🐧 Day 7: Linux Command Line - The Universal Language of Data Engineering

📚 Complete Learning Guide

🎯 Learning Objectives

By the end of Day 7, you will:

- Navigate Linux systems confidently using command line
- Process and analyze large text files efficiently
- Monitor system resources and manage processes
- Connect to remote servers via SSH
- Create basic shell scripts for data automation
- Understand why Linux is the backbone of data engineering

📊 Datasets and Resources for Day 7

Primary Dataset: Server Log Files (We'll generate realistic logs)

- **Format**: Apache/Nginx access logs
- **Size**: 50,000+ log entries (~10MB)
- **Use Case**: Log analysis, text processing, pattern matching

Secondary Dataset: E-commerce Transaction Logs

- **Source**: Simulated transaction data
- **Format**: CSV and JSON mixed logs
- **Use Case:** Data parsing, file operations, automation

Required Tools:

- **Terminal/Command Prompt**: Built into all systems
- **AWS EC2 Instance**: We'll create via Console (free tier)
- **SSH Client**: Built into modern systems
- **Text Editor**: nano (beginner-friendly) or vim

Sample Log Files Sources:

- **Apache Log Format**: Standard web server logs

- **System Logs**: /var/log examples
- **Application Logs**: Custom format examples

🎓 Conceptual Understanding First (60 minutes)

Why Linux Command Line is Essential for Data Engineers

The Reality Check:

- **90%+ of cloud servers** run Linux (AWS, Google Cloud, Azure)
- **100% of big data tools** (Spark, Hadoop, Kafka) are Linux-native
- **Enterprise data pipelines** run on Linux infrastructure
- **Docker containers** are based on Linux
- **Kubernetes clusters** manage Linux containers

Command Line vs GUI:

```
GUI Approach:
Click → Navigate → Find File → Open → Process → Save
Time: 5-10 minutes for simple tasks

Command Line Approach:
grep "ERROR" logs/*.txt | wc -l
Time: 2 seconds for complex analysis
```

Real-World Data Engineering Scenarios:

1. **Log Analysis**: Process millions of log entries to find errors
2. **Data Pipeline Monitoring**: Check if processes are running correctly
3. **File Processing**: Transform CSV files before loading to databases
4. **Remote Server Management**: Connect to cloud instances for maintenance
5. **Automation**: Script repetitive data processing tasks

Linux in Modern Data Stack

Typical Data Engineering Workflow:

```
Developer Laptop (Any OS) → SSH → Linux Server → Process Data → Cloud Storage
```

Why Not Windows?:

- Linux is open source (no licensing costs for servers)
- Better resource efficiency (no GUI overhead)
- Superior text processing capabilities
- Vast ecosystem of data tools
- Industry standard for production systems

Essential Linux Concepts

File System Structure:

```
/ (root)
├── home/          # User directories
├── var/           # Variable data (logs, databases)
├── etc/           # Configuration files
├── tmp/           # Temporary files
├── usr/           # User programs
└── opt/           # Optional software
```

Everything is a File:

- Regular files (data, scripts)
- Directories (folders)
- Devices (hard drives, network)
- Processes (running programs)

🖥️ Getting Started: Setting Up Your Linux Environment (45 minutes)

Option 1: Create AWS EC2 Instance (Recommended) (30 minutes)

Why EC2?:

- Real Linux server experience
- Free tier eligible
- Same environment used in production
- SSH practice essential for data engineers

Step-by-Step EC2 Setup:

1. **Navigate to EC2**:
    - AWS Console → Services → EC2
    - Click "Launch Instance"
2. **Choose AMI (Operating System)**:
    - Name: `DataEngineering-Linux-Practice`
    - Select "Amazon Linux 2 AMI (HVM)" - Free tier eligible
    - Architecture: 64-bit (x86)
3. **Choose Instance Type**:
    - Select `t2.micro` (Free tier eligible)
    - 1 vCPU, 1 GiB Memory
4. **Configure Instance**:
    - Number of instances: 1
    - Network: Default VPC
    - Auto-assign Public IP: Enable
5. **Add Storage**:
    - Size: 8 GiB (Free tier limit)
    - Volume Type: General Purpose SSD (gp2)
6. **Configure Security Group**:
    - Create new security group
    - Name: `DataEngineering-SSH`
    - Rule: SSH (port 22) from My IP only
    - **Security Note**: Never allow SSH from 0.0.0.0/0 in production
7. **Key Pair Creation**:
    - Create new key pair
    - Name: `data-engineering-key`
    - Download the .pem file
    - **Critical**: Save this file securely - you can't download it again
8. **Launch Instance**:
    - Review all settings
    - Click "Launch Instance"
    - Wait 2-3 minutes for instance to start

Option 2: Use Local Terminal (Alternative) (15 minutes)

For Windows Users:

- Install WSL2 (Windows Subsystem for Linux)
- Windows PowerShell: `wsl --install`
- Restart computer, Ubuntu will be available

For Mac Users:

- Built-in Terminal application
- Located in Applications → Utilities

For Linux Users:

- You already have terminal access
- Ctrl+Alt+T usually opens terminal

🔗 Connecting to Your Linux Server (20 minutes)

SSH Connection Setup

What is SSH?: SSH (Secure Shell) is the standard way to securely connect to remote Linux servers. Think of it as a secure telephone line to your server.

1. **Locate Your Key File**:
    - Find the downloaded `data-engineering-key.pem`
    - Move it to a secure location (e.g., `~/.ssh/`)
2. **Set Key Permissions** (Security Requirement):

    ```bash
    # On Mac/Linux/WSL
    chmod 400 ~/.ssh/data-engineering-key.pem

    # On Windows (if using native SSH)
    icacls data-engineering-key.pem /grant:r "%username%:(R)"
    icacls data-engineering-key.pem /inheritance:r
    ```

3. **Get Instance Public IP**:
    - EC2 Console → Instances
    - Click on your instance

    - Copy "Public IPv4 address"

4. **Connect via SSH**:

    bash

    ```
    ssh -i ~/.ssh/data-engineering-key.pem ec2-user@YOUR_PUBLIC_IP

    # Example:
    # ssh -i ~/.ssh/data-engineering-key.pem ec2-user@54.123.45.67
    ```

5. **First Connection**:
    - Type "yes" when prompted about host authenticity
    - You should see a welcome message
    - Prompt changes to: `[ec2-user@ip-xxx-xxx-xxx-xxx ~]$`

Troubleshooting Connection Issues:

- Permission denied: Check key file permissions
- Connection timeout: Verify security group allows SSH from your IP
- Host key verification failed: Remove old entries from ~/.ssh/known_hosts

---

📁 Essential File Operations (60 minutes)

Basic Navigation and File Management

Current Working Directory:

bash

```
# Where am I?
pwd
# Output: /home/ec2-user

# What's in this directory?
ls
# Enhanced listing with details
ls -la
# Output shows: permissions, owner, size, date modified
```

Directory Navigation:

```bash
# Go to home directory
cd ~
# or simply
cd

# Go to root directory
cd /

# Go to previous directory
cd -

# Go up one level
cd ..

# Create directories
mkdir data
mkdir -p projects/data-engineering/day-7
# -p flag creates parent directories if they don't exist
```

File Creation and Viewing:

```bash
# Create empty file
touch sample.txt

# Create file with content
echo "Hello, Data Engineering!" > greeting.txt

# View file contents
cat greeting.txt

# View large files page by page
less /var/log/messages
# Use space to scroll down, 'q' to quit

# View first 10 lines
head greeting.txt

# View last 10 lines
tail greeting.txt

# View last 20 lines
tail -n 20 greeting.txt
```

Working with Real Data Files

Download Sample Dataset:

bash

```bash
# Create project directory
mkdir -p ~/data-engineering/day-7
cd ~/data-engineering/day-7

# Download sample e-commerce data (using curl)
curl -o sample-transactions.csv "https://raw.githubusercontent.com/plotly/datasets/mas

# Alternative: Create sample data
cat > sample-logs.txt << 'EOF'
2025-01-15 10:30:01 INFO User login successful - user_id:12345
2025-01-15 10:30:15 ERROR Database connection failed - timeout
2025-01-15 10:30:22 INFO Data processing started - batch_id:67890
2025-01-15 10:30:45 WARN Memory usage high - 85% utilized
2025-01-15 10:31:01 ERROR Payment processing failed - amount:$125.50
2025-01-15 10:31:15 INFO User logout - user_id:12345
EOF
```

File Information and Management:

```bash
# Get file information
ls -lh sample-transactions.csv
# -h flag shows human-readable file sizes

# Get file type
file sample-transactions.csv

# Count lines, words, characters
wc sample-transactions.csv
wc -l sample-transactions.csv  # Just line count

# Copy files
cp sample-logs.txt backup-logs.txt

# Move/rename files
mv backup-logs.txt logs-backup.txt

# Remove files (be careful!)
rm logs-backup.txt

# Remove directories
rmdir empty_directory
rm -rf directory_with_contents  # Use with extreme caution!
```

🔍 Text Processing and Data Analysis (75 minutes)

Searching and Filtering with grep

Basic Pattern Matching:

```bash
# Search for specific text
grep "ERROR" sample-logs.txt
# Output: Lines containing "ERROR"

# Case-insensitive search
grep -i "error" sample-logs.txt

# Show line numbers
grep -n "ERROR" sample-logs.txt

# Count occurrences
grep -c "ERROR" sample-logs.txt

# Search for multiple patterns
grep -E "ERROR|WARN" sample-logs.txt
```

Advanced grep Patterns:

```bash
# Search for lines starting with specific text
grep "^2025-01-15" sample-logs.txt

# Search for lines ending with specific text
grep "failed$" sample-logs.txt

# Search for lines containing numbers
grep "[0-9]" sample-logs.txt

# Search for email patterns (basic)
grep -E "[a-zA-Z0-9]+@[a-zA-Z0-9]+\.[a-zA-Z]{2,}" sample-logs.txt

# Search in multiple files
grep "ERROR" *.txt
```

Advanced Text Processing with awk

What is awk?: awk is a powerful text processing tool that can analyze structured data column by column.

Basic awk Usage:

```bash
# Print specific columns (space-delimited)
echo "John 25 Engineer" | awk '{print $1, $3}'
# Output: John Engineer

# Process CSV-like data
echo "John,25,Engineer" | awk -F',' '{print $1, $3}'
# -F',' sets comma as field separator

# Print line numbers with content
awk '{print NR ": " $0}' sample-logs.txt
# NR = line number, $0 = entire line
```

Real Log Analysis with awk:

```bash
# Extract timestamp and log level
awk '{print $1, $2, $3}' sample-logs.txt

# Count different log levels
awk '{print $3}' sample-logs.txt | sort | uniq -c

# Extract user IDs from logs
awk '/user_id/ {print $NF}' sample-logs.txt
# $NF = last field
```

Advanced awk for Data Processing:

```bash
# Create a more complex log file for analysis
cat > web-access.log << 'EOF'
192.168.1.100 - - [15/Jan/2025:10:30:01 +0000] "GET /api/users HTTP/1.1" 200 1234
192.168.1.101 - - [15/Jan/2025:10:30:02 +0000] "POST /api/login HTTP/1.1" 200 567
192.168.1.102 - - [15/Jan/2025:10:30:03 +0000] "GET /api/data HTTP/1.1" 404 0
192.168.1.100 - - [15/Jan/2025:10:30:04 +0000] "GET /api/users HTTP/1.1" 200 1456
192.168.1.103 - - [15/Jan/2025:10:30:05 +0000] "POST /api/upload HTTP/1.1" 500 0
EOF

# Analyze HTTP status codes
awk '{print $9}' web-access.log | sort | uniq -c
# $9 is the status code field

# Calculate total bytes transferred
awk '{sum += $10} END {print "Total bytes:", sum}' web-access.log

# Find 404 errors with IP addresses
awk '$9 == 404 {print $1, $7}' web-access.log
# $1 = IP address, $7 = requested URL

# Extract unique IP addresses
awk '{print $1}' web-access.log | sort | uniq
```

Text Transformation with sed

Basic sed Usage:

```bash
# Replace text (first occurrence per line)
sed 's/ERROR/CRITICAL/' sample-logs.txt

# Replace all occurrences
sed 's/ERROR/CRITICAL/g' sample-logs.txt

# Save changes to file
sed 's/ERROR/CRITICAL/g' sample-logs.txt > modified-logs.txt

# In-place editing (be careful!)
sed -i 's/ERROR/CRITICAL/g' sample-logs.txt

# Delete lines containing specific text
sed '/WARN/d' sample-logs.txt

# Print only lines containing specific text
sed -n '/INFO/p' sample-logs.txt
```

Advanced sed for Data Cleaning:

```bash
# Remove leading whitespace
sed 's/^[ \t]*//' file.txt

# Add line numbers
sed '=' sample-logs.txt | sed 'N;s/\n/\t/'

# Extract specific fields from structured text
sed 's/.*user_id:\([0-9]*\).*/\1/' sample-logs.txt
```

⚙️ Process Management and System Monitoring (45 minutes)

Understanding Processes

What are Processes?: Every running program on Linux is a process. As a data engineer, you'll need to monitor data pipelines, databases, and processing jobs.

Basic Process Commands:

bash

```bash
# Show running processes
ps aux
# a = all users, u = user format, x = include background processes

# Show processes in real-time
top
# Press 'q' to quit

# Better version of top
htop  # If available

# Show process tree
ps auxf

# Find specific processes
ps aux | grep python
ps aux | grep mysql
```

Process Management:

```bash
# Start a long-running process (example)
ping google.com &
# & runs process in background

# List background jobs
jobs

# Bring job to foreground
fg %1

# Send to background
bg %1

# Kill a process by PID
kill 1234

# Force kill a process
kill -9 1234

# Kill processes by name
pkill python
killall ping
```

System Resource Monitoring

Memory and CPU Usage:

```bash
# Check memory usage
free -h
# -h for human-readable format

# Check disk usage
df -h
# Shows disk space usage by filesystem

# Check directory sizes
du -sh *
# -s for summary, -h for human-readable

# Monitor system resources
vmstat 1 5
# Update every 1 second, 5 times

# Network connections
netstat -tuln
# -t TCP, -u UDP, -l listening, -n numeric
```

Log File Monitoring:

```bash
# Watch log files in real-time
tail -f /var/log/messages

# Follow multiple log files
tail -f /var/log/messages /var/log/secure

# Monitor with line numbers
tail -fn 50 /var/log/messages

# Watch for specific patterns
tail -f /var/log/messages | grep ERROR
```

Simulating Real Data Engineering Scenarios

Scenario 1: Monitor a Data Processing Job:

```bash
# Simulate a long-running data process
cat > data_processor.sh << 'EOF'
#!/bin/bash
echo "Starting data processing job..."
for i in {1..100}; do
    echo "Processing batch $i of 100"
    sleep 1
done
echo "Data processing complete!"
EOF

# Make script executable
chmod +x data_processor.sh

# Run in background
./data_processor.sh &

# Monitor the job
jobs
ps aux | grep data_processor
```

Scenario 2: Analyze System Performance During Data Load:

```bash
# Start monitoring system resources
vmstat 1 > system_performance.log &

# Simulate heavy disk I/O (data loading)
dd if=/dev/zero of=large_file.dat bs=1M count=100

# Stop monitoring
kill %1

# Analyze the performance data
cat system_performance.log
```

🔧 File Permissions and Security (30 minutes)

Understanding Linux Permissions

Permission System:

bash

```
# Check file permissions
ls -l sample-logs.txt
# Output: -rw-r--r-- 1 ec2-user ec2-user 1234 Jan 15 10:30 sample-logs.txt

# Permission breakdown:
# - = file type (- for file, d for directory)
# rw- = owner permissions (read, write, no execute)
# r-- = group permissions (read only)
# r-- = other permissions (read only)
```

Permission Numbers:

- r (read) = 4
- w (write) = 2
- x (execute) = 1

Common Permission Patterns:

bash

```
# Make file readable/writable by owner only
chmod 600 private_data.txt

# Make file readable by everyone
chmod 644 public_data.txt

# Make script executable
chmod +x script.sh
# or
chmod 755 script.sh

# Remove all permissions for others
chmod 770 sensitive_file.txt

# Recursive permission change
chmod -R 755 directory/
```

Ownership Management:

bash

```
# Change file owner (requires sudo)
sudo chown username:group file.txt

# Change group only
sudo chgrp newgroup file.txt

# Recursive ownership change
sudo chown -R username:group directory/
```

Security Best Practices for Data Engineers

File Security:

bash

```
# Create secure directory for credentials
mkdir -p ~/.config/credentials
chmod 700 ~/.config/credentials

# Store database passwords securely
echo "db_password=secret123" > ~/.config/credentials/db.conf
chmod 600 ~/.config/credentials/db.conf

# Never store credentials in scripts!
# Instead, read from secure files:
cat > secure_script.sh << 'EOF'
#!/bin/bash
source ~/.config/credentials/db.conf
echo "Connecting to database with password: $db_password"
EOF
```

📜 Introduction to Shell Scripting (45 minutes)

Basic Shell Script Structure

Your First Data Processing Script:

bash

```
# Create a script for daily log analysis
cat > log_analyzer.sh << 'EOF'
#!/bin/bash

# Daily Log Analysis Script
# Purpose: Analyze web server logs for errors and traffic patterns

echo "=== Daily Log Analysis Report ==="
echo "Generated on: $(date)"
echo

# Check if log file exists
if [ ! -f "web-access.log" ]; then
    echo "Error: web-access.log not found!"
    exit 1
fi

# Count total requests
total_requests=$(wc -l < web-access.log)
echo "Total requests: $total_requests"

# Count by status code
echo
echo "=== Status Code Summary ==="
awk '{print $9}' web-access.log | sort | uniq -c | sort -nr

# Top 5 IP addresses
echo
echo "=== Top 5 IP Addresses ==="
awk '{print $1}' web-access.log | sort | uniq -c | sort -nr | head -5

# Error analysis
error_count=$(awk '$9 >= 400' web-access.log | wc -l)
echo
echo "=== Error Analysis ==="
echo "Total errors (4xx/5xx): $error_count"

if [ $error_count -gt 0 ]; then
    echo "Error details:"
    awk '$9 >= 400 {print $1, $9, $7}' web-access.log
fi

echo
```

```
echo "=== Report Complete ==="
EOF

# Make script executable
chmod +x log_analyzer.sh

# Run the script
./log_analyzer.sh
```

Advanced Script with Functions:

bash

```
cat > data_pipeline.sh << 'EOF'
#!/bin/bash

# Data Pipeline Automation Script
# Purpose: Download, process, and upload data

# Configuration
DATA_DIR="./data"
PROCESSED_DIR="./processed"
LOG_FILE="pipeline.log"

# Function to log messages
log_message() {
    echo "$(date '+%Y-%m-%d %H:%M:%S') - $1" | tee -a $LOG_FILE
}

# Function to create directories
setup_directories() {
    log_message "Setting up directories"
    mkdir -p $DATA_DIR $PROCESSED_DIR
}

# Function to download data
download_data() {
    log_message "Starting data download"

    # Simulate data download
    for i in {1..5}; do
        echo "data_file_$i,value_$i,$(date)" > $DATA_DIR/file_$i.csv
        log_message "Downloaded file_$i.csv"
    done

    log_message "Data download complete"
}

# Function to process data
process_data() {
    log_message "Starting data processing"

    for file in $DATA_DIR/*.csv; do
        filename=$(basename $file)
        # Simple processing: add header and convert to uppercase
        echo "FILE,VALUE,TIMESTAMP" > $PROCESSED_DIR/$filename
        tail -n +1 $file | tr '[:lower:]' '[:upper:]' >> $PROCESSED_DIR/$filename
        log_message "Processed $filename"
    done

    log_message "Data processing complete"
}

# Function to generate summary
generate_summary() {
    log_message "Generating summary report"

    echo "=== Data Pipeline Summary ===" > summary.txt
    echo "Execution Date: $(date)" >> summary.txt
    echo "Files Processed: $(ls $PROCESSED_DIR/*.csv | wc -l)" >> summary.txt
    echo "Total Records: $(cat $PROCESSED_DIR/*.csv | wc -l)" >> summary.txt

    log_message "Summary report generated"
}

# Main execution
main() {
    log_message "Starting data pipeline"

    setup_directories
    download_data
    process_data
    generate_summary

    log_message "Data pipeline complete"
    echo "Check summary.txt for results"
}

# Error handling
set -e  # Exit on error
trap 'log_message "Pipeline failed with error"' ERR

# Run main function
main
EOF

chmod +x data_pipeline.sh
./data_pipeline.sh
```

Variables and Control Structures

Working with Variables:

```bash
# Environment variables
echo $HOME
echo $USER
echo $PATH

# Custom variables
name="Data Engineer"
echo "Hello, $name"

# Command substitution
current_date=$(date)
echo "Today is $current_date"

# Arrays (useful for processing multiple files)
files=("file1.txt" "file2.txt" "file3.txt")
for file in "${files[@]}"; do
    echo "Processing $file"
done
```

Conditional Logic:

bash

```bash
cat > file_checker.sh << 'EOF'
#!/bin/bash

file_to_check="$1"

if [ -z "$file_to_check" ]; then
    echo "Usage: $0 <filename>"
    exit 1
fi

if [ -f "$file_to_check" ]; then
    echo "File $file_to_check exists"
    file_size=$(ls -lh "$file_to_check" | awk '{print $5}')
    echo "Size: $file_size"

    if [ -r "$file_to_check" ]; then
        echo "File is readable"
        line_count=$(wc -l < "$file_to_check")
        echo "Lines: $line_count"
    else
        echo "File is not readable"
    fi
else
    echo "File $file_to_check does not exist"
fi
EOF

chmod +x file_checker.sh
./file_checker.sh sample-logs.txt
```

🌐 Network Operations and File Transfer (30 minutes)

Working with Remote Files

Download Files with curl and wget:

bash

```bash
# Download with curl
curl -o dataset.csv "https://raw.githubusercontent.com/plotly/datasets/master/iris.csv"

# Download with wget (if available)
wget https://raw.githubusercontent.com/plotly/datasets/master/tips.csv

# Download and extract compressed files
curl -L "https://github.com/plotly/datasets/archive/master.zip" -o datasets.zip
unzip datasets.zip
```

Transfer Files to/from S3 (if AWS CLI installed):

bash

```bash
# Install AWS CLI on Amazon Linux
sudo yum install awscli -y

# Configure AWS CLI (use your credentials from Day 6)
aws configure

# Upload file to S3
aws s3 cp sample-logs.txt s3://your-bucket-name/processed/day7-logs/

# Download from S3
aws s3 cp s3://your-bucket-name/raw/superstore/Sample\ -\ Superstore.csv ./

# Sync directories
aws s3 sync ./processed/ s3://your-bucket-name/processed/day7/
```

Network Monitoring and Troubleshooting

Network Diagnostics:

```bash
# Test connectivity
ping -c 4 google.com

# Test specific ports
telnet google.com 80

# Check network configuration
ifconfig  # or 'ip addr show' on newer systems

# Monitor network traffic
netstat -i  # Interface statistics

# Check listening ports
ss -tuln
# Modern replacement for netstat
```

🧪 Real-World Data Engineering Scenarios (60 minutes)

Scenario 1: Log Analysis Pipeline

Setting Up Realistic Log Data:

bash

```
# Create a log generator script
cat > generate_logs.sh << 'EOF'
#!/bin/bash

# Generate realistic web server logs
LOG_FILE="production.log"
IPS=("192.168.1.100" "192.168.1.101" "192.168.1.102" "10.0.0.50" "10.0.0.51")
ENDPOINTS=("/api/users" "/api/orders" "/api/products" "/login" "/logout" "/dashboard")
STATUS_CODES=(200 200 200 404 500 401)
USER_AGENTS=("Mozilla/5.0" "Chrome/96.0" "Safari/14.0")

for i in {1..1000}; do
    ip=${IPS[$RANDOM % ${#IPS[@]}]}
    endpoint=${ENDPOINTS[$RANDOM % ${#ENDPOINTS[@]}]}
    status=${STATUS_CODES[$RANDOM % ${#STATUS_CODES[@]}]}
    size=$((RANDOM % 5000 + 100))
    timestamp=$(date -d "$((RANDOM % 3600)) seconds ago" '+%d/%b/%Y:%H:%M:%S +0000')

    echo "$ip - - [$timestamp] \"GET $endpoint HTTP/1.1\" $status $size" >> $LOG_FILE
done

echo "Generated 1000 log entries in $LOG_FILE"
EOF

chmod +x generate_logs.sh
./generate_logs.sh
```

Real-Time Log Analysis:

bash

```
# Create comprehensive log analyzer
cat > production_log_analyzer.sh << 'EOF'
#!/bin/bash

LOG_FILE="production.log"
ALERT_THRESHOLD=10

echo "=== Production Log Analysis ==="
echo "Analyzing: $LOG_FILE"
echo "Generated: $(date)"
echo

# Basic statistics
total_requests=$(wc -l < $LOG_FILE)
echo "Total requests: $total_requests"

# Time range analysis
first_request=$(head -1 $LOG_FILE | awk '{print $4}' | tr -d '[]')
last_request=$(tail -1 $LOG_FILE | awk '{print $4}' | tr -d '[]')
echo "Time range: $first_request to $last_request"
echo

# Status code distribution
echo "=== Status Code Distribution ==="
awk '{print $9}' $LOG_FILE | sort | uniq -c | sort -nr

# Error analysis
error_count=$(awk '$9 >= 400' $LOG_FILE | wc -l)
echo
echo "=== Error Analysis ==="
echo "Total errors: $error_count"

if [ $error_count -gt $ALERT_THRESHOLD ]; then
    echo "🚨 ALERT: Error count ($error_count) exceeds threshold ($ALERT_THRESHOLD)"
    echo "Top error sources:"
    awk '$9 >= 400 {print $1}' $LOG_FILE | sort | uniq -c | sort -nr | head -5
fi

# Traffic analysis
echo
echo "=== Traffic Analysis ==="
echo "Top 5 IP addresses:"
awk '{print $1}' $LOG_FILE | sort | uniq -c | sort -nr | head -5
```

```
echo
echo "Top 5 endpoints:"
awk '{print $7}' $LOG_FILE | sort | uniq -c | sort -nr | head -5

# Performance analysis
echo
echo "=== Performance Analysis ==="
avg_response_size=$(awk '{sum += $10; count++} END {print sum/count}' $LOG_FILE)
echo "Average response size: ${avg_response_size} bytes"

# Hourly traffic pattern
echo
echo "=== Hourly Traffic Pattern ==="
awk '{
    gsub(/\[|\]/, "", $4);
    split($4, dt, ":");
    hour = dt[2];
    traffic[hour]++
} END {
    for (h in traffic) {
        printf "%02d:00 - %d requests\n", h, traffic[h]
    }
}' $LOG_FILE | sort

echo
echo "=== Analysis Complete ==="
EOF

chmod +x production_log_analyzer.sh
./production_log_analyzer.sh
```

Scenario 2: Data File Processing Pipeline

CSV Data Processing Workflow:

bash

```
# Create sample e-commerce data
cat > create_sample_data.sh << 'EOF'
#!/bin/bash

# Generate sample e-commerce transaction data
TRANSACTIONS_FILE="transactions.csv"
CUSTOMERS_FILE="customers.csv"

# Create transactions file
echo "transaction_id,customer_id,product_name,quantity,price,transaction_date" > $TRANS

for i in {1..500}; do
    trans_id="TXN$(printf "%06d" $i)"
    customer_id=$((RANDOM % 100 + 1))
    products=("Laptop" "Mouse" "Keyboard" "Monitor" "Webcam" "Headphones")
    product=${products[$RANDOM % ${#products[@]}]}
    quantity=$((RANDOM % 5 + 1))
    price=$((RANDOM % 500 + 50))
    days_ago=$((RANDOM % 30))
    date=$(date -d "$days_ago days ago" '+%Y-%m-%d')

    echo "$trans_id,$customer_id,$product,$quantity,$price,$date" >> $TRANSACTIONS_FIL
done

# Create customers file
echo "customer_id,customer_name,email,city,signup_date" > $CUSTOMERS_FILE

for i in {1..100}; do
    name="Customer_$i"
    email="customer$i@example.com"
    cities=("New York" "Los Angeles" "Chicago" "Houston" "Phoenix")
    city=${cities[$RANDOM % ${#cities[@]}]}
    days_ago=$((RANDOM % 365))
    signup_date=$(date -d "$days_ago days ago" '+%Y-%m-%d')

    echo "$i,$name,$email,$city,$signup_date" >> $CUSTOMERS_FILE
done

echo "Generated $TRANSACTIONS_FILE (500 records)"
echo "Generated $CUSTOMERS_FILE (100 records)"
EOF
```

```
chmod +x create_sample_data.sh
./create_sample_data.sh
```

Advanced Data Processing Script:

bash

```
cat > data_processor.sh << 'EOF'
#!/bin/bash

# E-commerce Data Processing Pipeline
# Purpose: Clean, validate, and analyze transaction data

set -e  # Exit on error

# Configuration
INPUT_DIR="."
OUTPUT_DIR="./processed"
REPORTS_DIR="./reports"
LOG_FILE="processing.log"

# Create directories
mkdir -p $OUTPUT_DIR $REPORTS_DIR

# Logging function
log() {
    echo "$(date '+%Y-%m-%d %H:%M:%S') - $1" | tee -a $LOG_FILE
}

# Data validation function
validate_csv() {
    local file=$1
    local expected_columns=$2

    log "Validating $file"

    if [ ! -f "$file" ]; then
        log "ERROR: File $file not found"
        return 1
    fi

    actual_columns=$(head -1 "$file" | tr ',' '\n' | wc -l)

    if [ $actual_columns -ne $expected_columns ]; then
        log "ERROR: Expected $expected_columns columns, found $actual_columns in $file"
        return 1
    fi

    log "✓ $file validation passed"
    return 0
```

```
}

# Clean transaction data
clean_transactions() {
    log "Cleaning transaction data"

    # Remove duplicates, sort by date
    (head -1 transactions.csv; tail -n +2 transactions.csv | sort -t, -k6) > $OUTPUT_D

    # Create summary statistics
    log "Generating transaction statistics"

    echo "=== Transaction Data Summary ===" > $REPORTS_DIR/transaction_summary.txt
    echo "Generated: $(date)" >> $REPORTS_DIR/transaction_summary.txt
    echo >> $REPORTS_DIR/transaction_summary.txt

    # Total records
    total_records=$(($(wc -l < $OUTPUT_DIR/transactions_clean.csv) - 1))
    echo "Total transactions: $total_records" >> $REPORTS_DIR/transaction_summary.txt

    # Date range
    echo "Date range:" >> $REPORTS_DIR/transaction_summary.txt
    tail -n +2 $OUTPUT_DIR/transactions_clean.csv | awk -F, '{print $6}' | sort | head
    echo " to " >> $REPORTS_DIR/transaction_summary.txt
    tail -n +2 $OUTPUT_DIR/transactions_clean.csv | awk -F, '{print $6}' | sort | tail

    # Product analysis
    echo >> $REPORTS_DIR/transaction_summary.txt
    echo "=== Product Sales ===" >> $REPORTS_DIR/transaction_summary.txt
    tail -n +2 $OUTPUT_DIR/transactions_clean.csv | awk -F, '{
        product[$3] += $4 * $5
    } END {
        for (p in product) {
            printf "%s: $%.2f\n", p, product[p]
        }
    }' | sort -t: -k2 -nr >> $REPORTS_DIR/transaction_summary.txt

    log "✓ Transaction cleaning complete"
}

# Analyze customer data
analyze_customers() {
    log "Analyzing customer data"
    # Customer city distribution
    echo "=== Customer Analysis ===" > $REPORTS_DIR/customer_analysis.txt
    echo "Generated: $(date)" >> $REPORTS_DIR/customer_analysis.txt
    echo >> $REPORTS_DIR/customer_analysis.txt

    echo "Customers by city:" >> $REPORTS_DIR/customer_analysis.txt
    tail -n +2 customers.csv | awk -F, '{print $4}' | sort | uniq -c | sort -nr >> $RE

    # Customer lifetime value (simplified)
    echo >> $REPORTS_DIR/customer_analysis.txt
    echo "=== Customer Lifetime Value (Top 10) ===" >> $REPORTS_DIR/customer_analysis.

    # Join transactions with customers and calculate CLV
    tail -n +2 transactions.csv | awk -F, '{
        customer_total[$2] += $4 * $5
    } END {
        for (c in customer_total) {
            printf "Customer %d: $%.2f\n", c, customer_total[c]
        }
    }' | sort -t: -k2 -nr | head -10 >> $REPORTS_DIR/customer_analysis.txt

    log "✓ Customer analysis complete"
}

# Generate daily sales report
daily_sales_report() {
    log "Generating daily sales report"

    echo "=== Daily Sales Report ===" > $REPORTS_DIR/daily_sales.txt
    echo "Generated: $(date)" >> $REPORTS_DIR/daily_sales.txt
    echo >> $REPORTS_DIR/daily_sales.txt

    tail -n +2 $OUTPUT_DIR/transactions_clean.csv | awk -F, '{
        daily_sales[$6] += $4 * $5
        daily_transactions[$6]++
    } END {
        for (date in daily_sales) {
            printf "%s: $%.2f (%d transactions)\n", date, daily_sales[date], daily_tra
        }
    }' | sort >> $REPORTS_DIR/daily_sales.txt

    log "✓ Daily sales report complete"
}
```

```
# Main execution
main() {
    log "Starting data processing pipeline"

    # Validate input files
    validate_csv "transactions.csv" 6 || exit 1
    validate_csv "customers.csv" 5 || exit 1

    # Process data
    clean_transactions
    analyze_customers
    daily_sales_report

    # Generate final summary
    echo "=== Processing Summary ===" > $REPORTS_DIR/processing_summary.txt
    echo "Pipeline executed: $(date)" >> $REPORTS_DIR/processing_summary.txt
    echo "Files processed:" >> $REPORTS_DIR/processing_summary.txt
    echo "- transactions.csv ($(wc -l < transactions.csv) rows)" >> $REPORTS_DIR/proce
    echo "- customers.csv ($(wc -l < customers.csv) rows)" >> $REPORTS_DIR/processing_
    echo >> $REPORTS_DIR/processing_summary.txt
    echo "Output files generated:" >> $REPORTS_DIR/processing_summary.txt
    ls -lh $OUTPUT_DIR/ $REPORTS_DIR/ >> $REPORTS_DIR/processing_summary.txt

    log "Data processing pipeline complete"
    log "Check $REPORTS_DIR/ for analysis results"
}

# Run main function
main
EOF

chmod +x data_processor.sh
./data_processor.sh
```

Scenario 3: System Health Monitoring

Comprehensive System Monitor:

bash

```
cat > system_monitor.sh << 'EOF'
#!/bin/bash

# System Health Monitor for Data Engineering Infrastructure
# Purpose: Monitor system resources and alert on issues

ALERT_THRESHOLD_CPU=80
ALERT_THRESHOLD_MEM=85
ALERT_THRESHOLD_DISK=90
LOG_FILE="system_health.log"
REPORT_FILE="health_report.txt"

log_alert() {
    echo "$(date '+%Y-%m-%d %H:%M:%S') ALERT: $1" | tee -a $LOG_FILE
}

log_info() {
    echo "$(date '+%Y-%m-%d %H:%M:%S') INFO: $1" | tee -a $LOG_FILE
}

check_cpu_usage() {
    # Get CPU usage (simplified)
    cpu_usage=$(top -bn1 | grep "Cpu(s)" | awk '{print $2}' | cut -d'%' -f1)

    if (( $(echo "$cpu_usage > $ALERT_THRESHOLD_CPU" | bc -l) )); then
        log_alert "High CPU usage: ${cpu_usage}%"
        return 1
    else
        log_info "CPU usage normal: ${cpu_usage}%"
        return 0
    fi
}

check_memory_usage() {
    # Get memory usage percentage
    mem_usage=$(free | awk 'NR==2{printf "%.0f", $3*100/$2}')

    if [ $mem_usage -gt $ALERT_THRESHOLD_MEM ]; then
        log_alert "High memory usage: ${mem_usage}%"
        return 1
    else
        log_info "Memory usage normal: ${mem_usage}%"
        return 0
    fi
}

check_disk_usage() {
    # Check root filesystem usage
    disk_usage=$(df / | awk 'NR==2 {print $5}' | cut -d'%' -f1)

    if [ $disk_usage -gt $ALERT_THRESHOLD_DISK ]; then
        log_alert "High disk usage: ${disk_usage}%"
        return 1
    else
        log_info "Disk usage normal: ${disk_usage}%"
        return 0
    fi
}

check_critical_processes() {
    # Check if important processes are running
    critical_processes=("sshd" "systemd")

    for process in "${critical_processes[@]}"; do
        if pgrep "$process" > /dev/null; then
            log_info "Process $process is running"
        else
            log_alert "Critical process $process is not running"
        fi
    done
}

generate_health_report() {
    echo "=== System Health Report ===" > $REPORT_FILE
    echo "Generated: $(date)" >> $REPORT_FILE
    echo >> $REPORT_FILE

    # System information
    echo "=== System Information ===" >> $REPORT_FILE
    echo "Hostname: $(hostname)" >> $REPORT_FILE
    echo "Uptime: $(uptime)" >> $REPORT_FILE
    echo "Load Average: $(uptime | awk -F'load average:' '{print $2}')" >> $REPORT_FIL
    echo >> $REPORT_FILE

    # Resource usage
    echo "=== Resource Usage ===" >> $REPORT_FILE
    echo "CPU Usage:" >> $REPORT_FILE
    top -bn1 | grep "Cpu(s)" >> $REPORT_FILE
    echo >> $REPORT_FILE

    echo "Memory Usage:" >> $REPORT_FILE
    free -h >> $REPORT_FILE
    echo >> $REPORT_FILE

    echo "Disk Usage:" >> $REPORT_FILE
    df -h >> $REPORT_FILE
    echo >> $REPORT_FILE

    # Network status
    echo "=== Network Status ===" >> $REPORT_FILE
    echo "Active connections:" >> $REPORT_FILE
    netstat -tuln | head -10 >> $REPORT_FILE
    echo >> $REPORT_FILE

    # Recent alerts
    echo "=== Recent Alerts ===" >> $REPORT_FILE
    if [ -f "$LOG_FILE" ]; then
        tail -20 $LOG_FILE | grep ALERT >> $REPORT_FILE
    else
        echo "No alerts found" >> $REPORT_FILE
    fi
}

main() {
    log_info "Starting system health check"

    alert_count=0

    # Run all checks
    check_cpu_usage || ((alert_count++))
    check_memory_usage || ((alert_count++))
    check_disk_usage || ((alert_count++))
    check_critical_processes

    # Generate report
    generate_health_report

    if [ $alert_count -gt 0 ]; then
        log_alert "Health check completed with $alert_count alerts"
        echo "🚨 System issues detected! Check $REPORT_FILE for details"
    else
```

```
        log_info "Health check completed - all systems normal"
        echo "✅ System health check passed"
    fi

    echo "Full report available in: $REPORT_FILE"
}

# Install bc if not available (for CPU calculations)
if ! command -v bc &> /dev/null; then
    echo "Installing bc for calculations..."
    sudo yum install bc -y 2>/dev/null || sudo apt-get install bc -y 2>/dev/null || ec
fi

main
EOF

chmod +x system_monitor.sh
./system_monitor.sh
```

🔄 Automation and Cron Jobs (20 minutes)

Introduction to Cron

What is Cron?: Cron is a time-based job scheduler in Linux. Essential for automating data engineering tasks like:

- Daily data processing
- Log rotation and cleanup
- System health checks
- Database backups

Cron Syntax:

```
* * * * * command
│ │ │ │ │
│ │ │ │ └── Day of week (0-7, 0 or 7 = Sunday)
│ │ │ └──── Month (1-12)
│ │ └────── Day of month (1-31)
│ └──────── Hour (0-23)
└────────── Minute (0-59)
```

Common Cron Patterns:

```bash
# Every minute
* * * * * /path/to/script.sh

# Every hour at minute 0
0 * * * * /path/to/script.sh

# Every day at 2:30 AM
30 2 * * * /path/to/script.sh

# Every Monday at 9:00 AM
0 9 * * 1 /path/to/script.sh

# First day of every month at midnight
0 0 1 * * /path/to/script.sh
```

Setting Up Automated Jobs:

```bash
# View current cron jobs
crontab -l

# Edit cron jobs
crontab -e

# Example: Run system health check every hour
echo "0 * * * * /home/ec2-user/system_monitor.sh" | crontab -

# Example: Run data processing every day at 3 AM
echo "0 3 * * * /home/ec2-user/data_processor.sh" | crontab -

# Example: Clean up log files weekly
echo "0 0 * * 0 find /home/ec2-user -name '*.log' -mtime +7 -delete" | crontab -
```

📈 Performance Optimization and Best Practices (15 minutes)

Efficient Text Processing

Performance Tips:

bash

```
# Use appropriate tools for the job size

# Small files (< 1MB): grep, awk, sed
grep "pattern" small_file.txt

# Medium files (1-100MB): Still grep/awk, but consider parallel processing
# Large files (> 100MB): Use more efficient tools

# For very large files, use parallel processing
split -l 10000 large_file.txt chunk_
for chunk in chunk_*; do
    grep "pattern" $chunk > results_$chunk &
done
wait
cat results_chunk_* > final_results.txt
rm chunk_* results_chunk_*

# Use built-in commands when possible (faster than external tools)
# Count lines efficiently
wc -l file.txt          # Fast
cat file.txt | wc -l    # Slower (unnecessary cat)

# Process columns efficiently
awk '{print $1}' file.txt           # Fast
cut -d' ' -f1 file.txt              # Also fast
sed 's/\s.*//' file.txt             # Slower for this task
```

Memory Management

Monitor and Optimize Memory Usage:

```bash
# Check memory usage of commands
time grep "pattern" large_file.txt

# Use streaming processing for large files
tail -f large_log_file.txt | grep "ERROR" | while read line; do
    echo "Found error: $line"
done

# Avoid loading entire files into memory
# Instead of: content=$(cat large_file.txt)
# Use: while read line; do ... done < large_file.txt
```

✅ Success Metrics and Assessment (10 minutes)

Day 7 Mastery Checklist

Command Line Navigation ✅:

- □ Navigate Linux file system confidently
- □ Create, copy, move, and delete files/directories
- □ Understand and modify file permissions
- □ Use absolute and relative paths effectively

Text Processing ✅:

- □ Search files with grep using patterns and regex
- □ Process structured data with awk
- □ Transform text with sed
- □ Combine commands with pipes

Process Management ✅:

- □ Monitor system resources (CPU, memory, disk)
- □ Manage background processes
- □ Kill unresponsive processes
- □ Understand process hierarchy

Shell Scripting ✅:

- □ Write executable shell scripts
- □ Use variables and control structures

- □ Implement error handling
- □ Create functions for reusable code

Real-World Applications ✅ :

- □ Analyze log files for troubleshooting
- □ Process CSV data with command-line tools
- □ Monitor system health automatically
- □ Set up automated tasks with cron

Knowledge Self-Assessment

Rate Your Confidence (1-10):

- Basic Linux navigation and file operations: ____/10
- Text processing with grep, awk, sed: ____/10
- Process management and system monitoring: ____/10
- Shell scripting and automation: ____/10
- Real-world data engineering applications: ____/10

Practical Challenges

Complete These Tasks:

1. Analyze a log file with 10,000+ entries to find error patterns
2. Write a script that processes multiple CSV files and generates a summary
3. Set up a cron job to monitor disk usage and send alerts
4. Create a data pipeline that downloads, processes, and uploads files
5. Monitor a long-running process and generate performance reports

---

🔗 Essential Resources for Continued Learning

Documentation and Guides

Linux Documentation:

- Linux Command Line Cheat Sheet: `linuxcommand.org`
- Advanced Bash Scripting Guide: `tldp.org/LDP/abs/html/`
- GNU Coreutils Manual: `gnu.org/software/coreutils/manual/`

Text Processing Resources:

- Awk Tutorial: `grymoire.com/Unix/Awk.html`
- Sed Tutorial: `grymoire.com/Unix/Sed.html`
- Regular Expressions Guide: `regexr.com`

Practice Environments

Online Linux Terminals:

- OverTheWire Bandit: `overthewire.org/wargames/bandit/`
- Linux Survival: `linuxsurvival.com`
- Bash Academy: `bash.academy`

Local Practice:

- VirtualBox with Ubuntu/CentOS
- Docker containers with Linux
- Windows WSL2 for Windows users

🚀 Tomorrow's Preview: Day 8 - Git and Version Control

What You'll Learn Tomorrow

Core Focus: Professional code collaboration and version management

- Git fundamentals and workflows
- GitHub/GitLab collaboration patterns
- Branching strategies for data projects
- Code review processes for data engineering

Why Git Matters for Data Engineers:

- Track changes in data pipelines and SQL scripts
- Collaborate with team members on data projects
- Maintain different versions of ETL processes
- Roll back problematic deployments quickly
- Document data transformation logic

Tomorrow's Preparation

Tools to Install:

- Git (command line tool)
- GitHub account setup
- Visual Studio Code (optional, for better Git integration)
- Your Day 7 scripts (we'll version control them)

Concepts to Review:

- The difference between local and remote repositories
- Why version control is essential for data projects
- How teams collaborate on code

🎉 Congratulations on Mastering Linux!

What You've Accomplished Today

You've gained the fundamental Linux skills that every data engineer uses daily. You now understand:

Technical Skills:

- Command-line navigation and file management
- Advanced text processing for data analysis
- Process monitoring and system administration
- Shell scripting for automation
- Real-world troubleshooting techniques

Data Engineering Applications:

- Log analysis for pipeline monitoring
- Automated data processing workflows
- System health monitoring for infrastructure
- File processing without memory limitations
- Remote server management via SSH

Production-Ready Patterns:

- Error handling in scripts
- Performance optimization techniques

- Security best practices for file permissions
- Automation with cron jobs
- Monitoring and alerting systems

Your Linux Foundation is Solid

You're now equipped with the same command-line skills used by data engineers at major technology companies. These skills form the foundation for managing cloud infrastructure, processing big data, and automating data pipelines.

Progress: 14% (7/50 days) | **Next**: Day 8 - Git and Version Control **Skills Mastered**: Python ✅ + SQL ✅ + Advanced SQL ✅ + Cloud Fundamentals ✅ + Linux CLI ✅

Learning Journal Template

```markdown
# Day 7: Linux Command Line - Learning Notes

## Command Line Skills Mastered
- File system navigation and management
- Text processing with grep, awk, sed
- Process monitoring and management
- Shell scripting and automation

## Real-World Applications
- Production log analysis techniques
- Automated data processing pipelines
- System health monitoring
- Remote server management

## Key Insights
- Command line is faster than GUI for repetitive tasks
- Text processing tools are incredibly powerful for data work
- Automation saves hours of manual work
- Linux skills are essential for cloud data engineering

## Tomorrow's Goals
- Learn Git for version control
- Understand collaborative development workflows
- Apply version control to data engineering projects
```

This comprehensive guide provides everything needed to master Linux fundamentals for data engineering. You now have the command-line skills to manage servers, process data, and automate workflows like a professional data engineer!