_type  application/octet-stream;
    ...(more)...
}
```

How about appending something at the end? Let's create a new `location` block after the `http` block:

```
sed -i '/^}/a\
location /status {\n\
    return 200 '\''ok'\'';\n\
}
' nginx.conf
```

In nginx-speak, this creates a static URL `/status` that always returns a `200 OK` HTTP response, but all you need to understand is how the location block is appended. Again, breaking it down:

- `-i` replaces the file with the new content.
- matching on `^}` means we're looking for a closing curly brace to appear right at the start of the line (indicated by `^`).
- `/a` means we're going to append.

sed vs. *grep*

So when do you reach for `sed` instead of `grep`? Here's some general guidance:

- Use `grep` when you're filtering lines (i.e., showing only the ones that match a pattern).
- Use `sed` when you're transforming lines (i.e., changing the content but keeping all lines).

And if you need to both find and transform, you can start piping commands together into clever one-liners.

Clarity Over Cleverness

Yes, you could craft some impressively clever, lengthy command chains that accomplish wonders with nary a line break to be found. Text manipulation commands like `sed` are good at that, and there's a kind of cachet among Linux enthusiasts for those that

can craft the most powerful and esoteric one-line command. But please, resist. Because while programmers love to be clever, there's one thing far more valuable than cleverness, and that's readability. Programming is not just about accomplishing a task; it's about communication.

When you write code, you're communicating to future maintainers, and often, your future self. Fancy one-liners are like run-on sentences. They exhaust the reader and fail to communicate effectively. You or your colleagues won't want to carefully parse your past intentions from a 500-character line of `sed` script. You're not getting paid to conserve newlines, and "Faulknerian" is not an adjective to which coders should aspire.

When the task gets complicated, it's time to start writing scripts: scripts that strive for clarity, that decompose complex commands into legible, distinct functions. But we'll get to that. For now, let's look at the last of the three best-known command-line text manipulators: `awk`.

awk

With `grep`, you've got a tool for rapid pattern-matching; `sed` goes a step further and gives you powerful tools to match, replace, and edit text. Rounding out the classic trio of Unix text editing tools is *awk*, a command designed not just for pattern scanning and text processing. `awk` is often used to perform small data analysis jobs, particularly when text is laid out in rows and columns, like CSVs, TSVs, or your average log file.

Unlike `grep` or `sed`, which are shorthand for specific functions, `awk` gets its name from its creators: Alfred Aho, Peter Weinberger, and Brian Kernighan. Kernighan is also known for co-authoring the C Programming Language—no small addition to one's résumé!

Because it's technically an acronym, `awk` is sometimes rendered `AWK`, but not consistently. When you use the command on a terminal, it'll be lowercase, so we'll stick with that.

Power and Terseness

`awk` was created as a kind of successor to `sed`. Its purpose is to match patterns and then take an action on the matched text. In fact, `awk` bills itself as "the AWK Programming Language," which implies the depth contained within. It essentially has a run loop like a programming language, and it's stocked with functions that will remind you of C.

Wikipedia notes that the "power and terseness of early AWK programs . . . were important inspirations for the Perl language." Perl was one of my first coding experiences, and I've often noted that idiomatic Perl looks like the result of someone banging their head against the keyboard a few times. So as advised above, do your best to use `awk` in a way that folks can follow, or at least comment your scripts thoroughly!

I don't use `awk` as much as `sed` or `grep`, but it's powerful and useful in the right situations, so let's equip ourselves with some knowledge.

When to Use *awk*

`awk` reads input line-by-line, splits each line into fields (by default, on whitespace), and lets you write mini-programs to decide what to do with each line. It shines when you're working with structured data like:

- Tabular reports
- Space- or comma-delimited logs
- Config or log files with predictable structure
- Output of other commands like `ps`, `df`, `ls`, etc.

You'll use `awk` to filter, calculate, and summarize. It's like Excel, if Excel only had one cell and ran entirely in your terminal.

awk Basics

The canonical `awk` command looks like this:

```
awk 'pattern { action }' filename
```

Often, you're parsing the output of some other command with a pipe:

```
command | awk 'pattern { action }'
```

By default, awk splits each line into fields using whitespace, and you can reference them as `$1`, `$2`, `$3`, etc. `$0` is the entire line.

Back to our nginx example, let's say you had an nginx log file like this:

```
127.0.0.1 - - [10/Aug/2025:13:00:00 +0000] "GET /index.html HTTP/1.1" 200 612
```

To print just the HTTP method (which is the sixth field, counting from 1), use:

```
awk '{ print $6 }' nginx.log
```

The result:

```
"GET
```

You'll get a bunch of `"GET` or `"POST"` strings, with those quotes included, since they're in the raw log. Remember, `awk` is separating tokens by whitespace, so it's looking at the single spaces around that string.

To clean it up:

```
awk '{ gsub(/"/, "", $6); print $6 }' nginx.log
```

What's going on here? Note the semicolon in the middle. This indicates that, in between the brackets, we are executing two distinct commands. First, we do a `gsub`, or "global substitution" on the item at `$6`. The first parameter you'll recognize as a regex bracketed by forward slashes, matching a double-quote character. The second parameter is delimited by double quotes and is an empty string. So this `gsub` command says, "find any double-quotes in the pattern at `$6` and replace them with nothing." Then, we print `$6`.

This demonstrates an important point about `awk`, which is that these pattern-match variables have a state within your `awk` command, and those states can be mutated. `$6` starts off as one thing and ends up as another before being printed. The result:

```
GET
```

We can even use those variables in conditional statements, like in a traditional programming language. Let's look for any failed HTTP requests by checking for lines where the response code is not 200:

```
awk '$9 != 200 { print $0 }' nginx.log
```

Note that I use `$9` (the status code) to check the condition, but `print $0` for the whole matching row.

Let's mix up the delimiter `awk` uses. The `/etc/passwd` file uses colons as delimiters, not whitespace, so tell `awk` about that with the `-F` command:

```
awk -F: '{ print $1 }' /etc/passwd
```

This will take an `/etc/passwd` file like this:

```
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
alice:x:1000:1000:Alice Smith:/home/alice:/bin/bash
bob:x:1001:1001:Bob Jones:/home/bob:/bin/zsh
```

and print this:

```
root
daemon
bin
sys
alice
bob
```

This prints the username. Note that because `$0` represents the entire line, `awk` pattern variables are essentially "base 1 indexed." The first match is `$1`.

While the exam only requires that you know a bit about `awk` and when to use it, there's much more to this diverse and powerful tool. Look for any of the numerous online tutorials or e-books available, such as the one at `https://opensource.com/article/20/9/awk-ebook`, to learn more.

Structured Text: JSON, YAML, INI

While the tools we've been learning are great at helping us wrangle unstructured text, it's structured documents that make the world go 'round. Formats like JSON, YAML, and INI are the critical foundation for many systems, defining standardized ways to clearly express environment variables, configuration values, infrastructure requirements, and more. Let's start with the one that's practically everywhere: JSON.

JSON

JSON, or JavaScript Object Notation, was designed to make data interchange easy and predictable. The specification was created, at least in part, as a response to the growing complexity of XML (itself inspired by HTML), with its complex combination of opening and closing tags, angle brackets, and object attributes that so often produces a wall-of-text effect.

As an example, imagine a response from a weather API returning something like this in XML:

```
<weatherResponse>
  <location>
    <city>South Bend</city>
    <state>IN</state>
    <country>USA</country>
  </location>
  <current>
    <temperature unit="F">73</temperature>
    <condition>Partly Cloudy</condition>
    <humidity>64</humidity>
    <wind>
      <speed unit="mph">9</speed>
      <direction>SW</direction>
    </wind>
  </current>
  <forecast>
    <day name="Friday">
      <high>78</high>
```

```
      <low>62</low>
      <condition>Scattered Showers</condition>
    </day>
    <day name="Saturday">
      <high>80</high>
      <low>65</low>
      <condition>Sunny</condition>
    </day>
  </forecast>
</weatherResponse>
```

The same, in JSON, is less verbose, requiring fewer characters and providing enhanced readability:

```
{
  "location": {
    "city": "South Bend",
    "state": "IN",
    "country": "USA"
  },
  "current": {
    "temperature": {
      "value": 73,
      "unit": "F"
    },
    "condition": "Partly Cloudy",
    "humidity": 64,
    "wind": {
      "speed": 9,
      "unit": "mph",
      "direction": "SW"
    }
  },
  "forecast": [
    {
      "day": "Friday",
      "high": 78,
      "low": 62,
      "condition": "Scattered Showers"
    },
    {
      "day": "Saturday",
      "high": 80,
      "low": 65,
      "condition": "Sunny"
    }
  ]
}
```

Much nicer. JSON is also fairly permissive with its structure. For instance, this snippet:

```
"wind": {
    "speed": 9,
    "unit": "mph",
    "direction": "SW"
}
```

can just as easily be rendered on a single line, like this:

```
    "wind": {"speed": 9, "unit": "mph","direction": "SW" }
```

Both are valid, machine-parseable, equivalent JSON, but while the latter may conserve a little space, it sacrifices readability. REST responses are often "minified" in this way to reduce the number of bytes transmitted over the web. As a result, developers commonly use browser plug-ins or editor extensions that will "prettify" JSON into the nicely tabbed, clearer format shown previously.

In JSON, anything between two brackets is an object, and an object has fields or attributes. Objects can be nested within other objects. If you need to define a list of values (or a list of objects), that list must be enclosed in square brackets, delimited by commas, like so:

```
{
  "user": {
    "id": 42,
    "first_name": "Alice",
    "roles": ["admin", "editor", "viewer"],
    "profile": {
      "email": "alice@example.com",
      "preferences": {
        "theme": "dark",
        "notifications": true
      }
    }
  }
}
```

This example shows all of those conditions. The `user` object has both simple values, like `first_name`, as well as arrays, like `roles`, which has three string values separated by commas. The `profile` object has some scalar values like `email`, but also has a nested object, `preferences`. For an example of an array of objects, see the `forecast` object in the weather example earlier.

One advantage of JSON's ubiquity is that, like XML before it, it has been blessed with developer support and tooling to help make it easier to work with. An example is the command `jq`, which helps parse JSON.

Navigating JSON with *jq*

jq is a command-line JSON processor. It reads JSON input, applies a filter or transformation, and prints the result. You can think of it as `awk` or `sed`, but for structured JSON data.

Let's assume the previous JSON snippet is in the file `alice.json`. With `jq`, you can reference nested values like this (first the command, then the output):

```
jq '.user.first_name' alice.json
"Alice"
jq '.user.profile.email' alice.json
"alice@example.com"
```

To lose the quotes, include -r:

```
jq '.user.profile.email' alice.json
alice@example.com
```

You can fetch Alice's first-listed role by using square brackets:

```
jq -r '.user.roles[0]' user.json
admin
```

You can also create new JSON by filtering the original:

```
jq '{name: .user.first_name, email: .user.profile.email}' alice.json > new_
output.json
```

Here, I use the > character to pipe the usual console output into a file. This command renders the file `new_output.json` as this:

```
{
  "name": "Alice",
  "email": "alice@example.com"
}
```

As a command-line tool, `jq` is great for parsing the JSON-based output of other commands. Here, we ask the Amazon Web Services command-line interface for information on a cloud server. The output of AWS EC2 `describe-instances` returns information

about the virtual machines running in your AWS account. The output is much too verbose to show in its entirety, but it starts like this:

```
{
    "Reservations": [
        {
            "ReservationId": "r-07b991feb612345",
            "OwnerId": "03187812345",
            "Groups": [],
            "Instances": [
                {
                    "Architecture": "x86_64",
                    …many other fields…
                    "InstanceId": " i-0123456789abcdef0"
```

You need to understand the structure of your JSON before you can parse it with `jq`. We're after the instance IDs of our virtual machines. From reading the unfiltered output, we know that it returns a `Reservations` array, that each `Reservation` object has an `Instances` array, and that each `Instance` object has an `InstanceId` attribute. We can then pipe that output into `jq` and have it grab the one value we want: the instance ID.

```
aws ec2 describe-instances | jq -r
'.Reservations[].Instances[].InstanceId'
```

Using the square brackets but omitting an index, causes `jq` to fetch all array elements. The output looks a bit like this:

```
i-0123456789abcdef0
i-0fedcba9876543210
i-0fedcba9876543999
```

`jq` is extremely helpful, as are similar packages in node, Python, and other languages. The existence of such tools is a big part of JSON's appeal. Now that you know how to navigate JSON with ease, let's look at JSON's little cousin, YAML.

YAML

YAML's whole deal is that it's a data interchange format that's even more human-friendly than JSON. One of my favorite fun facts is that it stands for Yet Another Markup Language, a nod to the fact that markup languages were a dime-a-dozen back in 2001, when it was first created. Since then, the lore has been revised to say that the acronym stands for YAML Ain't Markup Language, acknowledging YAML's growing diversity of use. For what it's worth, that's what's called a recursive acronym. As with PHP, which stands for PHP: Hypertext Preprocessor, the acronym incorporates itself into its definition. Try not to think about it too long.

I love YAML. JSON is great and incredibly widely used, but JSON can be tedious and more than a bit touchy, treating you to a constant need to match curly braces, quote strings, escape special characters, and use extra tools to force it into a human-readable format—not to mention the endless comma management.

YAML, on the other hand, eschews all that. As in Python, indentation in YAML documents is syntactically significant. This means you're required to format documents well, which makes them instantly more readable. Even better, it's rarely required in YAML to enclose strings in quotation marks, so you're less often required to escape characters and are free to type in a more natural way.

Let's take our weather example and express it in YAML:

```
location:
  city: South Bend
  state: IN
  country: USA

current:
  temperature:
    value: 73
    unit: F
  condition: Partly Cloudy
  humidity: 64
  wind:
    speed: 9
    unit: mph
    direction: SW

forecast:
  - day: Friday
    high: 78
    low: 62
    condition: Scattered Showers

  - day: Saturday
    high: 80
    low: 65
    condition: Sunny
```

You can see how YAML can be a better experience. If you're getting one thing from this chapter, it's that I'm a big fan of readability. Code and configuration files are ultimately processed by computers, but they are written, read, and maintained by humans. The easier they are for people to parse, the better.

In YAML, most things are key-value pairs. Nesting, indicated in JSON by nested braces, is depicted with indentation:

```
location:
  city: South Bend
  state: IN
  country: USA
```

Indentation is commonly kept to two spaces only. Four is fine as well, but you must be consistent. Oh, and don't even think about using tabs! YAML will not allow it.

In this case, `location` comprises three fields: `city`, `state`, and `country`.

Multivalue arrays are designated with hyphens, a style YAML shares with another document markup format popular with developers, Markdown:

```
ports:
  - 80
  - 443
```

In this web server–based example, the value of `ports` is an array of two values, 80 and 443.

You Can Comment YAML

Another great feature YAML has over JSON is that it supports Python-style comments. Use a hash symbol, like this:

```
# expose ports for HTTP and HTTPS. Any HTTP traffic will be redirected to
HTTPS by
# the load balancer
ports:
  - 80
  - 443
```

You Still Need Quotes, Sometimes

In YAML, you often don't need quotation marks around values, but there are a few exceptions. Primarily, anything that uses special characters needs quotation marks. For example, in this case the use of the colon in the movie title requires the title to be enclosed in quotes:

```
film:
  title: "Mission: Impossible"
  release_year: 1996
```

The same goes for any string with brackets, asterisks, question marks, and more.

If you want values to be parsed literally, use single quotes. If you want escape characters interpolated, use double. For example:

```
# this is a newline
my_value: "\n"

# this is a backslash followed by the letter n
another_value: '\n'
```

Multi-line Strings

You can also define multi-line strings using a pipe, giving you even more flexibility in defining long values:

```
description: |
  This is a multi-line
  string in YAML.

  It preserves all line breaks,
  including blank lines.
```

You'll find YAML in all kinds of places, from Ansible playbooks to infrastructure-as-code templates. For instance, AWS's CloudFormation, which lets you define cloud architectures via text files, initially required JSON but now fully supports YAML. Guess which one I use?

INI Files

JSON and YAML are both commonly used formats for defining configuration files. However, Windows has one style of config file that goes back decades and persists to this day: the *INI file*. That's INI as in "initialization": these files were first used to tell Windows how to configure things as it started up.

INI files are wonderfully simple, defining key-value pairs using equals signs and designating sections via square brackets. INI is not really a properly defined specification in the same way as JSON or YAML, so your mileage may vary, but they look a lot like this:

```
[database]
host=localhost
port=5432
user=postgres
password=secret

# this is a comment
; this is also a comment
[server]
port=8080
debug=true
```

These files are less common today, but they still show up in some Windows contexts. For instance, a file called `desktop.ini` maintains user preferences for the display of things like hidden folders or displaying file extensions on users' desktops.

The simplicity of INI files is very similar to that found in `.env` (dotenv) files, which are often used in modern application stacks to define environment variables. Python's `dotenv` package expects a file called `.env` that might contain the following:

```
AWS_ACCESS_KEY_ID=ASIAABC123XYZPXG
AWS_SECRET_ACCESS_KEY=RA123fdakl35+fda928123
```

This format, similar to INI syntax, loads in key-value pairs and sets them in the environment.

Externalizing Secrets

One big use case for these kinds of files is externalizing configuration or secrets. You never want to embed (or hard-code) passwords, API keys, or other sensitive information in application code (or any code that goes into version control).

Not only is it a security risk, but these values can change. If a password is compromised, you want the ability to update it without the overhead of redeploying code. When your secrets or other configuration files are separated from your code, you can often update the configuration only, then reboot the software to have the new values be picked up.

Working with Application and System Logs

We've discussed how to navigate the terminal and work with common commands, we've seen how to wield powerful tools to manipulate and process text, and we've learned about the text formats that drive so much of our infrastructure's critical configuration. With this knowledge, we turn our attention to one of the most important text-based sources of system information you'll encounter: logging.

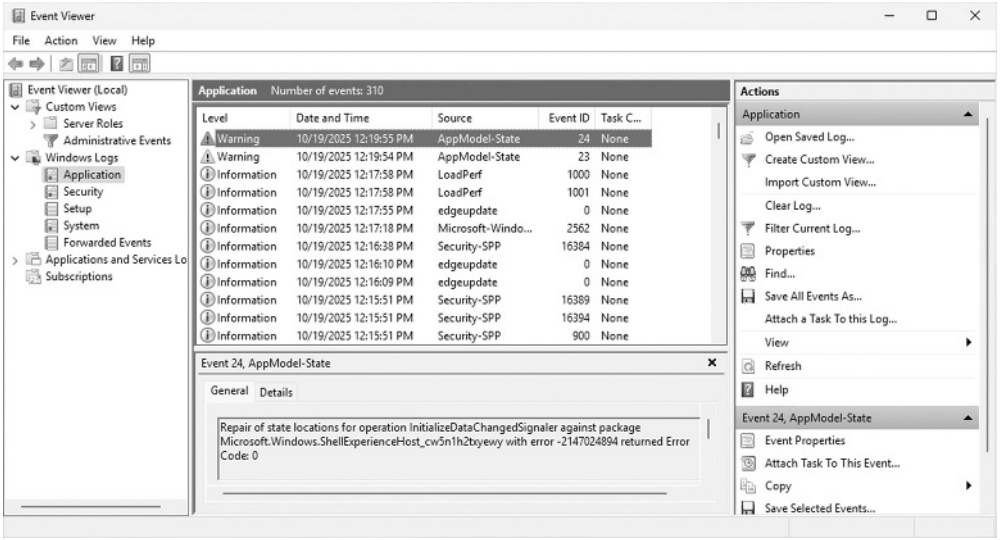
Types of Logs and Where to Find Them

System logs, server logs, application logs, and more: Your computing resources are constantly generating a chatty stream of helpful (if at times overly verbose) information. Logs are the first place you'll go to troubleshoot issues, perform security audits, and analyze performance. As a systems administrator, operations engineer, or DevOps professional, you need to know how to work with logs to successfully manage your environments.

System Logs

On Linux, you'll find systemwide activity in `/var/log/`. Windows keeps its records in the *Event Viewer* (see Figure 1.1), neatly split into categories like System, Security, and

FIGURE 1.1 The windows event viewer.



Application. These logs are your first stop when the machine itself is acting up—things like failed logins, hardware errors, or a service that mysteriously stopped at 2:00 in the morning.

Application Logs

Individual apps often keep their own journals of behavior. Frameworks like Python's `logging` module or Java's `log4j` allow developers to emit structured messages about what's happening inside their code. These are crucial for debugging; if your company's e-commerce site crashes whenever someone tries to make a purchase, detailed application logs should give you plenty of clues as to why (or at least leave you a cryptic stack trace to puzzle over). While developers decide what to log and how, using these frameworks can make it easier to parse large log dumps for relevant information.

Server Logs

Web and database servers can be especially chatty. Apache and nginx write access and error logs that show every request. Database engines like MySQL and PostgreSQL do the same for queries and performance issues. Server logs are often the first stop for performance tuning and intrusion detection.

Log Locations

Where you'll find logs depends on the environment and ecosystem in which you're working. Here are a few of the most common places to look:

- **Linux/Unix:** Almost everything funnels into `/var/log/`.
- **Windows:** Event Viewer (`eventvwr.msc`) is the control center for watching your system's logs.
- **Containers:** Logs usually print to `stdout` or `stderr`. You can monitor Docker logs at runtime with the command `docker logs`, or configure your Docker runtime to route output to another location.
- **Cloud:** Centralized services like CloudWatch (Amazon Web Services), Cloud Logging (Google Cloud Platform), and Log Analytics (Microsoft Azure) aggregate most logs in their respective cloud infrastructure systems.

In short, system logs explain the machine, application logs explain the code, server logs explain the services, and the location depends on your operating system and environment. Table 1.1 shows a few of the more common log locations for Linux variants.

Applying the Text Parsing Toolkit

Once you know where logs live, what do you do with them? Not only can application and system logs be quite verbose, but they can also grow very quickly. It can be a challenge to find what you're looking for in a sea of fast-moving logs. That's where our text parsing

TABLE 1.1 Common log locations

File	What it Tracks
`syslog` or `messages`	General system activity, kernel messages, daemons, and services
`auth.log/ secure`	Authentication attempts, sudo use, SSH logins, failures
`dmesg`	Kernel ring buffer; hardware and driver initialization
`kern.log`	Kernel-specific events (warnings, errors, panics)
`cron`	Scheduled jobs run by `cron`, including errors
`boot.log`	Boot process details, useful if startup goes sideways
`apt/`	Subdirectory of logs related to package management (Debian/Ubuntu)
`yum.log`	Package installation/removal on Red Hat/CentOS systems
`httpd/` or `nginx/`	Web server access/error logs, if those services are installed

toolkit of `grep`, `sed`, `awk`, and `jq` comes in. These tools turn a wall of log noise into something you can reason about. Let's look at a few real-world examples.

Catch Logs by the Tail

We haven't covered it yet, but the `tail` command in Unix-compatible systems shows you the last few lines of a file. Add `-f` to follow the file as it grows in real time:

```
tail -f /var/log/syslog
```

It's the system administrator's equivalent of watching a security camera feed. If you know a problem is taking place *right now*, or you have a collaborator who can trigger an error, tailing a log can be a great way to see what's going on.

Combine `tail` with `grep` for quick filtering. Maybe you're looking for web requests coming into `nginx` from a known malicious IP address (using a fake address here):

```
tail -f /var/log/nginx/access.log | grep "130.8.23.XY"
```

Now, the only thing streaming across your screen will be access log lines that contain this IP. Combining `tail` with `grep` can be a great way to get a handle on the real-time state of your system while you troubleshoot.

More Log Parsing

Let's look at some of our other commands. Our old pal `sed` is perfect for fixing text in bulk. Although best practice says that developers should avoid logging out sensitive values, you may still encounter logs that contain API keys, passwords, or other text that should not be shared. With `sed`, we can do some filtering to clean things up before passing the logs downstream for analysis. Let's redact any API keys, which in this example, we know begin with `sk-` and are always followed by a space in this log. Assuming the target lines look like this:

```
access granted with key sk-1289bDFx12 at 12:35
```

then the following command will clean them up:

```
sed 's/sk-[0-9a-zA-Z]* /\[REDACTED\] /' app.log > sanitized.log
```

Note how we're bringing in our knowledge of regular expressions to match keys that can contain numbers, lowercase letters, and uppercase letters, as well as how we must escape the square brackets so they are treated as literals. This turns each matching line into something safe, like this:

```
access granted with key [REDACTED]at 12:35
```

We can use `awk` to split Apache web server log lines into columns and process them. For a file that looks like this:

```
10.10.10.5 - - [05/Oct/2025:08:24:16 -0400] "GET /dashboard HTTP/1.1" 200 18321
"https://example.com/login" "Mozilla/5.0 (iPhone; CPU iPhone OS 17_0 like Mac
OS X)"
203.0.113.55 - - [05/Oct/2025:08:24:28 -0400] "GET /favicon.ico HTTP/1.1" 404
162 "https://example.com/" "Mozilla/5.0 (X11; Ubuntu; Linux x86_64)"
198.51.100.99 - - [05/Oct/2025:08:24:33 -0400] "PUT /api/items/129 HTTP/1.1"
201 67 "-" "PostmanRuntime/7.39.0"
```

We can get a summary of the URLs access like this:

```
awk '{print $7}' httpd.log
```

This pulls the seventh column (the URL) and prints it.

Admins use these tools daily to triage problems, generate reports, or monitor systems. A well-crafted command can save hours of log-diving, and once you've written it, you can add it to a script it for next time.

Best Practices for Log Management

Before we move on from logging, consider these additional points.

First, logs are both a powerful asset and a potential threat to your runtime system. Why? Because they can grow so quickly that they can fill up your system's drives if you're not careful. Use tools like Linux's `logrotate` to regularly archive or delete old log files to keep logs manageable.

Second, although it is somewhat beyond the scope of this book, many organizations centralize their logs into systems that aggregate information from many different sources. Sometimes this may be a locally run system, like an Elasticsearch/Logstash/Kibana (ELK) stack, a well-liked open source trio for centralizing log analytics; other times it may be a cloud service like Splunk that does the work. Often, these services provide not only log storage and access, but powerful analytics tools to help you gain more insights from your logs than you might be able to do on your own.

Summary

In this chapter, we explored the command line and dove into tools that help manipulate and extract meaning from plain text. We used regular expressions and command-line tools like `grep`, `sed`, and `awk`. You practiced parsing structured data with `jq` and learned how JSON, YAML, and INI formats are used to store configuration cleanly and predictably.

Finally, you turned these same skills toward application and system logs, the running commentary of your systems' behavior. You learned how to locate logs, tail them for real-time updates, and filter them for patterns or errors of interest.

Exam Essentials

Understand the role of text-processing tools. Remember what each tool does best: `grep` filters text, `sed` transforms it, and `awk` analyzes structured text. Be able to recognize how these tools can be combined in pipelines to clean or extract data from logs or other text sources.

Know the basics of regular expressions. Regex defines search patterns used by many automation tools. Recognize how metacharacters such as `^`, `$`, `*`, and `[]` affect pattern matching and how escaping special characters changes behavior.

Recognize structured data formats and tools. Be able to identify and read JSON, YAML, and INI syntax. Know that `jq` allows you to query and transform JSON output, such as filtering API responses or command-line tool output for specific fields.

Understand application and system log analysis. Know what logs are used for, where they're commonly located (e.g., `/var/log` on Linux and the console on macOS), and how to inspect them with commands like `tail` for streaming output. Understand how `grep`, `awk`, or `jq` can extract useful insights such as error codes or timestamps.

Know why separating configuration and secrets matters. Recognize the security and maintainability benefits of keeping credentials and environment-specific values outside of code. This prepares you for later topics in configuration management and secrets handling.

Review Questions

1. On a Windows system, you want to run Bash scripts that use common Linux commands like `grep` and `sed`. Which tool provides the most native Linux experience without needing a full virtual machine?
   - **A.** Git Bash
   - **B.** Cygwin
   - **C.** Windows Subsystem for Linux (WSL)
   - **D.** PowerShell
2. You've written a shell script that calls an API using `curl`, but you've hard-coded the API key and base URL. What is the best way to make this script reusable across multiple environments?
   - **A.** Leave the values in the script but comment them out for reference.
   - **B.** Use environment variables to externalize the API key and base URL.
   - **C.** Create separate copies of the script for each environment.
   - **D.** Store the API key in the script but obfuscate it with base64 encoding.
3. While working with logs, you want to find all lines ending in the string `404`. Which regular expression would you use with `grep`?
   - **A.** `grep '^404'`
   - **B.** `grep '/404/'`
   - **C.** `grep '404$'`
   - **D.** `grep '.*404.*'`
4. You need to replace every instance of `http://` with `https://` in a configuration file using `sed`. Which of the following commands is both correct and avoids unnecessary escaping?
   - **A.** `sed 's/http:\/\/https:\/\//g' config.yaml`
   - **B.** `sed 's|http://|https://|g' config.yaml`
   - **C.** `sed 'replace http:// with https://' config.yaml`
   - **D.** `sed 's/http:/https:/g' config.yaml`
5. You have a structured JSON file with user information. Which tool allows you to extract the user's email address directly from the command line in a clean, predictable way?
   - **A.** `awk`
   - **B.** `grep`
   - **C.** `jq`
   - **D.** `sed`

6. You're documenting config formats for your team. Which format is indentation-sensitive, supports inline comments with #, and represents lists with leading hyphens?
   - **A.** JSON
   - **B.** YAML
   - **C.** INI
   - **D.** XML

7. On macOS (BSD grep), you want to match the string `"dinner"` specifically by detecting two consecutive n characters without backslash-escaping the curly braces. Which command works?
   - **A.** `echo "dinner" | grep -e 'n{2}'`
   - **B.** `echo "dinner" | grep -E 'n{2}'`
   - **C.** `echo "dinner" | grep -P 'n{2}'`
   - **D.** `echo "dinner" | grep 'n{2}'`

8. You have a text file where each line contains several words separated by spaces. You want to print only the *sixth word* from every line. Which command is the best choice?
   - **A.** `grep "word6" file.txt`
   - **B.** `sed 's/ .*//' file.txt`
   - **C.** `awk '{ print $6 }' file.txt`
   - **D.** `jq '.fields[5]' file.txt`

9. A Bash script defines `API_BASE_URL="https://api.example.com"` and then calls `bash -c 'echo $API_BASE_URL'`, but nothing prints. What single change ensures child processes see the value?
   - **A.** Prepend the script with `set -a` (export all variables).
   - **B.** Use `export API_BASE_URL="https://api.example.com"` before invoking the child.
   - **C.** Quote the assignment: `API_BASE_URL='"https://api.example.com"'`.
   - **D.** Use `declare -g API_BASE_URL="https://api.example.com"` to force global scope.

10. When you type `ps -ef | grep httpd`, what role does the pipe (`|`) play in this command?
    - **A.** It comments out the first command.
    - **B.** It redirects standard output from the first command into the second.
    - **C.** It joins two files together for `grep` to read.
    - **D.** It prevents both commands from running simultaneously.

Answers to Review Questions

1. C. The correct answer is C. Windows Subsystem for Linux (WSL) runs genuine Linux distributions directly on Windows, offering near-native performance and compatibility. Options A and B (Git Bash and Cygwin) provide partial emulations, but not full Linux binaries. Option D, PowerShell, is powerful for Windows automation but is not Linux-compatible.

2. B. Option B is correct. Externalizing values into environment variables makes scripts reusable, portable, and more secure. Option A is incorrect because commenting values out still risks exposure. With option C, multiple copies of the script lead to poor maintainability, and option D uses base64 encoding, which does not secure secrets.

3. C. The correct answer is C. The dollar sign (`$`) in regex means end of line, so `grep '404$'` matches lines ending in `404`. Option A (`^404`) would only match lines starting with `404`. Option B is syntactically incorrect for `grep` unless using `-E` with slashes removed. Option D (`.*404.*`) matches lines containing `404` anywhere, not just at the end.

4. B. The correct answer is B. Using `|` as the delimiter keeps the command readable without escaping slashes. Option A works but is messy. Option C is not valid `sed` syntax. Option D would only replace the string `http:` and leave the rest unchanged.

5. C. The correct answer is C. `jq` is designed for parsing and filtering JSON and can directly access nested fields like `.user.profile.email`. While `awk`, `grep`, and `sed` can be made to work, they are text-based tools and not as suitable for structured JSON parsing.

6. B. The correct answer is B (YAML). YAML is indentation-significant, allows # comments, and uses leading hyphens for lists. Option A, JSON, does not support comments and isn't indentation-sensitive. Option C, INI files, support comments (# or ;) but not indentation-based structure or hyphenated lists. Option D, XML, uses tags, not indentation or hyphens, and comments use `<!-- ... -->`.

7. B. The correct answer is B. BSD `grep` supports extended regex via `-E`, which lets you use `{}` without escaping. Option A (`-e`) needs the braces escaped on many platforms. Option C uses `-P` (PCRE), which BSD grep doesn't support. Option D (no flags) treats `{` literally and won't match as intended.

8. C. The correct answer is C. `awk` automatically splits each line into fields (separated by whitespace) and lets you print any field by its position number. `$6` means "the sixth word." Option A uses grep to for text patterns, but `word6` is only going to match that literal value. In Option B, `sed` will edit the text rather than extracts columns, and `jq` (Option D) only works with structured JSON data.

9. B. The correct answer is B. Environment variables must be exported to be visible to child processes like a `bash -c` subshell. Option A uses `set -a`, which could work, but changes behavior for all subsequent assignments (not the simplest, single fix). In Option C, quoting doesn't export. Option D uses `declare -g`, which affects shell scope, not environment inheritance to child processes.

10. B. The correct answer is B. The pipe sends the standard output of `ps -ef` into `grep httpd` as input, letting `grep` filter only the matching lines. Option A confuses it with a comment symbol; C and D are incorrect interpretations of the pipe's function.