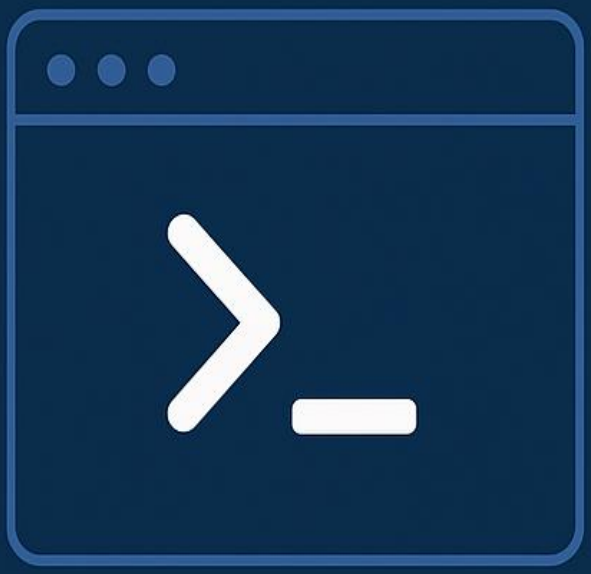
Linux Text Processing & Editing Commands



GREP

AWK

SED

CUT

SORT

UNIQ

Linux Text Processing & Search Commands

Theme: Log Analysis • Text Filtering • Automation • Troubleshooting

Linux text processing is one of the most essential skills in DevOps and system administration.
Logs, configuration files, and command outputs are all plain text so knowing how to search, filter, extract, and update text makes you highly efficient in troubleshooting and automation.

This guide explains **grep**, **awk**, and **sed** in a clean, beginner-friendly, documentation-style format with practical DevOps examples.

PART 1 GREP (Search Tool)

What is grep?

grep stands for **Global Regular Expression Print**.
It is a command-line utility used to **search text for a specific pattern**.

Whenever grep finds a match, it prints the line containing the matching pattern.

By default:

- grep shows **entire lines** where the match appears
- grep is **case-sensitive**
- grep can search **multiple files and directories**
- grep understands **regular expressions** for advanced search

Grep is fast, scalable, and ideal for processing **GB-sized log files**.

Why We Use grep

Grep helps in:

- Searching application logs for errors

- Troubleshooting production issues
- Monitoring system activity
- Filtering large text files
- Extracting specific patterns (IP, username, timestamps)
- Searching across directories, codebases, or configurations

➢ DevOps engineers often use grep inside CI/CD pipelines, Kubernetes pods, and Linux servers.

Where grep Is Used

- **/var/log** system logs
- Application debug logs
- Kubernetes pod logs
- CI/CD pipeline logs
- Configuration audits
- Security investigations

grep Syntax

grep '[pattern]' [file_name]

Explanation:

Part	Meaning
grep	The command
'pattern'	Text/regex to search
file_name	File or path to search in

Example:

grep 'cloud computing' example.txt

To search whole directory:

grep 'error' /home/user/logs/*.log

Important grep Options

OPTION	DESCRIPTION
-i	Ignore case
-v	Show lines NOT matching the pattern
-w	Match whole word

-r / -R	Search recursively in directories
-x	Match whole line
-l	List filenames containing the pattern
-L	List filenames WITHOUT the pattern
-c	Count matching lines
-o	Show only the matching part
-a	Treat binary files as text
-m N	Stop after first N matches
-A N	Show N lines *after* match
-B N	Show N lines *before* match
-C N	Show N lines *around* match
-n	Show line numbers
-E	Extended regex
-h	Hide filename in output
-q	Quiet mode (just return exit code)
-s	Silent errors

Common grep Examples

1. Search inside a file

```
grep 'bare metal' example.txt
```

2. Case-insensitive search

```
grep -i 'bare metal' example.txt
```

3. Inverse search (show NOT matching lines)

```
grep -v 'bare metal' example.txt
```

4. Search multiple files

```
grep 'server' file1.txt file2.txt file3.txt
```

5. Search all files in directory

```
grep 'ransomware' *
```

6. Search only .txt files

```
grep 'error' *.txt
```

7. Match whole word

grep -w server *

8. Recursive directory search

grep -r 'phoenix' *

9. List filenames containing pattern

grep -l 'server' *

10. Count matches

grep -c server *

11. Show only the matched part

grep -o 'server' example.txt

12. Show lines before/after match

grep -A 2 'ransomware' example.txt
grep -B 2 'ransomware' example.txt
grep -C 2 'ransomware' example.txt

13. Show line numbers

grep -n 'error' system.log

14. Search multiple patterns

grep -E 'server|cloud computing|phoenix' example.txt

15. Use with pipes

cat file.txt | grep 'server'

◍ PART 2 AWK (Field Extractor & Text Processor)

What Is awk?

awk is a powerful text-processing tool used to extract specific columns and filter rows.
It reads input **line by line**, splits it into **fields**, and allows:

- extracting columns
- filtering using conditions
- formatting output
- doing calculations

➢ Think of AWK as **Excel for the Linux terminal**.

Why We Use awk

- Extract specific fields from logs
- Filter rows based on conditions
- Format command output
- Generate reports (CPU usage, memory usage)
- Process CSV or structured text
- Parse **/etc/passwd**, Kubernetes, process lists

Where awk Is Used

- ps aux, top
- kubectl get pods
- Log analysis
- CSV, colon-separated data
- Shell script automation

awk Syntax

```
awk '{ action }' file
```

Example:

```
awk '{print $1}' file.txt
```

With delimiter:

awk -F: '{print $1,$3}' /etc/passwd

Important Awk Flag Options

Flag / Feature	Description
-F	Set field separator
$1,$2,$3…	Access fields
$0	Entire line
NR	Current line number
NF	Number of fields
BEGIN{}	Run before processing
END{}	Run after processing
if(){}	Conditional filtering
print	Print values
printf	Formatted printing
ORS	Output row separator
OFS	Output field separator
/pattern/	Filter matching lines
!/pattern/	Exclude matching lines
length($0)	Length of line
tolower()	Convert to lowercase
toupper()	Convert to uppercase
{sum+=$3}	Add numbers

Common awk Examples

1. Print specific columns

awk '{print $1,$3}' file

2. Use delimiter

awk -F: '{print $1,$3}' /etc/passwd

3. Extract process info

ps aux | awk '{print $1,$2,$11}'

4. Filter using conditions

```
awk '$3>=1000 {print $1,$3}' /etc/passwd
```

5. Count occurrences

```
awk '{count[$1]++} END{for(i in count) print i,count[i]}' file
```

🔵 PART 3 SED (Stream Editor & Auto Text Modifier)

What Is sed?

sed is a **stream editor** it edits text automatically without opening the file.

You can:

- replace text
- delete lines
- insert or append lines
- perform bulk edits
- automate config changes

➢ Think of sed as **Find & Replace for the entire Linux server**.

Why We Use sed

- Edit configuration files automatically
- Bulk-update environment values
- Remove sensitive data from logs
- Clean large log files
- Modify YAML/INI/ENV files during deployment

Where sed Is Used

- Config management
- Deployment automation
- Dockerfile/YAML updates
- Cleanup scripts

sed Syntax

sed 'command' file

In-place edit:
sed -i 's/old/new/' file

Important sed Flags & Commands

Command	Description
s/old/new/	Replace once
s/old/new/g	Replace all
-i	Edit file in place
-n	Suppress auto print
p	Print match
d	Delete lines
q	Quit
a	Append after match
i	Insert before match
c	Change entire line
/pattern/	Find pattern
/pattern/d	Delete matching lines
/pattern/p	Print matching lines
1,10p	Print line range
3d	Delete line 3
s/[0-9]//g	Remove numbers
-e	Multiple sed expressions

Common sed Examples

1. Print lines

sed -n '1,10p' file

2. Replace text

sed -i 's/nginx/apache/g' config

3. Remove DEBUG logs

sed '/DEBUG/d' logs.txt

4. Remove digits

```
sed 's/[0-9]//g' data.txt
```

🔥 COMMON SCENARIO (Grep + Awk + Sed Together)

Scenario: Payment Service Failing in Production — Investigate Logs + Extract Impacted Users + Fix Config

This is a very real DevOps situation.

A payment service is failing. Customers are complaining. You must:

- **Search logs** to find where the error is (grep)
- **Extract important data** (user IDs) from those logs (awk)
- **Fix a wrong configuration** that caused the issue (sed)

This is EXACTLY how DevOps/SRE teams combine these commands in real incidents.

🔵 Step 1 GREP: Find the Exact Error in the Logs

✔ What is GREP

Grep searches text files and shows only the lines that match your keyword.
Think of it like **Ctrl+F**, but for Linux logs.

✔ Why we use it here

The log file is huge (100 MB+).
We need only the lines containing the payment failure.

```
grep -Ri "Payment failed" /var/log/app/ > errors.log
```

✔ Output

```
2025-11-24 ERROR Payment failed for userID=1043 amount=500
2025-11-24 ERROR Payment failed for userID=2091 amount=1200
```

✔ Explanation of What's Happening

- grep searches inside ALL files in /var/log/app/
- -R = recursive (search inside subfolders)
- "Payment failed" = pattern
- saves matching lines into a new file

Now you have a **clean file** containing ONLY the error lines.

🔘 Step 2 AWK: Extract User IDs From the Error Lines

✔ What AWK does

Awk reads each line and picks specific parts (fields).
Useful for extracting numbers, words, IPs, usernames, etc.

✔ Why we use it here

Each error line contains the **user ID** that faced the payment failure.

We want only the IDs.

```
awk -F'userID=' '{print $2}' errors.log | awk '{print $1}'
```

✔ Output

```
1043
2091
```

✔ Explanation of What's Happening

- -F'userID=' → split lines at "userID="
- {print $2} → print the part AFTER "userID="
- second awk '{print $1}' → take only the number

This gives a clean list of affected users.

🔵 Step 3 SED: Fix the Incorrect Configuration Automatically

✔ What SED does

Sed edits text automatically it can replace, delete, or modify lines in a file.

✔ Why we use it here

Issue caused by:

❌ Wrong

```
retry_count=0
```

Meaning the service NEVER retries failed transactions.

Correct setting:

✔ Correct

```
retry_count=3
```

✔ Command to fix it

```
sed -i 's/retry_count=0/retry_count=3/' /etc/payment-service/config.ini
```

✔ Before

```
retry_count=0
timeout=30
```

✔ After

```
retry_count=3
timeout=30
```

✔ Explanation of What's Happening

- sed searches for retry_count=0
- replaces it with retry_count=3
- -i means save the change to the file instantly

Now the configuration is fixed across all servers.

🔵 PART 4 CUT (Extracting Fields, Characters & Bytes)

What Is cut?

The **cut command** is a Linux command-line utility used to **extract specific sections** from each line of a file or piped data.

It can extract text based on:

- **Fields** (columns separated by a delimiter)
- **Characters** (positions)
- **Bytes** (raw byte offsets)

➢ Think of cut as a **column picker** perfect for CSV files, logs, and structured text.

cut **does NOT modify** the original file it only extracts and prints selected parts.

Why We Use cut

- To extract usernames from /etc/passwd
- To extract columns from CSV files
- To extract IPs, ports, timestamps from logs
- To process command output (ps, who, ls, etc.)
- To clean & prepare data for sorting or frequency analysis

Where cut Is Used

- Log analysis
- CSV processing
- Shell scripts
- System monitoring
- Kubernetes / DevOps automation
- Output formatting

cut Command Syntax

```
cut [option] [file]
```

✔ Important Notes:

- An **option is mandatory** (e.g., -f, -c, -b)
- If you don't provide a file, cut reads from **standard input**
- You can pipe any command into cut

Example:

```
ps -e | cut -c1-5
```

cut Options

Option	Description
-f LIST	Extract fields (columns)
-d DELIM	Set delimiter (default = TAB)
-b LIST	Extract bytes
-c LIST	Extract characters
--complement	Show everything EXCEPT selected fields
-s	Skip lines without the delimiter
--output-delimiter	Set custom delimiter in output

✔ About LIST Syntax:

- N → field/character/byte number
- N-M → from N to M
- N- → from N to end
- -M → from start to M
- N,M,K → multiple values

Example:

-f 1,3,5
-c 1-4
-b 2-

CUT EXAMPLES

To understand examples better, let's create a sample CSV file:

echo -e "name,age,city\nAlice,30,New York\nBob,25,Los Angeles\nCharlie,35,Chicago" > example.csv

Contents:

name,age,city
Alice,30,New York
Bob,25,Los Angeles
Charlie,35,Chicago

1. Field Selection Based on Character

Syntax:

cut -c [LIST] [file]

Example: extract the 1st and 5th characters

cut -c1,5 example.csv

Output:

na
Ai
Bi
Ce

Explanation:

- -c1,5 → show character 1 and 5 from each line

2. Field Selection Based on Bytes

Syntax:

cut -b [LIST] [file]

Example: extract the 1st byte

cut -b 1 example.csv

Output:

n
A
B
C

Explanation:

- Useful for binary/encoded files

3. Field Selection Based on Field Numbers

Syntax:

cut -d[delimiter] -f[field_number] [file]

Example: extract the age field (second column)

cut -d',' -f2 example.csv

Output:

age
30
25
35

ADDITIONAL CUT OPTIONS

1. Complement (everything except selected fields)
cut example.csv -f 1 --complement

Output:

```
age,city
30,New York
25,Los Angeles
35,Chicago
```

2. Set custom output delimiter

```
cut -d',' -f1,3 --output-delimiter='_' example.csv
```

Output:

```
name_city
Alice_New York
Bob_Los Angeles
Charlie_Chicago
```

COMBINING CUT WITH OTHER COMMANDS

1. Combine ps with cut (process list)

```
ps -e | cut -c1-5,25-
```

Extracts:

- PID (1–5)
- Command name (25 onward)

2. Combine head + cut (extract specific rows and columns)

```
head -n 3 example.csv | cut -d ',' -f1,3
```

Output:

```
name,city
Alice,New York
Bob,Los Angeles
```

CUT WITH IRREGULAR DATA FORMATS

cut struggles when data does NOT have consistent delimiters.

Example of irregular spacing:

```
John     24    Developer
```

Lisa 29 Senior Engineer

Fix using tr -s (squeeze spaces)

```
cat data.txt | tr -s ' ' | cut -d ' ' -f1,3
```

Output:

```
John Developer
Lisa Engineer
```

Fix mixed delimiters using sed

```
sed 's/ /,/g' data.txt | cut -d ',' -f1,3
```

🔵 PART 5 SORT (Arranging & Organizing Text Data)

What Is sort?

The **sort** command in Linux arranges lines of text in **alphabetical**, **numerical**, or **custom** order.

It is one of the simplest yet most powerful text-processing tools.

- Think of sort as the Linux equivalent of **sorting rows in Excel**, but much faster and usable directly in scripts and pipelines.

Why We Use sort

- To sort large log files
- To group together similar entries before using uniq
- To sort CSV/structured data
- To identify top repeated patterns (when used with uniq)
- To clean and order output in automation scripts

Where sort Is Used

- Security logs (sort IPs)
- Process lists (sort CPU or memory usage)
- Error logs (top repeated error types)
- DevOps pipelines (sorting output before parsing)

- CSV & data processing

sort Command Syntax

sort [options] [file]

If no file is provided, sort reads from **standard input**, making it ideal in command pipelines.

Example:

cat file.txt | sort

Important sort Options

Option	Description
-n	Numeric sort
-r	Reverse (descending) order
-u	Unique sort (removes duplicates)
-k	Sort by a specific column
-t	Specify a delimiter
-f	Ignore case
-h	Human-readable sort (1K, 2M, 3G)
-V	Version sort (v1.10 < v1.2)
-o	Write result to file
--parallel=N	Multi-threaded sorting

SORT Examples

1. Alphabetical sort

sort names.txt

Example input:

banana
apple
carrot

Output:

apple
banana

```
carrot
```

2. Reverse sort

```
sort -r names.txt
```

Output:

```
carrot
banana
apple
```

3. Numeric sort

```
sort -n numbers.txt
```

Input:

```
50
2
100
12
```

Output:

```
2
12
50
100
```

4. Unique sort

```
sort -u file.txt
```

Removes duplicates **and** sorts output.

5. Sort by a column (important for CSV/logs)

Example line:

```
alice 50 developer
bob 30 designer
charlie 40 tester
```

Sort by **second column**:

```
sort -k2 file.txt
```

6. Sort using custom delimiter

```
sort -t',' -k3 example.csv
```

Sort by 3rd column in a CSV file.

7. Human-readable sort (K/M/G)

```
du -h | sort -h
```

Sorts sizes like:

```
10K
2M
500K
3G
```

SORT Summary

Command	Description
sort file	Alphabetical sort
sort -n	Numeric sort
sort -r	Reverse sort
sort -k2	Sort by field 2
sort -u	Sorted + unique

🔵 PART 6 UNIQ (Removing & Counting)

What Is uniq?

The **uniq** command filters out **duplicate lines** from sorted data.

- **IMPORTANT**

uniq only removes **adjacent duplicates**, which means the input must be **sorted first**. That is why sort | uniq is the most common pattern in Linux.

Why We Use uniq

- To find duplicate log entries
- To count how many times an IP appears
- To check repeated error messages
- To clean up lists (remove duplicates)
- To summarize data (top N repeats with sort -nr)

Where uniq Is Used

- Security (count login attempts)
- Log analysis (frequency of errors)
- Monitoring (most active users, top IPs)
- CSV cleaning
- DevOps report generation

uniq Command Syntax

uniq [options] [file]

Works best when used like:

sort file | uniq

Important uniq Options

Option	Description
-c	Count occurrences
-d	Show only duplicate lines
-u	Show only unique lines
-i	Ignore case
-f N	Skip first N fields

-s N	Skip first N characters
--group	Group duplicates visually

UNIQ Examples

1. Remove duplicates

```
sort file.txt | uniq
```

2. Count occurrences

```
sort file.txt | uniq -c
```

Output example:

```
  5 error
  2 warning
 10 info
```

3. Show only duplicates

```
sort file.txt | uniq -d
```

Shows entries that appear **more than once**.

4. Show only unique lines

```
sort file.txt | uniq -u
```

5. Frequency sort (VERY Important in DevOps)

Find the most frequent errors:

```
sort errors.log | uniq -c | sort -nr
```

Output:

```
 120 TimeoutError
 85  ConnectionRefused
 15  AuthFailure
```

❖ Conclusion

Linux text processing is one of the most valuable skills for any DevOps, SRE, or system engineer.
Whether you are troubleshooting a production outage, analyzing logs, automating deployments, or extracting meaningful data from large files commands like **grep, awk, sed, cut, sort, and uniq** give you control, speed, and clarity.

By mastering these commands, you gain the ability to:

- Diagnose issues faster
- Automate repetitive tasks
- Parse and transform data efficiently
- Handle massive log files
- Debug Kubernetes, servers, and pipelines
- Work confidently in real-world production environments

These are the **real tools DevOps engineers use daily** during on-call rotations, CI/CD troubleshooting, and log analysis.

Keep practicing the examples, try the mini-tasks, and apply the commands in small real-life scenarios.
Consistency will turn these commands into second nature and drastically improve your problem-solving abilities.

👉 **This skill alone can save hours of debugging time in production.**
👉 **Master them, and you level up as a DevOps professional.**