**Class 1: Linux Fundamentals and System Architecture**

**Opening Hook & Problem Statement**

**The DevOps Engineer's Dilemma**

*"Imagine you're Netflix's newest DevOps engineer. It's 9 PM on a Friday, and suddenly millions of users can't stream their favorite shows. Your phone is buzzing with alerts, your manager is calling, and social media is exploding with complaints. You have seconds to make decisions that could affect millions of users and millions in revenue.*

*You need to:*

* *Quickly identify which of the 1000+ servers are failing*
* *Understand what processes are consuming resources*
* *Navigate through complex directory structures to find logs*
* *Execute commands that could either fix the issue or make it worse*

*But here's the catch - you can only do this through a black terminal window with white text. No fancy GUI, no drag-and-drop. Just you, a command line, and your understanding of how computer systems really work."*

**Question to Students:** *"How many of you have felt intimidated by that black terminal window? Raise your hands!"*

**The Reality Check:** *"By the end of this module, that intimidating black screen will become your superpower. You'll understand computers at a level that puts you in the top 5% of developers. But first, we need to understand: Why does any of this matter?"*

**Module Introduction: Why This Matters**

**The DevOps Reality**

**Analogy: The Complete Tech Stack** *"Think of software development like building a modern smart city. Frontend developers are the urban planners who design beautiful, user-friendly spaces. Backend developers are the engineers who build robust infrastructure and services. DevOps engineers? You're the city operations specialists who ensure everything runs smoothly 24/7 - from traffic flow (networking) to power distribution (servers) to emergency response (monitoring and incident management). Each role is crucial, but DevOps gives you the unique ability to see and manage the entire ecosystem."*

**What You'll Become**

**By the end of this module, you'll be able to:**

1. **Diagnose system issues** faster than senior engineers with 10+ years of experience
2. **Automate repetitive tasks** that take others hours to do manually
3. **Understand what happens behind the scenes** when a production system serves millions of users - from how the OS manages memory and processes, to how networks route packets across continents, to how databases handle concurrent transactions
4. **Master advanced system concepts** like kernel internals, network protocols, database replication, and performance optimization that separate good engineers from exceptional ones
5. **Think in systems** - see the interconnections between operating systems, networks, and databases that most developers treat as black boxes

**Module Structure Overview**

*"Before we dive deep, let me give you the roadmap of our 24-class journey. This module is carefully designed to build your systems expertise layer by layer:"*

**Our Learning Path:**

**1. Linux Shell Scripting, Operating Systems and System Administration (Classes 1-8)**

* Master the command line and understand how operating systems really work
* Learn system administration tasks that keep production systems running
* Build automation scripts that save hours of manual work

**2. Computer Networks and Troubleshooting (Classes 9-14)**

* Understand how data travels across the internet
* Master network debugging tools and techniques
* Learn to diagnose and fix connectivity issues

**3. Databases, DB Scaling and Troubleshooting (Classes 15-18)**

* Deep dive into database internals and performance optimization
* Understand replication, scaling, and high availability
* Master database troubleshooting and monitoring

**4. Python Scripting for Systems (Classes 19-22)**

* Build powerful automation scripts for DevOps tasks
* Create monitoring and alerting systems
* Develop tools for log analysis and system management

**5. Advanced Troubleshooting Case Studies (Classes 23-24)**

* Apply everything you've learned to complex real-world scenarios
* Practice incident response and root cause analysis
* Prepare for production environment challenges

*"Each section builds on the previous one. By the end, you'll have the complete toolkit of a senior DevOps engineer!"*

**How Computer Systems Really Work**

**The Foundation: Hardware and Software Dance**

*"Let's start with something you do every day - double-clicking the Google Chrome icon. It seems simple, but behind that innocent click lies one of the most complex orchestrations in computer science. What really happens in those 2-3 seconds before you see the Chrome window?"*

**The Two-Layer Reality**

**Hardware Layer: The Physical World**

* **CPU:** The brain that executes instructions (billions per second)
* **RAM:** Fast temporary storage for active programs and data
* **Storage:** Permanent data storage (SSDs, HDDs)
* **Network Interface:** Communication gateway to other computers
* **Peripherals:** Input/output devices (keyboard, screen, etc.)

**Software Layer: The Instructions**

* **Applications:** Your Python code, web browsers, databases
* **System Software:** Programs that manage other programs
* **Firmware:** Low-level code embedded in hardware

**The Problem:** *"Hardware only understands electrical signals and binary. Your Python code is text. How do they communicate?"*

**Enter the Operating System: The Universal Translator**

**OS as the Middleman Analogy**

*"Think of the OS like a hotel concierge. Guests (applications) have needs: 'I need a room' (memory), 'I want to make a call' (network), 'I need to store my luggage' (file storage). The concierge (OS) manages all hotel resources and fulfills these requests without guests needing to know how the hotel works internally."*

**What the OS Actually Does**

**For Applications:**

* **"I need memory"** → OS allocates RAM and tracks usage
* **"I want to save a file"** → OS manages disk storage and file organization
* **"I need to communicate over network"** → OS handles network protocols
* **"I want to start another program"** → OS creates and manages processes

**For Hardware:**

* Manages CPU time between competing applications
* Controls memory allocation and prevents conflicts
* Handles hardware interrupts and device drivers
* Ensures system security and stability

**Real-World Example:** *"When you double-click Chrome, here's what really happens:*

1. *Your mouse click (hardware) generates an interrupt signal*
2. *OS detects the click and identifies you want to run Chrome*
3. *OS locates Chrome executable file on your disk*
4. *OS loads Chrome from storage into RAM (memory allocation)*
5. *OS creates a new process with unique Process ID for Chrome*
6. *OS allocates CPU time for Chrome to initialize*
7. *Chrome requests network access to load your homepage*
8. *OS manages the display to show Chrome's window*
9. *Chrome continues making requests (more memory, network calls, file access)*
10. *OS coordinates everything while protecting system security"*

**Why Linux? The Restaurant Chain Analysis**

Windows Restaurant:

* Expensive franchise fees (licensing costs)
* Fixed menu, can't customize much
* Corporate decides everything
* Great for family dining (desktop users)

macOS Restaurant:

* Premium experience, beautiful ambiance
* Limited locations, expensive
* Very controlled environment
* Perfect for upscale customers

Linux Restaurant:

* Free to open anywhere (open source)
* Completely customizable menu (configure everything)
* Chain of independent restaurants sharing recipes (community)
* Preferred by professional chefs (developers/DevOps)

The Numbers That Matter

Student Activity: *"Quick survey - how many applications do you think are running on your phone right now?"*

The Reality:

* Your smartphone: 50-200+ processes
* A typical web server: 500-2000+ processes
* Netflix's infrastructure: Millions of processes across thousands of servers

Why This Matters: Every Netflix stream, every Google search, every WhatsApp message travels through Linux systems. Understanding Linux means understanding the backbone of the internet.

**Linux OS Deep Dive: Kernel, User Space and System Calls**

**The Kernel: The Core Manager**

*"The Linux kernel is like the engine of a car. You interact with the steering wheel and pedals (applications), but the engine (kernel) does all the actual work."*

**Kernel Responsibilities:**

* **Process Management:** Creating, scheduling, and terminating programs
* **Memory Management:** Allocating RAM, handling virtual memory, managing swap
* **File System Management:** Organizing data storage, handling permissions
* **Device Management:** Communicating with hardware through drivers
* **Network Management:** Handling network protocols and connections
* **Security Management:** Enforcing permissions and access controls

**Why Kernel is Special:**

* Runs in **privileged mode** (can access hardware directly)
* Always in memory (never gets swapped out)
* Has complete control over system resources
* Applications **cannot** directly access kernel functions

**User Space vs Kernel Space: The Security Boundary**

**User Space (Applications):**

* Your Python programs, web browsers, text editors
* **Restricted access** - cannot directly control hardware
* If they crash, they don't crash the whole system
* Must ask kernel for everything through system calls

**Kernel Space (OS Core):**

* Device drivers, memory manager, process scheduler
* **Full hardware access** and control
* If kernel crashes, entire system crashes
* Directly manipulates CPU, memory, and devices

**System Calls: The Application-Kernel Interface**

**The API Between Worlds**

*"System calls are like ordering food at a restaurant. You (application) tell the waiter (system call) what you want. The waiter goes to the kitchen (kernel), gets your food (performs the operation), and brings it back to you."*

**Common System Calls Every DevOps Engineer Should Know:**

**File Operations:**

* **open():** "Give me access to this file"
* **read():** "Bring me data from this file"
* **write():** "Save this data to the file"
* **close():** "I'm done with this file"

**Process Operations:**

* **fork():** "Create an identical copy of my process"
* **exec():** "Replace my process with a different program"
* **wait():** "Wait for my child process to complete"
* **exit():** "I'm done, clean up my resources"

**Memory Operations:**

* **malloc():** "Give me some memory"
* **mmap():** "Map a file into memory"
* **brk():** "Expand my memory space"

**Live Demonstration: Seeing System Calls in Action**

# Let's see what system calls happen for a simple command

|  |
| --- |
| strace echo "Hello DevOps" 2>&1 | head -20 |

**Mind-Blowing Moment:** *"Look! Even printing 'Hello DevOps' required 20+ system calls - opening libraries, allocating memory, writing to terminal, cleaning up resources!"*

**Why This Matters for DevOps**

**Performance Debugging:**

* Understanding system calls helps identify bottlenecks
* Too many file operations? Memory leaks? Network issues?

**Security Understanding:**

* System calls are attack vectors - understanding them improves security
* File permissions, process isolation, network access controls

**Application Troubleshooting:**

* When applications fail, system call traces show exactly what went wrong
* "Permission denied" vs "File not found" vs "Out of memory"

**Example: Real DevOps Scenario** *"Your web application suddenly becomes slow. Using strace, you discover it's making thousands of unnecessary file system calls. Root cause: inefficient database connection handling. Fix: implement connection pooling. Result: 10x performance improvement."*

**Shell Environments: Your Command Center**

**The Command Line vs GUI: Why Professionals Choose Text**

**The Great Interface Debate**

*"Imagine you're managing 100 servers simultaneously. With a GUI, you'd need to remote desktop into each server, click through menus, and manually perform tasks. That could take hours or days. With a shell, you can execute commands across all 100 servers in seconds. That's the power of the command line."*

**GUI vs Shell: The Real Comparison**

**Graphical User Interface (GUI) Strengths:**

* **Visual and intuitive** - Easy to learn for beginners
* **Discovery-friendly** - You can explore by clicking around
* **Good for one-off tasks** - Opening a file, viewing an image
* **Multitasking visual cues** - Windows, icons, visual feedback

**Shell Interface Strengths:**

* **Speed and efficiency** - Expert users are 10x faster
* **Automation capability** - Scripts can repeat tasks thousands of times
* **Remote accessibility** - Works over slow network connections
* **Precision and control** - Exact commands with specific parameters
* **Composability** - Combine simple commands to create powerful workflows
* **Resource efficiency** - Minimal memory and CPU usage
* **Scriptable and repeatable** - Document and share exact procedures

**Examples from a typical DevOps Engineer's Day:**

* Deploy application to 50 servers → Shell: 30 seconds, GUI: 2 hours
* Find all log files containing errors → Shell: 5 seconds, GUI: manually impossible
* Monitor system resources across cluster → Shell: real-time dashboard, GUI: not scalable
* Backup databases with compression → Shell: one command, GUI: multiple steps per database

**How the Shell Actually Works: Under the Hood**

**The Shell's Role in the System**

*"The shell sits between you and the kernel, like a translator who speaks both human language and computer language."*

┌─────────────────────────────────────────┐

│           Your Commands                 │

│        "ls -la /home"                  │

└─────────────────┬───────────────────────┘

                  │

┌─────────────────▼───────────────────────┐

│              SHELL                      │

│   • Parses your command                 │

│   • Finds the program (ls)             │

│   • Sets up environment                 │

│   • Handles input/output                │

└─────────────────┬───────────────────────┘

                  │

┌─────────────────▼───────────────────────┐

│           KERNEL                        │

│   • Executes the program               │

│   • Manages system resources           │

│   • Returns results                    │

└─────────────────┬───────────────────────┘

                  │

┌─────────────────▼───────────────────────┐

│           OUTPUT                        │

│    File listings displayed              │

└─────────────────────────────────────────┘

**Shell Processing: Step by Step (We will learn each of below in detailed depth in upcoming classes)**

**When you type: ls -la /home | grep user | wc -l**

**Step 1: Command Parsing**

* Shell breaks down the command into tokens: ls, -la, /home, |, grep, user, |, wc, -l
* Identifies three separate commands connected by pipes
* Recognizes flags (-la, -l) and arguments (/home, user)

**Step 2: Program Location**

* Shell searches $PATH directories to find executables
* /bin/ls, /bin/grep, /usr/bin/wc - found!
* If command not found → "command not found" error

**Step 3: Process Creation and Execution**

* Shell forks itself to create child processes
* Uses execve() system call to run each program
* Sets up pipes to connect output of ls to input of grep, etc.

**Step 4: Environment Setup**

* Passes environment variables to child processes
* Sets up standard input, output, and error streams
* Manages working directory and file descriptors

**Step 5: Process Management**

* Shell waits for processes to complete (foreground)
* Or manages background processes (with &)
* Collects exit codes and handles errors

**Step 6: Output Display**

* Receives results from final command in pipeline
* Displays output to your terminal
* Shows new prompt when ready for next command

**Evolution of Shells: From Basic to Powerful**

**The Shell Family Tree**

**1. Thompson Shell (sh) - 1971** *"The grandfather of all shells - basic but revolutionary"*

* First command-line interpreter for UNIX
* Simple command execution, basic I/O redirection
* No scripting capabilities, no history

**2. Bourne Again Shell (bash) - 1989** *"The people's champion - combines the best of all worlds"*

* **Why it became dominant:**
  + Added interactive features
  + Extensive scripting capabilities
  + Tab completion and command history
  + Powerful pattern matching and expansion

**3. Z Shell (zsh) - 1990** *"The modern developer's choice"*

* **Advanced features:**
  + Superior autocompletion (context-aware)
  + Spelling correction and suggestions
  + Plugin ecosystem (Oh My Zsh)
  + Advanced globbing and pattern matching
  + Theme support and customization

**Interactive Demo: Shell Superpowers**

**Power Feature 1: Command Composition**

**Challenge to Students:** *"I'll show you something that would require a custom application to do in GUI, but just one line in shell."*

bash

*# Find all Python files, count lines of code, sort by size*

find . -name "\*.py" -exec wc -l {} + | sort -n | tail -10

*# Translation: "Find all .py files, count lines in each, sort by line count, show top 10"*

**Mind = Blown Moment:** *"This one command just analyzed your entire codebase in seconds!"*

**Power Feature 2: Command History and Shortcuts**

bash

*# Navigate command history*

history | tail -10          *# See last 10 commands*

!!                          *# Repeat last command*

!grep                       *# Repeat last command starting with 'grep'*

!?python?                   *# Repeat last command containing 'python'*

^old^new                    *# Replace 'old' with 'new' in last command*

*# Quick navigation*

cd -                        *# Go back to previous directory*

pushd /tmp && popd          *# Visit directory and return*

**Power Feature 3: Tab Completion Intelligence**

bash

*# Type and press TAB to see the magic*

cd /ho<TAB>                 *# Completes to /home/*

ls -l /etc/pa<TAB>          *# Shows all files starting with 'pa'*

git che<TAB>                *# Completes git commands*

kill -9 <TAB>               *# Shows running process IDs*

**Power Feature 4: Variables and Environment**

bash

*# Environment variables - your shell's memory*

echo $HOME                  *# Your home directory*

echo $PATH                  *# Where shell looks for commands*

echo $USER                  *# Your username*

echo $PWD                   *# Current directory*

*# Create your own variables*

export API\_KEY="secret123"

export DATABASE\_URL="postgresql://localhost/myapp"

echo "Connecting to $DATABASE\_URL"

*# Use in scripts and commands*

cp important.txt $HOME/backups/

curl -H "Authorization: Bearer $API\_KEY" https://api.example.com/data

**Shell Customization: Making it Yours**

**The Power of Aliases**

*# DevOps-specific aliases*

alias k='kubectl'

alias d='docker'

alias g='git'

alias tf='terraform'

**Why Mastering Shell Makes You a Better DevOps Engineer**

**Real-World Scenarios**

**Incident Response:**

bash

*# Quick system health check (30 seconds vs 10 minutes GUI clicking)*

uptime && free -h && df -h && ps aux --sort=-%cpu | head -5

**Log Investigation:**

bash

*# Find all 500 errors in the last hour*

find /var/log -name "\*.log" -mtime -1 -exec grep -l "HTTP 500" {} \;

**Batch Operations:**

bash

*# Update configuration on 50 servers*

for server in $(cat servers.txt); do

    ssh $server "sudo systemctl restart nginx"

done

**Performance Monitoring:**

bash

*# Real-time resource monitoring*

watch -n 1 'ps aux --sort=-%cpu | head -10'

**Real-World Application & Hands-On**

**The DevOps Scenario: New Server Setup**

*"You've just been given access to a brand new Linux server. Your first day task: set up the environment for a Python web application."*

**Guided Hands-On:**

**Step 1: Understand Your Environment**

bash

*# Who am I?*

whoami

id

*# Where am I?*

pwd

ls -la

*# What kind of system is this?*

uname -a

cat /etc/os-release

**Step 2: Explore the System Structure**

bash

*# The root of everything*

ls /

*# Let's understand what each directory does*

echo "This is where programs live:" && ls /bin | head -5

echo "This is where configuration files live:" && ls /etc | head -5

echo "This is where user data lives:" && ls /home

**Step 3: Check System Resources**

bash

*# CPU information*

lscpu | head -10

*# Memory information*

free -h

*# Disk space*

df -h

Day 2

**Class 2: File System Operations and Navigation**

**Class Agenda: Your Journey to File System Mastery**

**What You'll Learn Today (And Why It's Exciting!)**

"Before we dive into our scenario, let me show you the incredible journey we're about to take together. By the end of these 2 hours, you'll go from being confused by the terminal to having superpowers that most people don't even know exist."

**🎯 Our Learning Adventure:**

**Phase 1: The Mind-Blowing Discovery (15 minutes)**

* Discover Linux's secret weapon: "Everything is a File"
* See how this one principle gives you control over your entire system
* Watch hardware, processes, and network connections become manageable through simple file operations

**Phase 2: Becoming a Navigation Expert (30 minutes)**

* Master the essential navigation commands that make you faster than any GUI user
* Learn to find any file on any system in seconds
* Understand the difference between absolute and relative paths like a pro

**Phase 3: File Operations Mastery (25 minutes)**

* Create, copy, move, and organize files with precision
* Learn the safe way to delete files (and avoid disasters!)
* Discover advanced techniques that save hours of manual work

**Phase 4: Unlocking Security Secrets (35 minutes)**

* Decode those mysterious permission symbols once and for all
* Set up proper file security for team collaboration
* Learn advanced permission techniques that separate beginners from experts

**Phase 5: Advanced File Magic (10 minutes)**

* Master symbolic and hard links for efficient file organization
* Learn techniques used by system administrators worldwide
* Set up shortcuts and connections that make your workflow lightning-fast

**Phase 6: Real-World Application (25 minutes)**

* Apply everything in a hands-on project that simulates actual DevOps work
* Build a complete project structure with proper permissions and organization
* Walk away with a practical skill set you can use immediately

**Why Stay Until the End?** "Each phase builds on the previous one, and the final hands-on project ties everything together. Students who've taken this class before say the 'aha moments' keep coming throughout the entire session - and the biggest one comes at the very end when everything clicks together!"

**Opening Hook & Problem Statement**

**The Simple Task That Becomes a Mystery**

"Your team lead walks over to your desk and says: 'Hey, can you help me with something quick? I need you to find that Python script we were working on yesterday - I think it's called data\_processor.py. Make a backup copy of it, and then move it to the shared project folder so everyone can access it.'

You think: 'Easy! I'll just search for it on my computer.'

But then you realize you're working on a Linux server through the terminal. No search bar. No file explorer with pretty folder icons. Just this:

student@server:~$

You need to:

* Find a file somewhere on this computer (but where?)
* Make a copy of it (but how?)
* Move it to the right folder (but which folder?)
* Make sure other people can access it (but what does that even mean?)

You start typing random commands you half-remember:

student@server:~$ find data\_processor.py

find: missing argument to `-exec'

student@server:~$ ls

Desktop Documents Downloads

student@server:~$ cd Documents

student@server:~/Documents$ ls

student@server:~/Documents$

Empty folder. The file could be anywhere. You're stuck."

**Question to Students:** "How many of you have been in this exact situation - knowing what you want to do, but having no idea how to do it in the terminal?"

**What We'll Learn Today**

"Today, we're going to turn you into someone who can handle this situation in under 2 minutes. You'll learn how to navigate, find, organize, and manage files through the command line - and actually prefer it to clicking around in folders.

By the end of class, when someone asks you to find and organize files, you'll think: 'No problem, I got this.'"

**Module Connection: Building on Class 1**

**Quick Recap: The Foundation We Built**

"Last class, we explored how computer systems work - the hardware layer, the operating system as the middleman, and how applications communicate through system calls. We learned about the shell as our command interface.

Today, we're going to discover one of Linux's most powerful design principles and learn to navigate the system like a pro."

**Core Concept 1: Everything is a File in Linux - The Revolutionary Design**

**Discovering Linux's Secret Superpower**

**The Big Reveal: What Makes Linux Special**

"Before we learn to navigate files, we need to understand something that will blow your mind about how Linux works. It's called the 'Everything is a File' philosophy, and it's the reason Linux powers most of the internet."

**The Universal File Philosophy: A Mind-Bending Concept**

**The Library Analogy**

"Imagine walking into the world's most organized library. In a normal library, you have books, magazines, computers, printers, card catalogs, and information desks - all different things that work differently.

But in this magical Linux library, everything is treated as a 'book' that you can read from or write to:

* Regular books (your documents and files)
* Interactive books that respond when you write in them (hardware devices)
* Live books that update in real-time (system information)
* Reference books that tell you what's happening right now (running processes)
* Communication books that let you talk to other libraries (network connections)

The revolutionary part? You use the same basic operations to interact with everything: open, read, write, close."

**What This Actually Means in Linux**

**Regular Files:** Your documents, scripts, images, videos

bash

*# A Python script - just a regular file*

-rw-r--r-- 1 user user 1234 Nov 24 10:00 app.py

**Hardware Devices:** Physical components treated as files

bash

*# Your hard disk - it's a file!*

brw-rw---- 1 root disk 8, 0 Nov 24 10:00 /dev/sda

*# Your mouse - also a file!*

crw-rw-rw- 1 root input 13, 32 Nov 24 10:00 /dev/input/mouse0

**System Information:** Live data about your computer

bash

*# CPU information - it's a file you can read!*

-r--r--r-- 1 root root 0 Nov 24 10:00 /proc/cpuinfo

*# Memory usage - also a file!*

-r--r--r-- 1 root root 0 Nov 24 10:00 /proc/meminfo

**Running Processes:** Each running program has its own "folder"

bash

*# Information about process ID 1234*

dr-xr-xr-x 9 user user 0 Nov 24 10:00 /proc/1234/

**Network Connections:** Network data as readable files

bash

*# All your network connections*

-r--r--r-- 1 root root 0 Nov 24 10:00 /proc/net/tcp

**The "Everything is a File" Advantages: Why This is Genius**

**Advantage 1: Universal Tools Work Everywhere**

"Because everything is a file, the same commands work on everything:"

bash

*# Read a regular text file*

cat my\_document.txt

*# Read CPU information (it's also a file!)*

cat /proc/cpuinfo

*# Read memory information (also a file!)*

cat /proc/meminfo

*# Read network connections (yes, also a file!)*

cat /proc/net/tcp

*# Even read random data from hardware*

cat /dev/urandom | head -c 10

**Mind-Blowing Moment:** "The same cat command that reads your Python script can also tell you about your CPU, memory, and network connections!"

**Advantage 2: Pipe Any Data Anywhere**

"Since everything is a file, you can connect any data source to any data destination:"

bash

*# Monitor CPU usage in real-time*

watch -n 1 'cat /proc/loadavg'

*# Save system information to a file*

cat /proc/cpuinfo > system\_info.txt

*# Search through running processes*

cat /proc/\*/comm | grep python

*# Monitor network connections*

watch -n 1 'cat /proc/net/tcp | wc -l'

**Advantage 3: Simple and Consistent Interface**

"No matter what you're working with, the interface is always the same:"

bash

*# Want to know about disk usage? Read a file*

cat /proc/diskstats

*# Want to control hardware? Write to a file*

echo "1" > /sys/class/leds/input0::capslock/brightness *# Turn on Caps Lock LED*

*# Want to see what a process is doing? Read its files*

cat /proc/1234/cmdline *# Command that started the process*

cat /proc/1234/status *# Process status information*

**Advantage 4: Scriptable Everything**

"Since everything is a file, you can automate everything with simple file operations:"

bash

*# Create a system monitoring script*

*#!/bin/bash*

echo "=== System Health Check ==="

echo "CPU Load: $(cat /proc/loadavg)"

echo "Memory Free: $(grep MemFree /proc/meminfo)"

echo "Disk Usage: $(df -h / | tail -1)"

echo "Active Connections: $(cat /proc/net/tcp | wc -l)"

**Interactive Demonstration: Seeing "Everything is a File" in Action**

**Live Demo: Exploring Different File Types**

bash

echo "=== Let's explore different types of 'files' ==="

echo "1. Regular file:"

ls -la README.txt

cat README.txt

echo "2. Device file (your terminal):"

ls -la /dev/tty

echo "Hello from device file!" > /dev/tty

echo "3. System information file:"

ls -la /proc/cpuinfo

head -5 /proc/cpuinfo

echo "4. Process information file:"

ls -la /proc/self/

cat /proc/self/comm *# Shows current command name*

echo "5. Network information file:"

ls -la /proc/net/tcp

head -3 /proc/net/tcp

Day 3

**Class 3: Process Management and System Monitoring**

**Opening Hook & Problem Statement**

**The Production Crisis Scenario**

"It's 3 AM. You're the on-call DevOps engineer for a major e-commerce platform. Your phone explodes with alerts: 'CRITICAL: Server CPU at 100%', 'Website response time > 30 seconds', 'Customer complaints flooding in'. You SSH into the server, and your screen fills with processes you don't recognize.

Questions racing through your mind:

* Which process is eating all the CPU?
* Is it legitimate traffic or a runaway process?
* How do I kill it without crashing the entire system?
* What if I accidentally kill the database process?
* How do I prevent this from happening again?

But here's the terrifying part: You have 5 minutes before your CEO calls asking why the company is losing $10,000 per minute in sales."

**Question to Students:** "How many of you would know exactly what commands to run to diagnose and fix this in under 5 minutes?"

**Why This Lecture Changes Everything**

"The skills you're about to learn are the difference between a developer and a DevOps engineer. These aren't just Linux commands - they're the foundation of:

* **Container Orchestration**: Every Docker container is just processes. Kubernetes manages millions of them using these exact concepts
* **Cloud Infrastructure**: AWS EC2, Google Compute, Azure VMs - all running Linux processes that you'll need to monitor and manage
* **Monitoring Systems**: Datadog, New Relic, Prometheus - they're all built on top of the process monitoring concepts we'll cover today
* **Incident Response**: When production fails, these commands are your first line of defense. Companies lose millions when engineers can't quickly identify and fix process issues
* **CI/CD Pipelines**: Jenkins, GitLab CI, GitHub Actions - all manage build processes using these fundamentals
* **Database Administration**: Understanding why your database is slow often comes down to process and resource management

Without mastering process management, you're flying blind in production. With it, you become the engineer everyone calls when things go wrong."

**The Reality Check**

"By the end of today's class, you'll be able to:

1. Identify any problematic process in seconds
2. Understand exactly what's happening on any Linux system
3. Control processes like a conductor controls an orchestra
4. Handle production emergencies with confidence
5. Prevent zombie apocalypses (yes, that's a real thing in Linux!)"

**Part 1: Understanding Processes - The Foundation**

**What is a Process? The Living Programs**

**The Restaurant Kitchen Analogy**

"Imagine a busy restaurant kitchen. The recipes are like your programs (code on disk). But a recipe sitting in a cookbook doesn't feed anyone. When a chef starts cooking from that recipe, it becomes an active dish being prepared - that's a process!

* **Program (Recipe)**: Static instructions stored on disk
* **Process (Cooking)**: Active execution of those instructions
* **Multiple Processes**: Multiple chefs can cook the same recipe simultaneously (multiple instances of Chrome)
* **Resources**: Each chef needs space, ingredients, tools (CPU, memory, file handles)"

**Process vs Program: The Critical Distinction**

bash

*# Let's see this in action*

ls -la /usr/bin/python3 *# This is a program (file on disk)*

python3 & *# Now it's a process (running in memory)*

python3 & *# Another process from the same program!*

ps aux | grep python *# See both processes with different PIDs*

**Mind-Blowing Fact:** "When you open 10 Chrome tabs, you're not running 10 different programs - you're running 10+ processes from ONE program!"

**The Process Lifecycle: Birth to Death**

**The Human Life Cycle Analogy**

"A process lifecycle is remarkably similar to human life:

* **Birth (Creation)**: Parent process creates a child
* **Youth (Running)**: Actively doing work
* **Sleep**: Waiting for something (like waiting for user input)
* **Death**: Process completes or is terminated
* **Zombie**: Dead but not buried (parent hasn't acknowledged death)
* **Orphan**: Parent died, adopted by init process (PID 1)"

**Process States: The Complete Picture**

NEW ──────► READY ◄────────┐

│ │ │

│ ▼ │

│ RUNNING ──────────┤

│ │ │

│ ▼ │

└───► TERMINATED WAITING

│ │

▼ │

ZOMBIE ◄──────────┘

**State 1: Running (R) - The Active Worker**

"Process is actively using CPU right now. Like a cashier actively scanning items."

bash

*# Create a running process*

python3 -c "while True: pass" &

ps aux | grep python *# See the 'R' state*

**State 2: Sleeping (S) - The Patient Waiter**

"Process is waiting for something - user input, network response, disk I/O. Like a waiter waiting for the kitchen to prepare food."

**Two Types of Sleep:**

* **Interruptible Sleep (S)**: Can be woken by signals
* **Uninterruptible Sleep (D)**: Cannot be interrupted (usually waiting for I/O)

bash

*# Interruptible sleep example*

sleep 100 &

ps aux | grep sleep *# See the 'S' state*

*# Uninterruptible sleep (happens during heavy I/O)*

dd if=/dev/zero of=/tmp/bigfile bs=1M count=1000 &

ps aux | grep dd *# Might catch 'D' state*

**State 3: Stopped (T) - The Paused Worker**

"Process is paused, frozen in time. Like pressing pause on a video."

bash

*# Start a process and stop it*

ping google.com &

*# Press Ctrl+Z to stop it*

ps aux | grep ping *# See the 'T' state*

**State 4: Zombie (Z) - The Walking Dead**

**The Zombie Apocalypse Explained**

"A zombie process is dead but still listed in the process table. It's like a student who finished an exam but the teacher hasn't collected the paper yet."

bash

*# Create a zombie (demonstration)*

python3 -c "

import os

import time

pid = os.fork()

if pid > 0:

# Parent sleeps, doesn't collect child

time.sleep(60)

else:

# Child exits immediately

exit()

" &

ps aux | grep defunct *# See the zombie!*

**Why Zombies Exist:**

* Child process dies
* Parent hasn't called wait() to collect exit status
* Process table entry remains (taking up PID)
* Can't be killed (already dead!)

**The Zombie Problem:**

bash

*# Check for zombies on your system*

ps aux | awk '$8 ~ /^Z/ { print }'

*# Count zombies*

ps aux | awk '$8 ~ /^Z/ { print }' | wc -l

**State 5: Orphan - The Adopted Process**

"When a parent process dies before its children, those children become orphans. Linux automatically adopts them to init (PID 1) - the grandmother of all processes!"

bash

*# Create an orphan*

python3 -c "

import os

import time

pid = os.fork()

if pid > 0:

# Parent dies immediately

exit()

else:

# Child continues running

time.sleep(30)

print('I am an orphan!')

" &

*# Watch the orphan get adopted*

ps -ef | grep python *# Notice PPID becomes 1*

**Part 2: Process Monitoring Tools - Your X-Ray Vision**

**The ps Command: Process Snapshot**

**The Photography Analogy** "ps is like taking a photograph of all processes at this exact moment. It's a snapshot, not a video."

**Basic ps Usage**

bash

*# Simple process list*

ps

*# All processes with full details*

ps aux

*# Process tree view*

ps auxf

*# Custom columns*

ps -eo pid,ppid,user,cmd,%cpu,%mem --sort=-%cpu | head -10

**Understanding ps aux Output**

bash

USER PID %CPU %MEM VSZ RSS TTY STAT START TIME COMMAND

root 1 0.0 0.4 225584 9432 ? Ss 08:30 0:01 /sbin/init

**Column Breakdown:**

* **USER**: Process owner
* **PID**: Process ID (unique identifier)
* **%CPU**: CPU usage percentage
* **%MEM**: Memory usage percentage
* **VSZ**: Virtual memory size
* **RSS**: Resident Set Size (actual RAM usage)
* **TTY**: Terminal associated with process
* **STAT**: Process state (R/S/D/T/Z)
* **START**: When process started
* **TIME**: Total CPU time consumed
* **COMMAND**: The actual command

**Advanced ps Tricks**

bash

*# Find memory hogs*

ps aux --sort=-%mem | head -5

*# Find CPU hogs*

ps aux --sort=-%cpu | head -5

*# Processes of specific user*

ps -u username

*# Process with specific PID*

ps -p 1234

*# All Python processes*

ps aux | grep python

*# Process tree for specific process*

ps -auxf | grep -A5 -B5 process\_name

**The top Command: Real-Time Monitoring**

**The Security Camera Analogy** "If ps is a photograph, top is a live security camera feed. You see everything happening in real-time."

**Starting top and Understanding the Display**

bash

top

**Header Information Explained:**

top - 10:23:45 up 5 days, 3:15, 2 users, load average: 1.52, 1.38, 1.25

Tasks: 287 total, 2 running, 285 sleeping, 0 stopped, 0 zombie

%Cpu(s): 15.3 us, 5.2 sy, 0.0 ni, 78.5 id, 0.8 wa, 0.0 hi, 0.2 si

MiB Mem: 16384.0 total, 1234.5 free, 8765.4 used, 6384.1 buff/cache

MiB Swap: 8192.0 total, 8192.0 free, 0.0 used. 7234.5 avail Mem

**Line by Line Breakdown:**

1. **Uptime Line**: Current time, system uptime, users logged in, load average (1, 5, 15 min)
2. **Tasks**: Total processes and their states
3. **CPU**: us(user), sy(system), ni(nice), id(idle), wa(wait), hi(hardware interrupts), si(software interrupts)
4. **Memory**: Total, free, used, buffers/cache
5. **Swap**: Virtual memory statistics

**top Interactive Commands - The Power User's Toolkit**

bash

*# While top is running:*

h *# Help*

k *# Kill a process (enter PID)*

r *# Renice (change priority)*

u *# Filter by user*

M *# Sort by memory usage*

P *# Sort by CPU usage (default)*

c *# Show full command path*

1 *# Show individual CPU cores*

f *# Select fields to display*

q *# Quit*

**Real-World top Usage Scenarios**

bash

*# Monitor specific user's processes*

top -u username

*# Batch mode for scripts*

top -b -n 1 > system\_snapshot.txt

*# Monitor specific PIDs*

top -p 1234,5678

*# Update every 5 seconds*

top -d 5

**htop: top on Steroids**

"htop is like top after going to the gym - stronger, prettier, and more user-friendly!"

bash

*# Install if not available*

sudo apt install htop *# Debian/Ubuntu*

sudo yum install htop *# RedHat/CentOS*

htop

**htop Advantages:**

* Color-coded output
* Mouse support
* Easier process killing (F9)
* Better visualization of CPU/Memory
* Tree view (F5)
* Search functionality (F3)
* Filter processes (F4)

**Process Finding and Filtering Tools**

**pgrep: Find Process IDs**

bash

*# Find all Python process IDs*

pgrep python

*# Find with full command matching*

pgrep -f "python script.py"

*# Find processes by user*

pgrep -u username

*# Count matching processes*

pgrep -c nginx

**Real-World Process Hunt Scenarios**

bash

*# Is MySQL running?*

pgrep -x mysqld && echo "MySQL is running" || echo "MySQL is down"

*# Find all processes consuming port 8080*

lsof -i :8080

*# Find process using most memory*

ps aux | awk '{print $6, $11}' | sort -rn | head -1

*# Find all Java processes with their memory usage*

ps aux | grep java | awk '{sum+=$6} END {print "Total RSS:", sum/1024 "MB"}'

**Part 3: Process Control - Conducting the Orchestra**

**Job Control: Background and Foreground**

**The Theater Stage Analogy** "Think of your terminal as a theater stage. Foreground processes are actors currently performing (blocking the stage). Background processes are backstage crew working behind the scenes."

**Foreground vs Background Execution**

bash

*# Foreground - blocks terminal*

sleep 100

*# Can't type anything until it completes*

*# Background - runs without blocking*

sleep 100 &

*# Terminal is free for other commands*

*# Check background jobs*

jobs

*# Job notation*

*# [1]+ Running sleep 100 &*

*# [2]- Running python script.py &*

**Moving Between Foreground and Background**

bash

*# Start in foreground*

ping google.com

*# Press Ctrl+Z to suspend*

*# Check suspended job*

jobs

*# [1]+ Stopped ping google.com*

*# Resume in background*

bg %1

*# Or just: bg*

*# Bring to foreground*

fg %1

*# Or just: fg*

*# Multiple jobs management*

sleep 100 & *# Job 1*

sleep 200 & *# Job 2*

sleep 300 & *# Job 3*

jobs *# List all*

fg %2 *# Bring job 2 to foreground*

**nohup: Surviving Logout**

**The Babysitter Analogy** "nohup is like hiring a babysitter for your process. Even when you (the parent) leave (logout), the babysitter ensures your process keeps running."

bash

*# Regular command dies when you logout*

python long\_running\_script.py &

*# Logout... process dies*

*# With nohup - survives logout*

nohup python long\_running\_script.py &

*# Logout... process continues!*

*# Output goes to nohup.out*

tail -f nohup.out

*# Redirect output*

nohup python script.py > output.log 2>&1 &

*# Disown running process*

python script.py &

disown %1

*# Now it survives logout*

**screen: Virtual Terminals**

**The Multiple Desktop Analogy** "screen is like having multiple computer desktops in one terminal. You can switch between them, leave them running, and come back later."

**Basic screen Usage**

bash

*# Start new screen session*

screen

*# Start with a name*

screen -S mysession

*# Detach from screen*

*# Press: Ctrl+A, then D*

*# List screen sessions*

screen -ls

*# Reattach to session*

screen -r mysession

*# Kill a session*

screen -X -S mysession quit

**Advanced screen Workflows**

bash

*# Create multiple windows in screen*

screen

*# Ctrl+A, C # Create new window*

*# Ctrl+A, N # Next window*

*# Ctrl+A, P # Previous window*

*# Ctrl+A, " # List windows*

*# Ctrl+A, K # Kill current window*

*# Split screen*

*# Ctrl+A, S # Horizontal split*

*# Ctrl+A, | # Vertical split*

*# Ctrl+A, Tab # Switch region*

*# Ctrl+A, X # Close region*

**tmux: screen's Modern Cousin**

"tmux is like screen's younger, cooler sibling - more features, better looks, easier to use."

bash

*# Start tmux*

tmux

*# New session with name*

tmux new -s devsession

*# Detach*

*# Ctrl+B, D*

*# List sessions*

tmux ls

*# Attach to session*

tmux attach -t devsession

*# Kill session*

tmux kill-session -t devsession

**tmux Power Features**

bash

*# Split panes*

*# Ctrl+B, % # Vertical split*

*# Ctrl+B, " # Horizontal split*

*# Ctrl+B, Arrow # Navigate panes*

*# Windows*

*# Ctrl+B, C # New window*

*# Ctrl+B, N # Next window*

*# Ctrl+B, P # Previous window*

*# Ctrl+B, 0-9 # Switch to window number*

*# Scrolling*

*# Ctrl+B, [ # Enter scroll mode*

*# Use arrow keys or Page Up/Down*

*# Q to exit scroll mode*

**Part 4: Signal Handling and Process Termination**

**Understanding Signals: Process Communication**

**The Doorbell Analogy** "Signals are like different types of doorbells for processes:

* **Gentle knock** (SIGTERM): 'Please finish what you're doing and exit'
* **Loud banging** (SIGKILL): 'Get out NOW!' (can't be ignored)
* **Doorbell** (SIGINT): 'Someone pressed Ctrl+C'
* **Phone call** (SIGHUP): 'Your terminal disconnected'"

**Common Signals Every DevOps Engineer Must Know**

bash

*# List all signals*

kill -l

*# The Essential Signals:*

*# 1 SIGHUP - Hangup (terminal disconnected)*

*# 2 SIGINT - Interrupt (Ctrl+C)*

*# 3 SIGQUIT - Quit (Ctrl+\\)*

*# 9 SIGKILL - Kill (cannot be caught or ignored)*

*# 15 SIGTERM - Terminate (polite request to exit)*

*# 18 SIGCONT - Continue (resume after stop)*

*# 19 SIGSTOP - Stop (cannot be caught or ignored)*

*# 20 SIGTSTP - Terminal stop (Ctrl+Z)*

**Sending Signals: The kill Command**

bash

*# Default signal (SIGTERM - 15)*

kill 1234

*# Specific signal by number*

kill -9 1234 *# SIGKILL*

*# Specific signal by name*

kill -TERM 1234

kill -KILL 1234

kill -HUP 1234

*# Kill multiple processes*

kill 1234 5678 9012

*# Kill all processes of a program*

killall python

killall -9 nginx

*# Kill by pattern*

pkill -f "python script.py"

*# Interactive kill*

pkill -i python *# Asks for confirmation*

**Signal Handling in Practice**

bash

*# Graceful shutdown (SIGTERM)*

kill -15 $(pgrep nginx)

*# Nginx finishes current requests, then exits*

*# Force kill (SIGKILL)*

kill -9 $(pgrep nginx)

*# Nginx dies immediately, may corrupt data*

*# Reload configuration (SIGHUP)*

kill -HUP $(pgrep nginx)

*# Nginx reloads config without dropping connections*

**The Right Way to Kill Processes**

**The Escalation Protocol**

bash

*# Step 1: Polite request (SIGTERM)*

kill PID

sleep 5

*# Step 2: Check if still running*

ps -p PID > /dev/null 2>&1

if [ $? -eq 0 ]; then

echo "Process still running, escalating..."

*# Step 3: Force kill (SIGKILL)*

kill -9 PID

fi

**Handling Zombie Processes**

bash

*# Find zombie processes*

ps aux | awk '$8 ~ /^Z/ { print $2, $11 }'

*# Find parent of zombie*

ps -o ppid= -p ZOMBIE\_PID

*# Kill parent to clean zombies*

kill -TERM PARENT\_PID

*# If parent won't die, force it*

kill -9 PARENT\_PID

**Part 5: Parent-Child Process Relationships**

**The Process Family Tree**

**The Royal Family Analogy** "In Linux, processes form a royal family tree:

* **init (PID 1)**: The Queen - mother of all processes
* **System services**: The nobles - critical system processes
* **User processes**: The citizens - your applications
* **Child processes**: The offspring - created by fork()"

**Understanding Process Hierarchy**

bash

*# View process tree*

pstree

*# Detailed tree with PIDs*

pstree -p

*# Tree for specific process*

pstree -p 1234

*# Show command line arguments*

pstree -a

*# Highlight specific process*

pstree -h PID

**Process Creation: fork() and exec()**

bash

*# See parent-child relationship*

ps -ef | head -20

*# Notice PPID column (Parent PID)*

*# Create parent-child demonstration*

python3 -c "

import os

import time

print(f'Parent PID: {os.getpid()}')

pid = os.fork()

if pid == 0:

# Child process

print(f'Child PID: {os.getpid()}, Parent: {os.getppid()}')

time.sleep(30)

else:

# Parent process

print(f'Created child with PID: {pid}')

time.sleep(30)

" &

*# Watch the relationship*

ps -ef | grep python

**Process Groups and Sessions**

bash

*# See process groups*

ps -eo pid,ppid,pgid,sid,cmd

*# All processes in a group*

ps -g GROUP\_ID

*# Kill entire process group*

kill -TERM -GROUP\_ID *# Note the minus sign*

*# Create new session*

setsid command

**Part 6: Real-World DevOps Scenarios**

**Scenario 1: The CPU Hog Hunt**

"Production server at 100% CPU. Find and fix in 60 seconds!"

bash

*# Step 1: Identify the culprit*

top -b -n 1 | head -20

*# or*

ps aux --sort=-%cpu | head -5

*# Step 2: Investigate the process*

lsof -p PID *# What files is it using?*

strace -p PID -c *# What system calls?*

*# Step 3: Take action*

renice +10 PID *# Lower priority*

*# or*

kill -STOP PID *# Pause it*

*# or*

kill -TERM PID *# Terminate gracefully*

**Scenario 2: The Memory Leak Detective**

bash

*# Find memory hogs*

ps aux --sort=-%mem | head -10

*# Monitor specific process memory growth*

watch -n 1 'ps -o pid,vsz,rss,comm -p PID'

*# Check for memory leaks*

while true; do

ps -o vsz= -p PID

sleep 5

done | awk '{print NR, $1}' > memory\_trend.txt

*# Graph it (if gnuplot available)*

gnuplot -e "plot 'memory\_trend.txt' with lines; pause -1"

**Scenario 3: The Zombie Apocalypse**

bash

*# Count zombies*

ps aux | awk '$8 ~ /Z/ { count++ } END { print "Zombies:", count }'

*# Find zombie details*

ps aux | awk '$8 ~ /Z/ { print $2, $11 }' | while read zpid cmd; do

echo "Zombie PID: $zpid, Command: $cmd"

ppid=$(ps -o ppid= -p $zpid)

echo " Parent PID: $ppid"

ps -p $ppid -o comm=

done

*# Clean up zombies*

for ppid in $(ps aux | awk '$8 ~ /Z/ { print }' | awk '{print $3}' | sort -u); do

echo "Killing parent $ppid to clean zombies"

kill -TERM $ppid

done

**Scenario 4: The Runaway Process Farm**

"A buggy script spawned 1000 processes!"

bash

*# Quick fix - kill all Python processes (example)*

pkill -9 python

*# Safer approach - kill specific script*

pkill -f "script\_name.py"

*# Even safer - confirm before killing*

pgrep -f "script\_name.py" | while read pid; do

ps -p $pid -o pid,cmd

read -p "Kill this process? (y/n): " answer

[ "$answer" = "y" ] && kill $pid

done

**Scenario 5: Long-Running Job Management**

bash

*# Start a long job that survives logout*

nohup python data\_processing.py > processing.log 2>&1 &

echo $! > processing.pid *# Save PID*

*# Check on it later*

tail -f processing.log

*# Monitor resource usage*

pid=$(cat processing.pid)

while ps -p $pid > /dev/null; do

ps -o %cpu,%mem,etime -p $pid

sleep 10

done

*# Set up email notification when done*

pid=$(cat processing.pid)

while ps -p $pid > /dev/null; do sleep 60; done

echo "Processing complete" | mail -s "Job Done" your@email.com

**Part 7: Performance Monitoring Best Practices**

**Building Your Monitoring Toolkit**

bash

#!/bin/bash

*# Ultimate system health check script*

echo "=== System Health Report ==="

echo "Date: $(date)"

echo

echo "=== CPU Top 5 ==="

ps aux --sort=-%cpu | head -6

echo -e "\\n=== Memory Top 5 ==="

ps aux --sort=-%mem | head -6

echo -e "\\n=== Process Count by State ==="

ps aux | awk 'NR>1 {state[$8]++} END {for(s in state) print s": "state[s]}'

echo -e "\\n=== Zombie Processes ==="

ps aux | awk '$8 ~ /Z/ { print $2, $11 }'

echo -e "\\n=== Load Average ==="

uptime

echo -e "\\n=== Memory Status ==="

free -h

echo -e "\\n=== Disk Usage ==="

df -h | grep -vE '^tmpfs|^udev'

**Creating Process Monitoring Alerts**

bash

#!/bin/bash

*# Process monitor with alerts*

MAX\_CPU=80

MAX\_MEM=80

*# Check for high CPU usage*

ps aux | awk -v max=$MAX\_CPU '$3 > max {print "HIGH CPU:", $3"%", $11}' | while read alert; do

echo "[ALERT] $alert"

*# Send to monitoring system*

*# curl -X POST monitoring.api/alert -d "$alert"*

done

*# Check for high memory usage*

ps aux | awk -v max=$MAX\_MEM '$4 > max {print "HIGH MEM:", $4"%", $11}' | while read alert; do

echo "[ALERT] $alert"

done

*# Check for zombies*

zombie\_count=$(ps aux | awk '$8 ~ /Z/ { count++ } END { print count+0 }')

if [ $zombie\_count -gt 10 ]; then

echo "[ALERT] High zombie count: $zombie\_count"

fi

Day 4

**Class 4: System Resources and Performance Monitoring**

**Opening Hook & Problem Statement**

**The 3 AM Crisis Call**

"It's 3 AM. Your phone is ringing. On the other end is your panicked colleague: 'The production server is down! Users can't access our application!' You groggily open your laptop, SSH into the server... and then what? The server is 'running' but something is clearly wrong. Is it running out of disk space? Is memory exhausted? Is the CPU overloaded? Or is it something else entirely?

You have exactly 5 minutes before your manager joins the call, expecting answers. The company is losing $10,000 every minute the service is down. What do you check first?"

**Question to Students:** "How many of you have ever had your laptop freeze or become incredibly slow? What did you do? Restart it? But what if you're managing a server that can't be restarted because 10,000 users are currently connected?"

**The Hospital Analogy**

"Think of a Linux server like a patient in a hospital. When a patient arrives at the emergency room, doctors don't immediately perform surgery. They check vital signs first:

* **Heart rate** (CPU usage)
* **Blood pressure** (System load)
* **Oxygen levels** (Memory availability)
* **Body temperature** (Disk I/O)

Today, you're going to become the doctor for Linux systems. You'll learn to read these vital signs, diagnose problems, and most importantly, prevent system failures before they happen."

**Module Introduction: Why Performance Monitoring Matters**

**The Hidden Cost of Poor Performance**

**Real-World Impact:**

* Amazon found that every 100ms of latency cost them 1% in sales
* Google discovered that a 500ms delay in search results caused a 20% drop in traffic
* A major Indian e-commerce platform lost ₹100 crores during a 2-hour downtime on Diwali sale day

**The DevOps Reality:** "As a DevOps engineer, you're not just keeping servers running - you're protecting revenue, user experience, and your company's reputation. One unnoticed performance issue can cascade into a complete system failure."

**What You'll Master Today**

By the end of this class, you'll be able to:

1. **Diagnose** system issues in under 60 seconds using the right commands
2. **Predict** failures before they happen by understanding resource patterns
3. **Identify** the exact bottleneck when your application slows down
4. **Optimize** system performance based on real metrics, not guesswork
5. **Create** monitoring scripts that alert you before users notice problems

**Part 1: Disk Space - The Silent Killer**

**The Restaurant Kitchen Analogy**

"Imagine you're running a busy restaurant kitchen. You have:

* **Refrigerators** (Hard drives) - Where you store all ingredients
* **Counter space** (RAM) - Where you actively prepare food
* **Chefs** (CPU) - Who do the actual cooking
* **Order tickets** (Processes) - Tasks that need to be completed

What happens when your refrigerator is full? You can't store new ingredients. Orders start backing up. Eventually, the entire kitchen stops functioning. This is exactly what happens when a server runs out of disk space!"

**Understanding Disk Usage: The Foundation**

**Why Disk Space Matters More Than You Think**

**The Domino Effect of Full Disks:**

Full Disk → Can't write logs → Can't debug issues

→ Can't create temp files → Applications crash

→ Can't save user data → Data loss

→ Can't update system → Security vulnerabilities

→ Database can't write → Complete service failure

**Student Activity:** "Let me show you something terrifying..."

bash

*# Simulate what happens when disk is full (DON'T RUN ON PRODUCTION!)*

*# Create a large file that fills up /tmp*

dd if=/dev/zero of=/tmp/bigfile bs=1M count=1000

*# Now try to do normal operations*

echo "Hello" > /tmp/test.txt

*# Error: No space left on device*

*# Even system commands might fail!*

sudo apt update

*# Could fail because it needs to write to disk*

**Command Deep Dive: df (Disk Free)**

**The Emergency Room X-Ray**

bash

*# Basic usage - Human readable format*

df -h

*# What you're actually seeing:*

Filesystem Size Used Avail Use% Mounted on

/dev/sda1 20G 15G 4.0G 79% /

/dev/sda2 100G 40G 55G 43% /home

tmpfs 2.0G 0 2.0G 0% /dev/shm

**Breaking Down Each Column:**

* **Filesystem**: The storage device or partition
* **Size**: Total capacity (like your refrigerator size)
* **Used**: Space already occupied (ingredients in the fridge)
* **Avail**: Space you can still use (room for more ingredients)
* **Use%**: THE CRITICAL METRIC - your early warning system
* **Mounted on**: Where this storage appears in your system

**DevOps Best Practices:**

bash

*# Check all filesystems including pseudo filesystems*

df -a

*# Focus on specific filesystem type*

df -t ext4

*# Exclude certain filesystem types (useful for monitoring scripts)*

df -x tmpfs -x devtmpfs

*# Show inode usage (file count limit)*

df -i

**Critical Insight - The 5% Reserved Space Mystery:** "Notice that Used + Avail doesn't equal Size? Linux reserves 5% of disk space for root user. This prevents complete system lockup when regular users fill the disk."

**Command Deep Dive: du (Disk Usage)**

**The Detective Tool - Finding Space Hogs**

"While df shows you the overall picture, du is your magnifying glass. It helps you find exactly what's eating your disk space."

bash

*# Basic usage - Current directory*

du -h

*# The most useful command you'll ever learn*

du -sh \* | sort -h

*# Find the top 10 space consumers*

du -ah /var | sort -rh | head -10

*# Real DevOps scenario - Finding large log files*

du -sh /var/log/\* | sort -rh | head -5

**Advanced Detective Work:**

bash

*# Find folders larger than 1GB*

find / -type d -size +1G 2>/dev/null

*# Track disk usage over time (create your own monitoring)*

while true; do

date

df -h / | tail -1

sleep 60

done > disk\_monitor.log

**Command Deep Dive: lsblk (List Block Devices)**

**Understanding Your Storage Architecture**

bash

*# See the complete storage hierarchy*

lsblk

*# More detailed view with filesystem info*

lsblk -f

*# Include device ownership and permissions*

lsblk -m

**What This Reveals:**

NAME MAJ:MIN RM SIZE RO TYPE MOUNTPOINT

sda 8:0 0 500G 0 disk

├─sda1 8:1 0 50G 0 part /

├─sda2 8:2 0 200G 0 part /home

└─sda3 8:3 0 250G 0 part /data

"This is like having an architectural blueprint of your storage system!"

**Real-World Disk Crisis Management**

**Scenario: Production Server at 95% Disk Usage**

bash

*# Step 1: Identify the problem*

df -h

*# Output: / is at 95%*

*# Step 2: Find the culprit*

du -sh /\* 2>/dev/null | sort -rh | head -5

*# Output: /var is using 80GB*

*# Step 3: Drill down*

du -sh /var/\* | sort -rh | head -5

*# Output: /var/log is using 75GB*

*# Step 4: Find specific files*

find /var/log -type f -size +1G -exec ls -lh {} \\;

*# Output: application.log is 70GB*

*# Step 5: Safe cleanup*

*# First, backup if needed*

tail -n 10000 /var/log/application.log > /tmp/recent\_app.log

*# Then truncate (safer than deleting)*

> /var/log/application.log

*# Or use logrotate for proper management*

logrotate -f /etc/logrotate.d/application

**Part 2: Memory - The Performance Multiplier**

**The Office Desk Analogy**

"Imagine your computer's memory like an office workspace:

* **Hard Drive** = Filing cabinet (slow to access, lots of storage)
* **RAM** = Your desk (fast to access, limited space)
* **CPU Cache** = The papers right in front of you (instant access, tiny space)
* **Swap** = That overflow table in the corner (backup space, but walking there is slow)

When your desk (RAM) gets full, you start putting papers on the overflow table (swap). Now every time you need those papers, you have to walk over there (slow). This is why running out of RAM makes everything slow!"

**Understanding Linux Memory Management**

**The Brilliant Memory Illusion**

"Linux does something clever - it lies to applications about memory! Each application thinks it has the entire system's memory to itself. This is called Virtual Memory."

bash

*# See the magic in action*

free -h

*# What you'll see:*

total used free shared buff/cache available

Mem: 16Gi 8.2Gi 1.4Gi 352Mi 6.4Gi 7.2Gi

Swap: 8.0Gi 2.0Gi 6.0Gi

**The Memory Mystery - Why Used + Free ≠ Total:**

* **Buffers**: Temporary storage for data being written to disk
* **Cache**: Recently accessed files kept in memory for speed
* **Available**: What applications can actually use (the REAL metric)

**Command Deep Dive: free**

**Your Memory Dashboard**

bash

*# Human readable*

free -h

*# See memory changes in real-time*

free -h -s 2

*# Detailed breakdown with totals*

free -h -t

*# The one-liner health check*

free -h | grep Mem | awk '{print "Memory Usage: " $3 "/" $2 " (" int($3/$2 \* 100) "%)"}'

**Critical Understanding - Linux Memory Philosophy:** "Unused RAM is wasted RAM! Linux uses 'free' memory for caching. This is GOOD! Don't panic if free memory seems low."

**Command Deep Dive: vmstat (Virtual Memory Statistics)**

**The Real-Time Performance Monitor**

bash

*# Basic usage - snapshot*

vmstat

*# The powerful real-time view (every 2 seconds)*

vmstat 2

*# What the columns mean:*

procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----

r b swpd free buff cache si so bi bo in cs us sy id wa st

2 0 245003 1456789 234567 6789012 0 0 45 123 567 890 23 5 70 2 0

**Decoding the Matrix:**

* **r**: Processes waiting for CPU (runnable)
* **b**: Processes in uninterruptible sleep (blocked)
* **si/so**: Swap in/out (BAD if consistently high)
* **bi/bo**: Blocks read/written to disk
* **us**: User CPU time %
* **sy**: System CPU time %
* **id**: Idle CPU % (lower is busier)
* **wa**: Waiting for I/O % (disk bottleneck indicator)

**The Golden Rule:** "If swap in/out (si/so) is consistently non-zero, you have a memory problem!"

**The Secret Files: /proc/meminfo**

**Behind the Scenes Intelligence**

bash

*# The complete memory story*

cat /proc/meminfo

*# Key metrics to monitor*

grep -E "MemTotal|MemFree|MemAvailable|Buffers|Cached|SwapTotal|SwapFree" /proc/meminfo

*# Create a memory pressure test*

stress --vm 2 --vm-bytes 1G --timeout 30s &

watch -n 1 'free -h; echo; grep -E "MemAvailable|SwapFree" /proc/meminfo'

**Memory Troubleshooting Patterns**

**Pattern 1: The Memory Leak Detection**

bash

*# Monitor a specific process memory usage*

PID=$(pgrep python3)

while true; do

ps aux | grep $PID | awk '{print strftime("%Y-%m-%d %H:%M:%S"), $6}'

sleep 10

done

**Pattern 2: Finding Memory Hogs**

bash

*# Top 10 memory consumers*

ps aux --sort=-%mem | head -11

*# Detailed memory map of a process*

pmap -x $PID

*# Watch memory usage sorted by consumption*

top -o %MEM

**Part 3: CPU - The Heartbeat of Your System**

**The Kitchen Brigade Analogy**

"A CPU is like a kitchen with multiple chefs (cores):

* **Single-core**: One chef doing everything (sequential)
* **Multi-core**: Multiple chefs working in parallel
* **Hyperthreading**: Each chef can work on two dishes simultaneously
* **Load Average**: Number of order tickets waiting to be cooked

If you have 4 chefs (cores) and 4 orders (load average = 4.0), perfect! If you have 4 chefs and 8 orders (load average = 8.0), orders are backing up!"

**Command Deep Dive: uptime**

**The Quick Health Check**

bash

*# Basic uptime*

uptime

*# Output analysis:*

14:23:01 up 45 days, 3:27, 5 users, load average: 2.45, 3.12, 2.89

*# └─────────────────────────────┘*

*# 1-min 5-min 15-min averages*

**The Load Average Decoder:**

bash

*# Get number of CPU cores*

nproc

*# If nproc = 4, then:*

*# Load < 4.0: System is healthy*

*# Load = 4.0: System at capacity*

*# Load > 4.0: Processes are waiting*

*# Quick system health score*

echo "CPU Cores: $(nproc)"

echo "Current Load: $(uptime | awk -F'load average:' '{print $2}')"

echo "Health Score: $(uptime | awk -F'load average:' '{print $2}' | awk '{printf "%.2f%%", ($1/$(nproc))\*100}')"

**Command Deep Dive: iostat**

**The I/O Detective**

bash

*# Install if not available*

sudo apt-get install sysstat

*# Basic CPU and I/O stats*

iostat

*# Continuous monitoring (every 2 seconds)*

iostat -x 2

*# Focus on CPU stats*

iostat -c 2

*# The DevOps special - extended disk statistics*

iostat -xz 2

**Critical Metrics to Watch:**

* **%iowait**: High values = disk bottleneck
* **%steal**: In virtual machines, CPU stolen by hypervisor
* **%idle**: Low values = CPU bottleneck
* **await**: Average time for I/O requests (ms)

**Command Deep Dive: sar (System Activity Reporter)**

**The Time Machine for System Metrics**

bash

*# Enable data collection*

sudo systemctl enable sysstat

sudo systemctl start sysstat

*# View CPU history*

sar -u

*# Memory usage history*

sar -r

*# Network statistics*

sar -n DEV

*# Everything from yesterday*

sar -A -f /var/log/sysstat/sa$(date -d yesterday +%d)

*# Create custom monitoring*

sar -u 1 10 > cpu\_report.txt

**The Power of Historical Data:** "sar is like a flight recorder for your system. When something crashed at 3 AM, sar tells you exactly what happened!"

**Part 4: Understanding System Load**

**The Traffic Jam Analogy**

"System load is like traffic on a highway:

* **CPU Cores** = Number of lanes
* **Processes** = Cars
* **Load Average** = Average number of cars waiting

A 4-lane highway (4 cores) with 4 cars (load = 4.0) flows perfectly. With 8 cars trying to use 4 lanes (load = 8.0), you have congestion. With 16 cars (load = 16.0), you have a traffic jam!"

**Load Average Deep Dive**

bash

*# The three numbers explained*

cat /proc/loadavg

*# 2.34 3.45 2.89 3/1247 28959*

*# 1min 5min 15min running/total last\_pid*

*# Visual load monitoring*

watch -n 1 'echo "=== SYSTEM LOAD ==="; uptime; echo; echo "=== PER CORE LOAD ==="; uptime | awk -F"load average:" "{print \\$2}" | awk "{print \\"1-min: \\" \\$1/$(nproc)\*100 \\"%\\"; print \\"5-min: \\" \\$2/$(nproc)\*100 \\"%\\"; print \\"15-min: \\" \\$3/$(nproc)\*100 \\"%\\"}"'

**The Load Investigation Framework**

bash

*# High load? Follow this diagnosis flow:*

*# 1. Check what kind of load*

top

*# Look at us (user) vs sy (system) vs wa (iowait)*

*# 2. If CPU bound (high us/sy):*

ps aux --sort=-%cpu | head

*# 3. If I/O bound (high wa):*

iotop *# or iostat -x 2*

*# 4. If memory pressure:*

vmstat 2

*# Watch si/so columns*

*# 5. Find the culprit process*

ps -eo pid,ppid,cmd,%cpu,%mem,stat,start --sort=-%cpu | head -20

**Part 5: Performance Bottleneck Identification**

**The Master Diagnosis Framework**

**The 60-Second Analysis**

"When a system is slow, you have 60 seconds to identify the bottleneck. Here's your checklist:"

bash

*# The Ultimate 60-Second Diagnosis Script*

echo "=== 60-SECOND SYSTEM ANALYSIS ==="

echo

echo "1. UPTIME & LOAD"

uptime

echo

echo "2. DISK SPACE"

df -h | grep -vE '^tmpfs|^udev'

echo

echo "3. MEMORY"

free -h

echo

echo "4. TOP PROCESSES"

ps aux --sort=-%cpu | head -5

echo

echo "5. RECENT SYSTEM MESSAGES"

dmesg | tail -5

echo

echo "6. NETWORK CONNECTIONS"

ss -tulpn | head -10

echo

echo "7. DISK I/O"

iostat -x 1 1

**Common Bottleneck Patterns**

**Pattern 1: CPU Bottleneck**

**Symptoms:**

* Load average > number of cores
* CPU usage near 100%
* Slow response times

**Diagnosis:**

bash

*# Check CPU-hungry processes*

top -b -n 1 | head -20

*# See CPU usage per core*

mpstat -P ALL 1

*# Check for runaway processes*

ps aux | awk '$3 > 50.0'

**Pattern 2: Memory Bottleneck**

**Symptoms:**

* High swap usage
* System becomes unresponsive
* OOM (Out of Memory) killer activating

**Diagnosis:**

bash

*# Check for memory leaks*

ps aux --sort=-%mem | head -10

*# Monitor swap activity*

vmstat 1 | awk '{print $7, $8}'

*# Check OOM killer activity*

dmesg | grep -i "killed process"

**Pattern 3: Disk I/O Bottleneck**

**Symptoms:**

* High iowait in top/iostat
* Slow file operations
* Database queries taking forever

**Diagnosis:**

bash

*# Check disk utilization*

iostat -xz 1

*# Find processes doing heavy I/O*

iotop -o

*# Check for disk errors*

dmesg | grep -i "error\\|fail" | grep -i "sd\\|disk"

**Pattern 4: Network Bottleneck**

**Symptoms:**

* Slow application response
* Timeout errors
* High network latency

**Diagnosis:**

bash

*# Check network throughput*

iftop *# or nload*

*# See network errors*

netstat -i

*# Check connection states*

ss -s

**Hands-On Lab: The Production Crisis Simulation**

**Scenario: E-Commerce Website Black Friday Crisis**

"It's Black Friday. Your e-commerce site is experiencing 10x normal traffic. Users are complaining about slow page loads. Your job: diagnose and fix the issue in real-time."

**Phase 1: Initial Assessment**

bash

*# Quick system health check*

uptime

free -h

df -h

**Student Task:** "Based on these outputs, what's your initial hypothesis?"

**Phase 2: Deep Dive Investigation**

bash

*# Let's simulate high load*

stress --cpu 2 --io 4 --vm 2 --vm-bytes 128M --timeout 60s &

*# Now monitor in real-time*

watch -n 1 'uptime; echo; free -h; echo; iostat -c 1 1'

**Phase 3: Finding the Culprit**

bash

*# Which process is causing issues?*

top -b -n 1 | head -20

*# Is it CPU, Memory, or I/O?*

vmstat 2 5

*# Get detailed process info*

ps -eo pid,ppid,cmd,%cpu,%mem,stat,etime --sort=-%cpu | head -10

**Phase 4: Taking Action**

bash

*# Kill the problematic process*

kill -15 $PID *# Graceful termination*

*# or*

kill -9 $PID *# Force kill*

*# Monitor recovery*

watch -n 1 'uptime; free -h'

**Creating Your Monitoring Toolkit**

**The DevOps Monitoring Script**

bash

#!/bin/bash

*# save as system\_monitor.sh*

*# Configuration*

THRESHOLD\_CPU=80

THRESHOLD\_MEM=90

THRESHOLD\_DISK=85

LOG\_FILE="/var/log/system\_monitor.log"

*# Function to check CPU*

check\_cpu() {

cpu\_usage=$(top -b -n1 | grep "Cpu(s)" | awk '{print $2}' | cut -d'%' -f1)

if (( $(echo "$cpu\_usage > $THRESHOLD\_CPU" | bc -l) )); then

echo "$(date): WARNING - CPU usage is ${cpu\_usage}%" >> $LOG\_FILE

return 1

fi

return 0

}

*# Function to check memory*

check\_memory() {

mem\_usage=$(free | grep Mem | awk '{print int($3/$2 \* 100)}')

if [ $mem\_usage -gt $THRESHOLD\_MEM ]; then

echo "$(date): WARNING - Memory usage is ${mem\_usage}%" >> $LOG\_FILE

return 1

fi

return 0

}

*# Function to check disk*

check\_disk() {

df -h | grep -vE '^tmpfs|^udev|^Filesystem' | while read line; do

usage=$(echo $line | awk '{print $5}' | cut -d'%' -f1)

partition=$(echo $line | awk '{print $1}')

if [ $usage -gt $THRESHOLD\_DISK ]; then

echo "$(date): WARNING - Disk usage on $partition is ${usage}%" >> $LOG\_FILE

return 1

fi

done

return 0

}

*# Main monitoring loop*

while true; do

check\_cpu

check\_memory

check\_disk

sleep 60

done

**The One-Liner Arsenal**

bash

*# Memory usage percentage*

free | grep Mem | awk '{printf "%.1f%%\\n", $3/$2 \* 100}'

*# CPU usage percentage*

top -b -n1 | grep "Cpu(s)" | awk '{print $2"%"}'

*# Disk usage for root partition*

df -h / | tail -1 | awk '{print $5}'

*# Load per CPU core*

echo "scale=2; $(uptime | awk -F'load average:' '{print $2}' | awk '{print $1}')/$(nproc)" | bc

*# Top 5 memory consumers*

ps aux --sort=-%mem | head -6 | tail -5 | awk '{print $11, $4"%"}'

*# Find large files modified in last 24 hours*

find / -type f -mtime -1 -size +100M 2>/dev/null

*# Monitor specific process*

watch -n 1 'ps aux | grep [p]ython'

**Advanced Performance Tuning Tips**

**The Performance Hierarchy**

1. **Eliminate** - Remove unnecessary processes
2. **Optimize** - Tune existing processes
3. **Cache** - Store frequently accessed data in memory
4. **Scale** - Add more resources only as last resort

**Quick Wins for Performance**

bash

*# 1. Clear package cache*

sudo apt-get clean

*# 2. Remove old kernels*

sudo apt-get autoremove

*# 3. Clear system cache (careful in production!)*

sync && echo 3 > /proc/sys/vm/drop\_caches

*# 4. Optimize swappiness (for servers with plenty of RAM)*

echo "vm.swappiness=10" >> /etc/sysctl.conf

sysctl -p

*# 5. Check for zombie processes*

ps aux | grep defunct

*# 6. Analyze and optimize logged errors*

journalctl -p err -b

Day 5

**Class 5: Text Processing and Pattern Matching**

**Opening Hook & Problem Statement**

**The Log Analysis Crisis**

**Setting the Scene:** "It's 3 AM. You're the on-call DevOps engineer for an e-commerce platform handling Black Friday traffic. Your phone explodes with alerts - the payment service is failing intermittently. You SSH into the server and find yourself staring at 50GB of log files from the past 24 hours."

**The Challenge:**

* 10 million log lines across 200 files
* Error messages buried somewhere in the noise
* Patterns that only appear when specific conditions align
* Customer transactions failing every 7-10 minutes
* Your CEO is texting: "How long until this is fixed?"

**The Dilemma:** "Opening these files in a text editor? Your laptop would crash. Reading them line by line? That would take weeks. But somewhere in those 50GB is a pattern - maybe a specific API endpoint timing out, maybe a database connection string malformed, maybe a memory leak manifesting every 1000th request."

**Question to Students:** "How many of you have ever tried to find a specific error in hundreds of log files? Or tried to extract data from messy CSV files? Or needed to compare configuration files across servers?"

**The Reality Check:** "By the end of this class, you'll be able to analyze those 50GB of logs in under 30 seconds, extract any pattern you need, and identify the root cause faster than senior engineers opening GUI tools. Today, we master the dark arts of text processing."

**Why Text Processing is a Superpower in DevOps**

**Everything is Text in Linux**

**The Universal Truth:** "In the Linux world, EVERYTHING is text. Configuration files, logs, source code, API responses, database exports, system metrics - all text. Master text processing, and you master Linux."

**Real DevOps Scenarios Where Text Processing Saves the Day:**

1. **Log Analysis and Debugging**
   * Finding error patterns across distributed systems
   * Tracking user journeys through microservices
   * Identifying performance bottlenecks from access logs
2. **Configuration Management**
   * Updating configs across 100s of servers
   * Finding configuration drift between environments
   * Extracting and validating environment variables
3. **Data Migration and ETL**
   * Cleaning messy CSV exports
   * Transforming data formats between systems
   * Validating data integrity during migrations
4. **Security and Compliance**
   * Finding exposed secrets in codebases
   * Analyzing authentication logs for suspicious patterns
   * Extracting audit trails for compliance reports
5. **Performance Analysis**
   * Parsing application metrics
   * Analyzing database slow query logs
   * Processing system performance data

**The Text Processing Toolkit Hierarchy**

**Analogy: The Carpenter's Tools** "A master carpenter doesn't use a sledgehammer to hang a picture frame or a tiny hammer to demolish a wall. Similarly, in text processing, we have different tools for different scales:"

Simple Search → grep (find the nail)

Pattern Matching → regex (measure precisely)

Stream Editing → sed (sand and polish)

Data Extraction → awk (cut to size)

File Comparison → diff/comm (quality control)

**Part 1: Regular Expressions - The Pattern Language**

**The Pattern Recognition Problem**

**Scenario Building:** "Your application logs look like this:"

2024-03-15 10:23:45 ERROR [api-gateway] User 12345 failed login from 192.168.1.100

2024-03-15 10:23:46 INFO [payment-service] Transaction tx\_9876 completed $45.99

2024-03-15 10:23:47 WARN [database] Connection pool at 85% capacity

2024-03-15 10:23:48 ERROR [api-gateway] User 67890 failed login from 10.0.0.50

**The Challenge:** "How do you find:

* All ERROR messages from api-gateway?
* All IP addresses attempting logins?
* All transaction amounts over $100?
* All timestamps between 10:20 and 10:25?
* All user IDs that are exactly 5 digits?"

**Regular Expressions: The Pattern Matching Language**

**Analogy: Regex as a Detective's Toolkit** "Think of regex like a detective's description of a suspect: 'Looking for someone wearing a red shirt (pattern), between 5-6 feet tall (range), with either brown or black hair (alternatives), carrying exactly 2 bags (quantity).'"

**Regex Fundamentals: Building Blocks**

**1. Literal Characters - The Basics**

bash

*# Finding exact text*

echo "Error in payment service" | grep "payment"

*# Matches: payment*

*# Case-sensitive by default*

echo "ERROR: Database connection failed" | grep "error"

*# No match (ERROR ≠ error)*

echo "ERROR: Database connection failed" | grep -i "error"

*# Match found (case-insensitive)*

**2. Metacharacters - The Special Agents**

**The Dot (.) - The Wildcard**

bash

*# . matches any single character*

echo "cat bat mat rat" | grep -o "..t"

*# Matches: cat, bat, mat, rat*

*# Real-world: Finding log entries with any severity*

echo -e "2024-03-15 ERROR msg\\n2024-03-15 DEBUG msg" | grep "2024-03-15 ..... msg"

*# Matches both ERROR and DEBUG (5 characters)*

**The Anchors (^ and $) - The Boundaries**

bash

*# ^ matches start of line*

echo -e "error at start\\nend has error" | grep "^error"

*# Matches only: error at start*

*# $ matches end of line*

echo -e "file.txt\\nfile.txt.backup" | grep ".txt$"

*# Matches only: file.txt*

*# Real-world: Finding completed transactions*

tail -n 100 transaction.log | grep "COMPLETED$"

**3. Character Classes - The Categories**

**Square Brackets [] - The Selector**

bash

*# Match any single character from the set*

echo "Test1 Test2 TestA TestB" | grep -o "Test[12AB]"

*# Matches: Test1, Test2, TestA, TestB*

*# Ranges*

echo "User123 User456 UserABC" | grep -o "User[0-9][0-9][0-9]"

*# Matches: User123, User456*

*# Negation with ^*

echo "log1.txt log2.txt logA.txt" | grep -o "log[^0-9].txt"

*# Matches: logA.txt*

**Predefined Character Classes - The Shortcuts**

bash

*# \\d = [0-9] (digits)*

*# \\w = [a-zA-Z0-9\_] (word characters)*

*# \\s = [ \\t\\n\\r\\f] (whitespace)*

*# Finding IP addresses (simplified)*

echo "Server at 192.168.1.100 is down" | grep -oE "[0-9]+\\.[0-9]+\\.[0-9]+\\.[0-9]+"

*# Matches: 192.168.1.100*

*# Finding email addresses*

echo "Contact admin@example.com or support@test.org" | grep -oE "\\w+@\\w+\\.\\w+"

*# Matches: admin@example.com, support@test.org*

**4. Quantifiers - The Counters**

**The Repetition Operators**

bash

*# \* = zero or more*

echo "color colour" | grep -o "colou\*r"

*# Matches both: color (0 'u's), colour (1 'u')*

*# + = one or more*

echo "1 12 123 1234" | grep -oE "[0-9]+"

*# Matches all numbers*

*# ? = zero or one (optional)*

echo "http https" | grep -o "https\\?"

*# Matches both: http, https*

*# {n} = exactly n times*

echo "1234 12345 123456" | grep -oE "[0-9]{5}"

*# Matches: 12345*

*# {n,m} = between n and m times*

echo "a aa aaa aaaa" | grep -oE "a{2,3}"

*# Matches: aa, aaa*

**Advanced Regex Patterns for DevOps**

**1. Capturing Groups and Backreferences**

bash

*# Parentheses create capturing groups*

echo "Error Code: 500, Message: Internal Server Error" | sed -E 's/Error Code: ([0-9]+).\*/\\1/'

*# Extracts: 500*

*# Backreferences - finding duplicates*

echo "the the quick brown fox" | grep -E "\\b(\\w+)\\s+\\1\\b"

*# Matches: the the*

*# Real-world: Extracting data from logs*

log\_line="2024-03-15 10:23:45 user\_id=12345 action=login status=failed"

echo "$log\_line" | sed -E 's/.\*user\_id=([0-9]+).\*status=([a-z]+).\*/User \\1: \\2/'

*# Output: User 12345: failed*

**2. Lookarounds - The Context Checkers**

bash

*# Positive lookahead (?=) - must be followed by*

echo "test123 test456 testABC" | grep -oP "test(?=[0-9]+)"

*# Matches: test (only when followed by numbers)*

*# Negative lookahead (?!) - must NOT be followed by*

echo "test.txt test.log test.tmp" | grep -oP "test\\.(?!txt)"

*# Matches: test. (when not followed by txt)*

*# Lookbehind (PCRE - Perl Compatible)*

echo "USD100 EUR200 GBP300" | grep -oP "(?<=USD)[0-9]+"

*# Matches: 100*

**Real-World Regex Patterns Every DevOps Engineer Needs**

bash

*# 1. IP Address Validation (accurate)*

ip\_regex="^((25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9]?[0-9])\\.){3}(25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9]?[0-9])$"

*# 2. Email Extraction*

email\_regex="[a-zA-Z0-9.\_%+-]+@[a-zA-Z0-9.-]+\\.[a-zA-Z]{2,}"

*# 3. URL Parsing*

url\_regex="https?://[a-zA-Z0-9.-]+\\.[a-zA-Z]{2,}(/[^\\\\s]\*)?"

*# 4. Log Timestamp Extraction*

timestamp\_regex="[0-9]{4}-[0-9]{2}-[0-9]{2}\\s+[0-9]{2}:[0-9]{2}:[0-9]{2}"

*# 5. AWS Resource IDs*

aws\_regex="arn:aws:[a-z0-9-]+:[a-z0-9-]\*:[0-9]{12}:[a-z0-9-/]+"

*# 6. Docker Image Tags*

docker\_regex="[a-z0-9-]+/[a-z0-9-]+:[a-z0-9.-]+"

*# 7. Kubernetes Resource Names*

k8s\_regex="[a-z0-9]([-a-z0-9]\*[a-z0-9])?"

*# 8. Credit Card Masking*

cc\_regex="[0-9]{4}[\\s-]?[0-9]{4}[\\s-]?[0-9]{4}[\\s-]?[0-9]{4}"

*# 9. Password Strength Check*

password\_regex="^(?=.\*[a-z])(?=.\*[A-Z])(?=.\*[0-9])(?=.\*[@$!%\*?&])[A-Za-z0-9@$!%\*?&]{8,}$"

*# 10. JSON Key-Value Extraction*

json\_regex='"([^"]+)":\\s\*"([^"]+)"'

**Part 2: sed - The Stream Editor**

**The Stream Processing Problem**

**Scenario:** "You have 100 configuration files across your servers. The database server IP just changed from 10.0.1.50 to 10.0.2.100. You need to update all config files, but also:

* Keep backups of original files
* Only change the database IP, not other IPs
* Add a timestamp comment when the change was made
* Do this without opening a single file in an editor"

**Understanding sed: The Assembly Line**

**Analogy: The Document Assembly Line** "Imagine sed as a factory assembly line for text. Documents flow through on a conveyor belt (stream), and at each station (command), specific modifications are made - replace this word, delete that line, insert a comment here - all without stopping the line."

**sed Fundamentals**

**Basic Syntax and Operation**

bash

*# sed 's/pattern/replacement/flags' file*

*# Simple substitution*

echo "Hello World" | sed 's/World/DevOps/'

*# Output: Hello DevOps*

*# The 's' command structure:*

*# s = substitute command*

*# / = delimiter (can be any character)*

*# pattern = what to find (can be regex)*

*# replacement = what to replace with*

*# flags = modifiers (g=global, i=case-insensitive, etc.)*

**sed's Pattern Space: The Workshop**

bash

*# sed processes one line at a time in its "pattern space"*

*# Think of it as sed's workspace where it performs operations*

*# Visualizing sed's workflow:*

echo -e "line1\\nline2\\nline3" | sed 's/line/LINE/'

*# Process:*

*# 1. Read "line1" into pattern space*

*# 2. Apply s/line/LINE/*

*# 3. Print "LINE1"*

*# 4. Clear pattern space, repeat for next line*

**Essential sed Commands**

**1. Substitution (s) - The Replacer**

bash

*# Global replacement (all occurrences)*

echo "test test test" | sed 's/test/TEST/g'

*# Output: TEST TEST TEST*

*# First occurrence only (default)*

echo "test test test" | sed 's/test/TEST/'

*# Output: TEST test test*

*# Nth occurrence*

echo "test test test" | sed 's/test/TEST/2'

*# Output: test TEST test*

*# Case-insensitive*

echo "Test test TEST" | sed 's/test/FOUND/gi'

*# Output: FOUND FOUND FOUND*

*# Using different delimiters (useful for paths)*

echo "/home/user/documents" | sed 's#/home#/Users#'

*# Output: /Users/user/documents*

**2. Deletion (d) - The Eraser**

bash

*# Delete lines matching pattern*

echo -e "keep this\\ndelete this\\nkeep this too" | sed '/delete/d'

*# Output: keep this*

*# keep this too*

*# Delete line numbers*

seq 1 10 | sed '5d'

*# Deletes line 5*

*# Delete range*

seq 1 10 | sed '3,7d'

*# Deletes lines 3-7*

*# Delete from pattern to end*

echo -e "start\\nkeep1\\nSTOP\\ndelete1\\ndelete2" | sed '/STOP/,$d'

*# Deletes from STOP to end*

**3. Insertion and Appending (i, a) - The Adder**

bash

*# Insert before line*

echo -e "line1\\nline2" | sed '2i\\INSERTED'

*# Output: line1*

*# INSERTED*

*# line2*

*# Append after line*

echo -e "line1\\nline2" | sed '1a\\APPENDED'

*# Output: line1*

*# APPENDED*

*# line2*

*# Insert at pattern*

echo -e "before\\ntarget\\nafter" | sed '/target/i\\### MARKER ###'

**4. Change (c) - The Replacer**

bash

*# Replace entire line*

echo -e "old line\\nkeep this\\nold line" | sed '/old/c\\new line'

*# Output: new line*

*# keep this*

*# new line*

*# Replace line number*

seq 1 5 | sed '3c\\THREE'

*# Line 3 becomes: THREE*

**Advanced sed Techniques**

**1. Multi-line Operations and Hold Space**

bash

*# The hold space - sed's clipboard*

*# Pattern space = current workspace*

*# Hold space = storage area*

*# Swap lines*

echo -e "first\\nsecond" | sed -n '1h;2g;2p;1g;1p'

*# Output: second*

*# first*

*# Join lines*

echo -e "Name: John\\nAge: 30\\nCity: NYC" | sed 'N;s/\\n/ | /'

*# Output: Name: John | Age: 30*

*# City: NYC*

**2. Advanced Pattern Matching**

bash

*# Address ranges with patterns*

cat << 'EOF' > config.txt

# Start Database Config

db\_host=10.0.1.50

db\_port=5432

db\_name=production

# End Database Config

# Start Cache Config

cache\_host=10.0.1.60

EOF

*# Modify only within database section*

sed '/^# Start Database/,/^# End Database/s/10.0.1.50/10.0.2.100/' config.txt

*# Multiple operations with -e*

sed -e 's/old/new/g' -e '/^#/d' -e '/^$/d' file.txt

*# Replace, delete comments, delete empty lines*

**3. Backreferences and Captures**

bash

*# Rearranging data*

echo "Lastname, Firstname" | sed 's/\\(.\*\\), \\(.\*\\)/\\2 \\1/'

*# Output: Firstname Lastname*

*# Extracting and reformatting*

echo "user@domain.com" | sed 's/\\(.\*\\)@\\(.\*\\)/Username: \\1, Domain: \\2/'

*# Output: Username: user, Domain: domain.com*

*# Adding quotes around values*

echo "key=value" | sed 's/\\(.\*\\)=\\(.\*\\)/\\1="\\2"/'

*# Output: key="value"*

**Part 3: awk - The Data Processor**

**The Structured Data Problem**

**Scenario:** "Your web server's access.log has 10 million entries. Management wants to know:

* Top 10 IP addresses by request count
* Average response time per hour
* Total bandwidth consumed per user
* Which API endpoints are slowest
* Error rate trends throughout the day"

"In Excel, this would crash. In a database, you'd need to import first. But with awk?"

**Understanding awk: The Spreadsheet in Terminal**

**Analogy: The Invisible Spreadsheet** "awk treats every text file like a spreadsheet where:

* Each line is a row
* Spaces/tabs create columns
* You can reference columns like $1, $2, $3
* You can perform calculations, aggregations, and transformations
* But it processes millions of rows without loading them into memory!"

**awk Fundamentals**

**Basic Structure**

bash

*# awk 'pattern { action }' file*

*# pattern = when to run (optional)*

*# action = what to do*

*# Print entire file (no pattern = always run)*

awk '{ print }' file.txt

*# Print lines matching pattern*

awk '/ERROR/ { print }' log.txt

*# The awk program structure:*

*# BEGIN { run once before processing }*

*# pattern { run for matching lines }*

*# END { run once after processing }*

**Field Separation Magic**

bash

*# Default: space/tab separated*

echo "John Doe 30 Engineer" | awk '{ print $1, $4 }'

*# Output: John Engineer*

*# Custom field separator with -F*

echo "name:john:age:30" | awk -F: '{ print $2, $4 }'

*# Output: john 30*

*# Multiple character separator*

echo "name==john||age==30" | awk -F'[|]+' '{ print $1, $2 }'

*# Output: name==john age==30*

**Built-in Variables - Your Swiss Army Knife**

bash

*# NF = Number of Fields*

echo "one two three" | awk '{ print NF }'

*# Output: 3*

*# NR = Number of Records (line number)*

seq 1 5 | awk '{ print "Line " NR ": " $0 }'

*# Output: Line 1: 1, Line 2: 2, etc.*

*# $0 = Entire line*

*# $1, $2, ... = Individual fields*

*# $NF = Last field*

echo "first middle last" | awk '{ print $NF }'

*# Output: last*

**Essential awk Operations**

**1. Pattern Matching and Filtering**

bash

*# Numeric comparisons*

df -h | awk '$5 > 80 { print $1 " is " $5 " full" }'

*# Shows filesystems over 80% full*

*# String matching*

ps aux | awk '$11 ~ /python/ { print $2, $11 }'

*# Shows PIDs of Python processes*

*# Multiple conditions*

awk '$3 > 100 && $4 == "ERROR"' logfile.txt

*# Lines where field 3 > 100 AND field 4 is ERROR*

*# Regex patterns*

awk '/^[0-9]/' file.txt

*# Lines starting with a number*

**2. Calculations and Aggregations**

bash

*# Sum a column*

awk '{ sum += $3 } END { print "Total:", sum }' data.txt

*# Average calculation*

awk '{ sum += $3; count++ } END { print "Avg:", sum/count }' data.txt

*# Running calculations*

echo -e "10\\n20\\n30" | awk '{ sum += $1; print "Running total:", sum }'

*# Min/Max tracking*

awk 'BEGIN { max = 0 } $3 > max { max = $3 } END { print "Max:", max }' data.txt

**3. Formatted Output**

bash

*# Printf for formatted output*

ps aux | awk '{ printf "%-10s %8s %s\\n", $1, $2, $11 }' | head -5

*# Formatted process list*

*# Creating reports*

df -h | awk 'BEGIN {

print "====== Disk Usage Report ======"

}

/^\\/dev/ {

printf "%-20s %5s %5s\\n", $1, $5, $6

}

END {

print "=============================="

}'

**Advanced awk Programming**

**1. Arrays and Associative Arrays**

bash

*# Counting occurrences*

awk '{ count[$1]++ } END {

for (ip in count)

print ip, count[ip]

}' access.log

*# Grouping and summing*

echo -e "dept1 100\\ndept2 200\\ndept1 150\\ndept2 50" | \\

awk '{ sum[$1] += $2 } END {

for (dept in sum)

print dept ":", sum[dept]

}'

*# Multi-dimensional arrays*

awk '{ traffic[$1][$2] += $3 } END {

for (source in traffic) {

for (dest in traffic[source]) {

print source, "->", dest, ":", traffic[source][dest]

}

}

}' network\_flow.txt

**2. Functions and Control Flow**

bash

*# User-defined functions*

awk '

function convert\_bytes(bytes) {

if (bytes >= 1073741824)

return bytes/1073741824 " GB"

else if (bytes >= 1048576)

return bytes/1048576 " MB"

else if (bytes >= 1024)

return bytes/1024 " KB"

else

return bytes " B"

}

{ print $1, convert\_bytes($2) }

' file\_sizes.txt

*# Control structures*

awk '{

if ($3 > 1000)

status = "HIGH"

else if ($3 > 500)

status = "MEDIUM"

else

status = "LOW"

print $1, status

}' metrics.txt

**3. Multiple File Processing**

bash

*# Process multiple files with different logic*

awk '

FNR == 1 { file\_num++ } # FNR resets for each file

file\_num == 1 { users[$1] = $2 } # First file: load users

file\_num == 2 && $1 in users { print $1, users[$1], $2 } # Second file: join

' users.txt transactions.txt

*# Compare files*

awk 'FNR==NR { a[$0]; next } !($0 in a)' file1 file2

*# Lines in file2 not in file1*

**Part 4: Text Comparison Tools**

**The Configuration Drift Problem**

**Scenario:** "You manage 20 web servers that should be identical. But users report that server-15 behaves differently. You need to:

* Compare configuration files between working and problematic servers
* Find what changed in the latest deployment
* Identify which servers have drifted from the baseline
* Merge configurations from development to production"

**diff - The Change Detective**

**Understanding diff Output**

bash

*# Basic diff*

echo -e "line1\\nline2\\nline3" > file1.txt

echo -e "line1\\nmodified2\\nline3\\nnewline4" > file2.txt

diff file1.txt file2.txt

*# Output explanation:*

*# 2c2 = line 2 changed to line 2*

*# < line2 = removed from file1*

*# ---*

*# > modified2 = added in file2*

*# 3a4 = after line 3, added line 4*

**diff Formats for Different Needs**

bash

*# Unified format (most readable)*

diff -u file1.txt file2.txt

*# Shows context with + and - indicators*

*# Context format*

diff -c file1.txt file2.txt

*# Shows surrounding lines for context*

*# Side-by-side comparison*

diff -y file1.txt file2.txt

*# Visual comparison in columns*

*# Brief mode (just report if files differ)*

diff -q file1.txt file2.txt

*# Output: Files file1.txt and file2.txt differ*

**Advanced diff Operations**

bash

*# Ignore whitespace changes*

diff -w file1.txt file2.txt

*# Ignore case*

diff -i file1.txt file2.txt

*# Recursive directory comparison*

diff -r dir1/ dir2/

*# Exclude files*

diff -r --exclude="\*.log" --exclude="\*.tmp" dir1/ dir2/

*# Generate patch file*

diff -u original.conf new.conf > changes.patch

*# Apply patch*

patch original.conf < changes.patch

**comm - The Set Operator**

bash

*# comm shows three columns:*

*# Column 1: Lines only in file1*

*# Column 2: Lines only in file2*

*# Column 3: Lines in both files*

*# Prepare sorted files (comm requires sorted input)*

sort file1.txt > sorted1.txt

sort file2.txt > sorted2.txt

comm sorted1.txt sorted2.txt

*# Show only unique to file1*

comm -23 sorted1.txt sorted2.txt

*# Show only unique to file2*

comm -13 sorted1.txt sorted2.txt

*# Show only common lines*

comm -12 sorted1.txt sorted2.txt

**cmp - The Binary Detective**

bash

*# Byte-by-byte comparison*

cmp file1.bin file2.bin

*# Output: file1.bin file2.bin differ: byte 25, line 2*

*# Silent mode for scripts*

cmp -s file1 file2

echo $? *# 0 if identical, 1 if different*

*# Show all differences*

cmp -l file1 file2

*# Shows byte number and differing bytes in octal*

**Part 5: Advanced Text Processing Techniques**

**Combining Tools: The Pipeline Symphony**

**The Power of Composition** "Like instruments in an orchestra, our text processing tools create magic when combined:"

bash

*# Real-world: Analyze failed SSH attempts*

grep "Failed password" /var/log/auth.log | \\

awk '{print $11}' | \\

sort | uniq -c | sort -rn | \\

head -10 | \\

while read count ip; do

echo "$ip: $count attempts"

whois $ip | grep -i country | head -1

done

**Performance Optimization**

**1. Choosing the Right Tool**

bash

*# Task: Count lines containing "ERROR"*

*# Slow (sed)*

time sed -n '/ERROR/p' huge.log | wc -l

*# Faster (awk)*

time awk '/ERROR/ {count++} END {print count}' huge.log

*# Fastest (grep)*

time grep -c ERROR huge.log

*# For simple pattern matching, grep > awk > sed*

*# For complex processing, awk > sed > multiple greps*

**2. Processing Large Files Efficiently**

bash

*# Parallel processing with GNU parallel*

cat huge.log | parallel --pipe grep ERROR | wc -l

*# Stream processing without loading entire file*

*# Good: Process line by line*

awk '{process}' huge.log

*# Bad: Load entire file*

awk '{lines[NR]=$0} END {process all}' huge.log

*# Use head/tail for sampling*

head -1000 huge.log | awk '{your test code}'

Day 6

**Class 6: Fundamentals of Shell Scripting**

**Opening Hook & Problem Statement**

**The 3 AM Crisis: When Repetition Becomes Your Enemy**

**Instructor:** "Let me tell you about Sarah, a junior DevOps engineer at a fast-growing startup. Every morning at 6 AM, before the team arrives, Sarah had to:

1. Check if all 15 microservices are running
2. Clean up log files older than 7 days from 20 different directories
3. Backup 5 databases to specific locations
4. Generate a health report and email it to the team
5. Restart any failed services

This took Sarah 2 hours every single morning. Manual work. Repetitive clicking. Copy-pasting commands.

One night, Sarah's phone rings at 3 AM. The CEO calls: 'We're launching in Japan tomorrow. Can you set up the same morning routine for our 10 new Tokyo servers?'

Sarah realizes: 10 servers × 2 hours = 20 hours of work. Physically impossible before launch.

**The Breaking Point:** Sarah had two choices:

1. Admit defeat and delay the launch
2. Write ONE shell script that does everything automatically

By 5 AM, Sarah had written a 50-line shell script. It now runs on all servers automatically. What took 2 hours now takes 2 minutes. Sarah sleeps peacefully while her script works.

**Question to Students:** "How many of you have done the same task more than 3 times? Copied the same commands repeatedly? That's exactly when you need shell scripting!"

**Why Shell Scripting Changes Everything**

**The Manual vs Automated Reality:**

Think of it like this:

* **Without Shell Scripts:** You're a chef who chops every vegetable by hand, measures every ingredient individually, watches every pot constantly
* **With Shell Scripts:** You're a chef with a food processor, preset timers, and sous chefs. You write the recipe once, and it executes perfectly every time

**Real DevOps Statistics:**

* Average DevOps engineer saves 15 hours/week through automation
* A good shell script can replace work of 5 manual operators
* Netflix deploys code 1000+ times per day - impossible without scripts
* One bug in production costs $5000/hour on average - scripts prevent human errors

**Part 1: Shell Script Structure - Building Your First Robot**

**The Shebang (#!): Your Script's Identity Card**

**Analogy:** "The shebang is like the 'From:' field in a letter. It tells the system WHO should read and execute this script."

bash

#!/bin/bash

*# This script will be executed by bash*

*#!/usr/bin/env python3*

*# This script will be executed by python3*

*#!/bin/sh*

*# This script will be executed by the default shell*

**The Shebang Deep Dive:**

When you run ./myscript.sh, here's what REALLY happens:

1. **Kernel reads first 2 bytes:** Sees #! (magic number)
2. **Kernel reads the interpreter path:** /bin/bash
3. **Kernel executes:** /bin/bash myscript.sh

**Live Demonstration:**

bash

*# Let's prove the shebang matters*

cat > test\_bash.sh << 'EOF'

#!/bin/bash

echo "I am: $BASH\_VERSION"

EOF

cat > test\_sh.sh << 'EOF'

#!/bin/sh

echo "I am: $0"

EOF

chmod +x test\_bash.sh test\_sh.sh

./test\_bash.sh

./test\_sh.sh

**Why This Matters:**

* Wrong shebang = script fails mysteriously
* Bash has features that sh doesn't have
* Python scripts need Python shebang
* Portability across different systems

**Your First Production-Grade Script Structure**

bash

#!/bin/bash

*#*

*# Script: health\_checker.sh*

*# Purpose: Monitor system health and alert on issues*

*# Author: DevOps Team*

*# Date: 2024-03-15*

*# Version: 1.0*

*#*

*# Exit on any error (we'll explain this later)*

set -e

*# Global variables (configuration)*

LOG\_DIR="/var/log/health"

ALERT\_EMAIL="team@company.com"

THRESHOLD\_CPU=80

THRESHOLD\_MEMORY=90

*# Functions (we'll cover these in detail)*

check\_cpu() {

echo "Checking CPU usage..."

*# Function code here*

}

check\_memory() {

echo "Checking memory usage..."

*# Function code here*

}

*# Main execution*

main() {

echo "Starting health check at $(date)"

check\_cpu

check\_memory

echo "Health check completed"

}

*# Run the main function*

main "$@"

**Key Structure Elements:**

1. **Shebang** - Identifies interpreter
2. **Header comments** - Documentation
3. **Safety settings** - Error handling
4. **Global variables** - Configuration
5. **Functions** - Modular code
6. **Main function** - Entry point
7. **Execution** - Start the script

**Part 2: Variables and Arrays - Your Script's Memory**

**Variables: The Building Blocks**

**The Variable Lifecycle:**

bash

*# 1. Declaration and Assignment (no spaces around =)*

name="DevOps"

port=8080

is\_running=true

*# 2. Using Variables ($ to access value)*

echo "Hello $name"

echo "Service running on port $port"

*# 3. Variable Manipulation*

name="Super $name" *# Now "Super DevOps"*

port=$((port + 1)) *# Now 8081*

**Common Variable Mistakes (and How to Fix Them):**

bash

*# WRONG: Spaces around =*

name = "John" *# Error: command 'name' not found*

*# RIGHT: No spaces*

name="John"

*# WRONG: Not quoting with spaces*

file=my file.txt *# Error: 'file.txt' command not found*

*# RIGHT: Quote when spaces present*

file="my file.txt"

*# WRONG: Using without $*

echo name *# Prints: name*

*# RIGHT: Use $ to get value*

echo $name *# Prints: John*

**Variable Types and Scopes**

bash

*# Global variable (available everywhere in script)*

GLOBAL\_CONFIG="/etc/myapp/config"

function demo() {

*# Local variable (only in this function)*

local temp\_file="/tmp/process.tmp"

*# Modifying global*

GLOBAL\_CONFIG="/new/path" *# Changes globally*

}

*# Environment variable (available to child processes)*

export DATABASE\_URL="postgresql://localhost/mydb"

*# Read-only variable (constant)*

readonly API\_VERSION="v2"

API\_VERSION="v3" *# Error: readonly variable*

**Arrays: When One Variable Isn't Enough**

**Analogy:** "Variables are like boxes that hold one item. Arrays are like shelves with multiple compartments."

bash

*# Creating Arrays*

servers=("web01" "web02" "db01" "cache01")

ports=(8080 8081 3306 6379)

status=(true false true true)

*# Accessing Array Elements*

echo ${servers[0]} *# First element: web01*

echo ${servers[2]} *# Third element: db01*

echo ${servers[@]} *# All elements*

echo ${#servers[@]} *# Number of elements: 4*

*# Adding to Arrays*

servers+=("web03") *# Add one element*

servers+=(web04 web05) *# Add multiple elements*

*# Practical Example: Checking Multiple Services*

services=("nginx" "mysql" "redis" "docker")

for service in "${services[@]}"; do

if systemctl is-active --quiet "$service"; then

echo "✓ $service is running"

else

echo "✗ $service is down - restarting..."

systemctl start "$service"

fi

done

**Parameter Expansion: The Swiss Army Knife**

**Basic Expansions:**

bash

name="DevOps Engineer"

*# Length*

echo ${#name} *# 15 characters*

*# Substring*

echo ${name:0:6} *# "DevOps" (from position 0, length 6)*

echo ${name:7} *# "Engineer" (from position 7 to end)*

*# Replace*

echo ${name/Engineer/Master} *# "DevOps Master"*

echo ${name//e/E} *# "DEvOps EnginEEr" (replace all)*

*# Remove patterns*

file="document.backup.tar.gz"

echo ${file%.gz} *# "document.backup.tar" (remove from end)*

echo ${file%.\*} *# "document.backup.tar" (remove last extension)*

echo ${file%%.\*} *# "document" (remove all extensions)*

echo ${file#\*.} *# "backup.tar.gz" (remove from beginning)*

**Advanced: Default Values and Validation**

bash

*# Use default if variable is unset or empty*

echo ${USERNAME:-"anonymous"} *# Use "anonymous" if USERNAME not set*

*# Set and use default*

: ${CONFIG\_FILE:="/etc/default.conf"} *# Set if not already set*

*# Exit if variable not set*

DB\_PASSWORD=${DB\_PASSWORD:?Error: DB\_PASSWORD must be set}

*# Use alternative value if set*

echo ${DEBUG:+"Debug mode is ON"} *# Print only if DEBUG is set*

**Real-World Example: Smart Configuration**

bash

#!/bin/bash

*# Smart configuration with defaults and validation*

*# Environment with defaults*

ENVIRONMENT=${ENVIRONMENT:-"development"}

PORT=${PORT:-8080}

LOG\_LEVEL=${LOG\_LEVEL:-"INFO"}

*# Validate required variables*

DATABASE\_URL=${DATABASE\_URL:?Error: DATABASE\_URL is required}

API\_KEY=${API\_KEY:?Error: API\_KEY is required}

*# Conditional configuration based on environment*

if [[ "$ENVIRONMENT" == "production" ]]; then

LOG\_FILE=${LOG\_FILE:-"/var/log/app.log"}

WORKERS=${WORKERS:-4}

else

LOG\_FILE=${LOG\_FILE:-"./app.log"}

WORKERS=${WORKERS:-1}

fi

echo "Starting application:"

echo " Environment: $ENVIRONMENT"

echo " Port: $PORT"

echo " Workers: $WORKERS"

echo " Log Level: $LOG\_LEVEL"

echo " Log File: $LOG\_FILE"

**Part 3: Control Structures - Your Script's Brain**

**If/Else: Making Decisions**

**The Decision Tree:**

bash

*# Basic if/else structure*

if [ condition ]; then

*# Do something*

elif [ another\_condition ]; then

*# Do something else*

else

*# Default action*

fi

**Test Conditions: The Building Blocks**

bash

*# File/Directory Tests*

if [ -f "/etc/passwd" ]; then

echo "File exists"

fi

if [ -d "/home/user" ]; then

echo "Directory exists"

fi

if [ -r "$file" ]; then

echo "File is readable"

fi

if [ -w "$file" ]; then

echo "File is writable"

fi

if [ -x "$script" ]; then

echo "File is executable"

fi

*# String Comparisons*

if [ "$name" = "admin" ]; then

echo "Admin user"

fi

if [ -z "$variable" ]; then

echo "Variable is empty"

fi

if [ -n "$variable" ]; then

echo "Variable is not empty"

fi

*# Numeric Comparisons*

if [ $age -gt 18 ]; then

echo "Adult"

fi

if [ $score -le 100 ]; then

echo "Valid score"

fi

*# Modern [[ ]] syntax (more powerful)*

if [[ $email =~ ^[a-z]+@[a-z]+\\.[a-z]+$ ]]; then

echo "Valid email format"

fi

if [[ $file == \*.log ]]; then

echo "This is a log file"

fi

**Real-World Example: Intelligent Backup Script**

bash

#!/bin/bash

backup\_database() {

local db\_name=$1

local backup\_dir="/backup/$(date +%Y%m%d)"

*# Create backup directory if it doesn't exist*

if [ ! -d "$backup\_dir" ]; then

echo "Creating backup directory: $backup\_dir"

mkdir -p "$backup\_dir"

fi

*# Check disk space (need at least 1GB)*

available\_space=$(df /backup | awk 'NR==2 {print $4}')

if [ $available\_space -lt 1048576 ]; then

echo "ERROR: Not enough disk space for backup"

return 1

fi

*# Check if database is accessible*

if mysqladmin ping -h localhost &>/dev/null; then

echo "Database is running, starting backup..."

*# Perform backup*

if mysqldump "$db\_name" > "$backup\_dir/${db\_name}.sql"; then

echo "✓ Backup successful"

*# Compress if larger than 100MB*

backup\_size=$(stat -f%z "$backup\_dir/${db\_name}.sql" 2>/dev/null || stat -c%s "$backup\_dir/${db\_name}.sql")

if [ $backup\_size -gt 104857600 ]; then

echo "Compressing large backup..."

gzip "$backup\_dir/${db\_name}.sql"

fi

else

echo "✗ Backup failed"

return 1

fi

else

echo "ERROR: Database is not running"

return 1

fi

}

*# Run backup*

backup\_database "production\_db"

**Case Statements: The Elegant Switch**

**When If/Else Becomes Messy:**

bash

*# Instead of this:*

if [ "$1" = "start" ]; then

start\_service

elif [ "$1" = "stop" ]; then

stop\_service

elif [ "$1" = "restart" ]; then

restart\_service

elif [ "$1" = "status" ]; then

check\_status

else

show\_help

fi

*# Use this elegant case statement:*

case "$1" in

start)

start\_service

;;

stop)

stop\_service

;;

restart)

stop\_service

start\_service

;;

status)

check\_status

;;

\*)

show\_help

;;

esac

**Advanced Pattern Matching:**

bash

#!/bin/bash

*# File processor based on extension*

process\_file() {

local file="$1"

case "$file" in

\*.jpg|\*.jpeg|\*.png)

echo "Processing image: $file"

convert "$file" -resize 800x600 "thumb\_$file"

;;

\*.mp4|\*.avi|\*.mov)

echo "Processing video: $file"

ffmpeg -i "$file" -codec copy "compressed\_$file"

;;

\*.log)

echo "Archiving log: $file"

gzip "$file"

;;

\*.tar.gz|\*.tgz)

echo "Extracting archive: $file"

tar -xzf "$file"

;;

\*.txt|\*.md)

echo "Counting words in: $file"

wc -w "$file"

;;

\*)

echo "Unknown file type: $file"

;;

esac

}

*# Process all files in current directory*

for file in \*; do

process\_file "$file"

done

**Loops: Automation at Scale**

**For Loops: The Workhorse**

bash

*# Loop through list*

for server in web01 web02 db01; do

echo "Checking $server..."

ping -c 1 "$server" &>/dev/null && echo "$server is up" || echo "$server is down"

done

*# Loop through array*

services=("nginx" "mysql" "redis")

for service in "${services[@]}"; do

systemctl status "$service"

done

*# Loop through numbers*

for i in {1..10}; do

echo "Creating user$i"

useradd "user$i"

done

*# C-style for loop*

for ((i=0; i<5; i++)); do

echo "Iteration $i"

done

*# Loop through command output*

for file in $(find /logs -name "\*.log" -size +100M); do

echo "Compressing large log: $file"

gzip "$file"

done

**While Loops: Conditional Repetition**

bash

*# While condition is true*

counter=0

while [ $counter -lt 10 ]; do

echo "Count: $counter"

((counter++))

done

*# Reading file line by line*

while IFS= read -r line; do

echo "Processing: $line"

*# Process each line*

done < servers.txt

*# Infinite loop with break condition*

while true; do

if ping -c 1 google.com &>/dev/null; then

echo "Internet is up!"

break

else

echo "Waiting for internet..."

sleep 5

fi

done

*# Menu system*

while true; do

echo "1) Start services"

echo "2) Stop services"

echo "3) Check status"

echo "4) Exit"

read -p "Choose option: " choice

case $choice in

1) start\_all ;;

2) stop\_all ;;

3) status\_all ;;

4) break ;;

\*) echo "Invalid option" ;;

esac

done

**Until Loops: The Persistent Trier**

bash

*# Keep trying until successful*

until mysqladmin ping &>/dev/null; do

echo "Waiting for MySQL to start..."

sleep 2

done

echo "MySQL is ready!"

*# Retry mechanism*

attempts=0

until [ $attempts -ge 5 ]; do

if curl -s http://api.example.com/health; then

echo "API is responding"

break

fi

attempts=$((attempts + 1))

echo "Attempt $attempts failed, retrying..."

sleep 5

done

**Part 4: Functions - Write Once, Use Everywhere**

**Function Basics: Your Code Building Blocks**

**Analogy:** "Functions are like recipes in a cookbook. Write the recipe once, then anyone can follow it by just calling its name."

bash

*# Function definition*

function say\_hello() {

echo "Hello, DevOps!"

}

*# Alternative syntax*

say\_goodbye() {

echo "Goodbye!"

}

*# Calling functions*

say\_hello

say\_goodbye

**Functions with Parameters**

bash

*# Function with parameters*

greet\_user() {

local name=$1

local role=$2

echo "Welcome $name, you are logged in as $role"

}

*# Calling with arguments*

greet\_user "Alice" "Admin"

greet\_user "Bob" "Developer"

*# Function with all parameters*

process\_all() {

echo "Processing $# items"

for item in "$@"; do

echo " - $item"

done

}

process\_all file1.txt file2.txt file3.txt

**Return Values and Exit Codes**

bash

*# Functions can return exit codes (0-255)*

validate\_email() {

local email=$1

if [[ $email =~ ^[a-zA-Z0-9.\_%+-]+@[a-zA-Z0-9.-]+\\.[a-zA-Z]{2,}$ ]]; then

return 0 *# Success*

else

return 1 *# Failure*

fi

}

*# Using the return value*

if validate\_email "user@example.com"; then

echo "Valid email"

else

echo "Invalid email"

fi

*# Returning data (using echo and command substitution)*

calculate\_percentage() {

local value=$1

local total=$2

local percentage=$(( (value \* 100) / total ))

echo $percentage

}

*# Capture the output*

result=$(calculate\_percentage 25 100)

echo "Percentage: $result%"

**Real-World Function Library**

bash

#!/bin/bash

*# lib/utils.sh - Reusable function library*

*# Logging functions*

log\_info() {

echo "[$(date '+%Y-%m-%d %H:%M:%S')] INFO: $\*" | tee -a "$LOG\_FILE"

}

log\_error() {

echo "[$(date '+%Y-%m-%d %H:%M:%S')] ERROR: $\*" >&2 | tee -a "$LOG\_FILE"

}

log\_success() {

echo "[$(date '+%Y-%m-%d %H:%M:%S')] ✓ SUCCESS: $\*" | tee -a "$LOG\_FILE"

}

*# Check if service is running*

is\_service\_running() {

local service=$1

systemctl is-active --quiet "$service"

}

*# Restart service with retry*

restart\_service\_safe() {

local service=$1

local max\_attempts=3

local attempt=1

while [ $attempt -le $max\_attempts ]; do

log\_info "Attempting to restart $service (attempt $attempt/$max\_attempts)"

systemctl restart "$service"

sleep 2

if is\_service\_running "$service"; then

log\_success "$service restarted successfully"

return 0

fi

((attempt++))

done

log\_error "Failed to restart $service after $max\_attempts attempts"

return 1

}

*# Check disk usage*

check\_disk\_usage() {

local threshold=${1:-90} *# Default 90%*

local alert\_sent=false

while IFS= read -r line; do

usage=$(echo "$line" | awk '{print $5}' | sed 's/%//')

partition=$(echo "$line" | awk '{print $6}')

if [ "$usage" -gt "$threshold" ]; then

log\_error "High disk usage on $partition: ${usage}%"

alert\_sent=true

fi

done < <(df -h | grep -v "^Filesystem")

if [ "$alert\_sent" = false ]; then

log\_info "All disk partitions below ${threshold}% threshold"

fi

}

*# Backup with rotation*

backup\_with\_rotation() {

local source\_dir=$1

local backup\_dir=$2

local keep\_days=${3:-7}

*# Create backup*

local backup\_name="backup\_$(date +%Y%m%d\_%H%M%S).tar.gz"

tar -czf "$backup\_dir/$backup\_name" -C "$source\_dir" .

if [ $? -eq 0 ]; then

log\_success "Backup created: $backup\_name"

*# Remove old backups*

find "$backup\_dir" -name "backup\_\*.tar.gz" -mtime +$keep\_days -delete

log\_info "Removed backups older than $keep\_days days"

else

log\_error "Backup failed for $source\_dir"

return 1

fi

}

**Script Modularization**

bash

#!/bin/bash

*# main\_script.sh - Using modular design*

*# Source the library*

source ./lib/utils.sh

source ./lib/database.sh

source ./lib/monitoring.sh

*# Configuration*

readonly SCRIPT\_DIR="$(cd "$(dirname "${BASH\_SOURCE[0]}")" && pwd)"

readonly CONFIG\_FILE="$SCRIPT\_DIR/config.conf"

readonly LOG\_FILE="/var/log/app\_monitor.log"

# Load configuration

load\_config() {

if [ -f "$CONFIG\_FILE" ]; then

source "$CONFIG\_FILE"

log\_info "Configuration loaded from $CONFIG\_FILE"

else

log\_error "Configuration file not found: $CONFIG\_FILE"

exit 1

fi

}

# Main monitoring function

perform\_health\_check() {

log\_info "Starting health check"

# Use functions from libraries

check\_disk\_usage 85

check\_memory\_usage

check\_database\_connectivity

check\_api\_endpoints

log\_info "Health check completed"

}

# Script entry point

main() {

load\_config

case "${1:-}" in

--check)

perform\_health\_check

;;

--backup)

backup\_with\_rotation "$APP\_DIR" "$BACKUP\_DIR" 7

;;

--monitor)

while true; do

perform\_health\_check

sleep 300 # Check every 5 minutes

done

;;

\*)

echo "Usage: $0 {--check|--backup|--monitor}"

exit 1

;;

esac

}

# Run main function with all arguments

main "$@"

**Part 5: Exit Codes and Error Handling - Building Robust Scripts**

**Understanding Exit Codes**

**The Silent Communication:**

"Exit codes are like traffic lights for scripts. Green (0) means go, everything worked. Red (1-255) means stop, something went wrong."

bash

*# Every command returns an exit code*

ls /etc/passwd

echo $? *# 0 - success*

ls /nonexistent

echo $? *# 2 - file not found*

*# Common exit codes*

*# 0 - Success*

*# 1 - General errors*

*# 2 - Misuse of shell commands*

*# 126 - Command cannot execute*

*# 127 - Command not found*

*# 128 - Invalid argument to exit*

*# 130 - Script terminated by Ctrl-C*

**Using Exit Codes in Scripts**

bash

#!/bin/bash

*# Define custom exit codes*

readonly EXIT\_SUCCESS=0

readonly EXIT\_GENERAL\_ERROR=1

readonly EXIT\_FILE\_NOT\_FOUND=2

readonly EXIT\_PERMISSION\_DENIED=3

readonly EXIT\_DEPENDENCY\_MISSING=4

*# Check dependencies*

check\_dependencies() {

local deps=("curl" "jq" "docker")

for dep in "${deps[@]}"; do

if ! command -v "$dep" &>/dev/null; then

echo "Error: Required dependency '$dep' is not installed"

exit $EXIT\_DEPENDENCY\_MISSING

fi

done

}

*# Process file with proper error handling*

process\_config\_file() {

local config\_file=$1

if [ ! -f "$config\_file" ]; then

echo "Error: Configuration file not found: $config\_file"

exit $EXIT\_FILE\_NOT\_FOUND

fi

if [ ! -r "$config\_file" ]; then

echo "Error: Cannot read configuration file: $config\_file"

exit $EXIT\_PERMISSION\_DENIED

fi

*# Process the file*

source "$config\_file" || exit $EXIT\_GENERAL\_ERROR

}

*# Main execution*

main() {

check\_dependencies

process\_config\_file "/etc/myapp/config.conf"

echo "Script completed successfully"

exit $EXIT\_SUCCESS

}

main "$@"

**Error Handling Strategies**

**Strategy 1: Exit on Error (set -e)**

bash

#!/bin/bash

set -e *# Exit immediately if any command fails*

*# This script will stop at the first error*

create\_user() {

useradd newuser *# If this fails, script stops*

mkdir /home/newuser/data *# Won't execute if useradd failed*

chown newuser /home/newuser/data

}

*# Problem: Sometimes we want to handle errors ourselves*

set +e *# Disable exit on error temporarily*

if grep "pattern" file.txt; then

echo "Pattern found"

else

echo "Pattern not found (this is okay)"

fi

set -e *# Re-enable exit on error*

**Strategy 2: Undefined Variable Protection (set -u)**

bash

#!/bin/bash

set -u *# Exit if undefined variable is used*

*# This will cause an error*

echo $UNDEFINED\_VAR *# Script exits here*

*# Safe way to handle potentially undefined variables*

echo ${MAYBE\_UNDEFINED:-"default value"}

**Strategy 3: Pipe Failure Detection (set -o pipefail)**

bash

#!/bin/bash

set -o pipefail *# Fail if any command in a pipeline fails*

*# Without pipefail: returns 0 (success) even though 'false' failed*

false | echo "test"

echo $? *# 0*

*# With pipefail: returns 1 (failure) because 'false' failed*

set -o pipefail

false | echo "test"

echo $? *# 1*

*# Real-world example*

cat /var/log/app.log | grep ERROR | mail -s "Errors found" admin@example.com

*# If any part fails, we know about it*

**Strategy 4: The Ultimate Safety Net**

bash

#!/bin/bash

*# The professional's setup*

set -euo pipefail *# Exit on error, undefined variable, or pipe failure*

IFS=$'\\n\\t' *# Set Internal Field Separator for safety*

*# Add debug mode*

if [[ "${DEBUG:-}" == "true" ]]; then

set -x *# Print commands as they execute*

fi

*# Trap errors*

trap 'echo "Error on line $LINENO"' ERR

*# Your safe script continues...*

**Error Handling with Traps**

bash

#!/bin/bash

*# Cleanup function*

cleanup() {

echo "Cleaning up temporary files..."

rm -f /tmp/process\_\*

echo "Cleanup completed"

}

*# Set trap for script exit*

trap cleanup EXIT

*# Set trap for errors*

trap 'echo "Error occurred on line $LINENO"; cleanup; exit 1' ERR

*# Set trap for Ctrl-C*

trap 'echo "Script interrupted by user"; cleanup; exit 130' INT TERM

*# Main script work*

echo "Creating temporary files..."

touch /tmp/process\_$$\_data.tmp

touch /tmp/process\_$$\_lock.tmp

*# Simulate some work*

sleep 5

*# Files will be cleaned up automatically on exit*

**Professional Error Handling Pattern**

bash

#!/bin/bash

*# Error handling library*

error\_exit() {

echo "ERROR: $1" >&2

exit "${2:-1}"

}

try() {

[[ $- = \*e\* ]]; SAVED\_OPT\_E=$?

set +e

}

catch() {

export exception\_code=$?

(( SAVED\_OPT\_E )) && set +e

return $exception\_code

}

*# Usage example*

deploy\_application() {

local app\_name=$1

local environment=$2

*# Validate inputs*

[[ -z "$app\_name" ]] && error\_exit "Application name is required" 2

[[ -z "$environment" ]] && error\_exit "Environment is required" 2

*# Try risky operation*

try

echo "Deploying $app\_name to $environment"

*# Stop old version*

docker stop "$app\_name" 2>/dev/null

docker rm "$app\_name" 2>/dev/null

*# Deploy new version*

docker run -d --name "$app\_name" "myapp:latest"

catch || {

case $exception\_code in

125)

error\_exit "Docker daemon is not running" 3

;;

126)

error\_exit "Permission denied - need sudo" 4

;;

\*)

error\_exit "Deployment failed with code $exception\_code" 5

;;

esac

}

echo "✓ Deployment successful"

}

**Practical Exercise: Building a Production Monitoring Script**

**The Challenge**

"Your team needs a monitoring script that runs every 5 minutes and:

1. Checks if critical services are running
2. Monitors disk usage
3. Checks memory usage
4. Validates database connectivity
5. Sends alerts if issues are found
6. Logs all activities
7. Auto-recovers when possible"

**Complete Solution**

bash

#!/bin/bash

*#*

*# production\_monitor.sh - Production System Monitor*

*# Purpose: Monitor critical services and system resources*

*# Author: DevOps Team*

*# Version: 1.0*

*#*

*# Safety settings*

set -euo pipefail

IFS=$'\\n\\t'

*# Configuration*

readonly SCRIPT\_NAME=$(basename "$0")

readonly LOG\_DIR="/var/log/monitoring"

readonly LOG\_FILE="$LOG\_DIR/monitor\_$(date +%Y%m%d).log"

readonly ALERT\_EMAIL="devops-team@company.com"

readonly SLACK\_WEBHOOK="https://hooks.slack.com/services/XXX/YYY/ZZZ"

*# Thresholds*

readonly DISK\_THRESHOLD=85

readonly MEMORY\_THRESHOLD=90

readonly CPU\_THRESHOLD=80

*# Critical services to monitor*

readonly CRITICAL\_SERVICES=(

"nginx"

"mysql"

"redis"

"docker"

)

*# Initialize*

initialize() {

*# Create log directory if it doesn't exist*

[[ ! -d "$LOG\_DIR" ]] && mkdir -p "$LOG\_DIR"

*# Check if running as root (for service management)*

if [[ $EUID -ne 0 ]]; then

log\_error "This script must be run as root"

exit 1

fi

log\_info "=== Starting $SCRIPT\_NAME ==="

}

*# Logging functions*

log\_info() {

local message="[$(date '+%Y-%m-%d %H:%M:%S')] [INFO] $\*"

echo "$message" | tee -a "$LOG\_FILE"

}

log\_error() {

local message="[$(date '+%Y-%m-%d %H:%M:%S')] [ERROR] $\*"

echo "$message" >&2 | tee -a "$LOG\_FILE"

}

log\_warning() {

local message="[$(date '+%Y-%m-%d %H:%M:%S')] [WARN] $\*"

echo "$message" | tee -a "$LOG\_FILE"

}

*# Alert functions*

send\_email\_alert() {

local subject=$1

local body=$2

echo "$body" | mail -s "$subject" "$ALERT\_EMAIL" 2>/dev/null || \\

log\_error "Failed to send email alert"

}

send\_slack\_alert() {

local message=$1

curl -X POST -H 'Content-type: application/json' \\

--data "{\\"text\\":\\"🚨 \*\*Alert\*\*: $message\\"}" \\

"$SLACK\_WEBHOOK" 2>/dev/null || \\

log\_error "Failed to send Slack alert"

}

*# Check critical services*

check\_services() {

log\_info "Checking critical services..."

local failed\_services=()

for service in "${CRITICAL\_SERVICES[@]}"; do

if systemctl is-active --quiet "$service"; then

log\_info " ✓ $service is running"

else

log\_error " ✗ $service is not running"

failed\_services+=("$service")

*# Attempt auto-recovery*

log\_info " Attempting to restart $service..."

if systemctl restart "$service" 2>/dev/null; then

sleep 2

if systemctl is-active --quiet "$service"; then

log\_info " ✓ Successfully restarted $service"

send\_slack\_alert "$service was down but has been automatically restarted"

else

log\_error " ✗ Failed to restart $service"

fi

fi

fi

done

if [[ ${#failed\_services[@]} -gt 0 ]]; then

send\_email\_alert "Critical Services Down" \\

"The following services are not running: ${failed\_services[\*]}"

return 1

fi

return 0

}

*# Check disk usage*

check\_disk\_usage() {

log\_info "Checking disk usage..."

local alert\_triggered=false

while IFS= read -r line; do

local usage=$(echo "$line" | awk '{print $5}' | sed 's/%//')

local partition=$(echo "$line" | awk '{print $6}')

if [[ $usage -gt $DISK\_THRESHOLD ]]; then

log\_warning " High disk usage on $partition: ${usage}%"

send\_slack\_alert "High disk usage on $partition: ${usage}% (threshold: ${DISK\_THRESHOLD}%)"

alert\_triggered=true

*# Auto-cleanup old logs if /var/log is affected*

if [[ "$partition" == "/var/log" ]] || [[ "$partition" == "/" ]]; then

log\_info " Cleaning old logs..."

find /var/log -name "\*.log" -mtime +30 -delete 2>/dev/null

find /tmp -type f -mtime +7 -delete 2>/dev/null

fi

else

log\_info " $partition: ${usage}% used"

fi

done < <(df -h | grep -E '^/dev/' | grep -v '/boot')

[[ "$alert\_triggered" == true ]] && return 1 || return 0

}

*# Check memory usage*

check\_memory\_usage() {

log\_info "Checking memory usage..."

local total\_mem=$(free -m | awk 'NR==2{print $2}')

local used\_mem=$(free -m | awk 'NR==2{print $3}')

local usage\_percent=$((used\_mem \* 100 / total\_mem))

log\_info " Memory usage: ${usage\_percent}% (${used\_mem}MB/${total\_mem}MB)"

if [[ $usage\_percent -gt $MEMORY\_THRESHOLD ]]; then

log\_warning " High memory usage: ${usage\_percent}%"

*# Find top memory consumers*

local top\_processes=$(ps aux --sort=-%mem | head -6 | tail -5)

send\_slack\_alert "High memory usage: ${usage\_percent}%\\n\\nTop processes:\\n$top\_processes"

return 1

fi

return 0

}

*# Check database connectivity*

check\_database() {

log\_info "Checking database connectivity..."

if mysqladmin ping -h localhost &>/dev/null; then

log\_info " ✓ Database is responding"

*# Check for slow queries*

local slow\_queries=$(mysql -e "SHOW STATUS LIKE 'Slow\_queries';" | awk 'NR==2{print $2}')

if [[ $slow\_queries -gt 100 ]]; then

log\_warning " High number of slow queries: $slow\_queries"

fi

return 0

else

log\_error " ✗ Database is not responding"

send\_email\_alert "Database Connection Failed" \\

"Unable to connect to MySQL database on localhost"

return 1

fi

}

*# Main monitoring function*

perform\_health\_check() {

local status=0

check\_services || status=1

check\_disk\_usage || status=1

check\_memory\_usage || status=1

check\_database || status=1

if [[ $status -eq 0 ]]; then

log\_info "=== All checks passed ✓ ==="

else

log\_error "=== Some checks failed ✗ ==="

fi

return $status

}

*# Cleanup on exit*

cleanup() {

log\_info "=== Monitoring script ended ==="

}

*# Set up signal handlers*

trap cleanup EXIT

trap 'log\_error "Script interrupted"; exit 130' INT TERM

*# Main execution*

main() {

initialize

case "${1:-}" in

--once)

perform\_health\_check

;;

--continuous)

log\_info "Running in continuous mode (Ctrl+C to stop)"

while true; do

perform\_health\_check || true *# Continue even if checks fail*

log\_info "Sleeping for 5 minutes..."

sleep 300

done

;;

\*)

echo "Usage: $0 {--once|--continuous}"

echo " --once Run checks once and exit"

echo " --continuous Run checks every 5 minutes"

exit 1

;;

esac

}

*# Run the script*

main "$@"

**Key Takeaways and Best Practices**

**The Golden Rules of Shell Scripting**

1. **Always use safety settings:**

bash

set -euo pipefail

IFS=

\\n\\t'

1. **Quote your variables:**

bash

*# Bad: $variable*

*# Good: "$variable"*

1. **Use local in functions:**

bash

function my\_func() {

local var="value"

}

1. **Check exit codes:**

bash

if command; then

echo "Success"

else

echo "Failed with code $?"

fi

1. **Provide helpful error messages:**

bash

[[ -f "$file" ]] || error\_exit "File not found: $file"

1. **Use meaningful variable names:**

bash

*# Bad: x=5*

*# Good: max\_retries=5*

1. **Comment complex logic:**

bash

*# Calculate percentage while handling division by zero*

[[ $total -eq 0 ]] && percentage=0 || percentage=$((value \* 100 / total))

1. **Make scripts idempotent:**
   * Running multiple times should have the same effect
   * Check before creating/deleting
2. **Log everything important:**
   * Actions taken
   * Errors encountered
   * Success confirmations
3. **Test your scripts:**
   * Test with missing files
   * Test with wrong permissions
   * Test with interrupt signals
   * Test with bad input

**The DevOps Mindset**

"Remember: If you do something more than twice, automate it. If it can fail, handle the failure. If it runs in production, log everything. Your future self will thank you!"