DevOps
AIDE501 – M2 – 3 Credits
2026/2027

Course Contents

- **Chapter 1: Cloud Computing**
- **Chapter 2: Containerization**
- **Chapter 3: Shell Scripting**
- **Chapter 4: DevOps**
- **Chapter 5: Orchestration**
- **Chapter 6: GitLab CI/CD**

Chapter 3:

Shell Scripting

Chap.3: Shell Scripting

- Basics of Shell Scripting
- Advanced Scripting Techniques
- YAML, JSON

Introduction to Shell Scripting in DevOps

Shell Scripting

- A shell is a command-line interpreter that provides an interface between the user and the operating system.
- A shell script is a file containing a sequence of commands executed by the shell.

Shell Scripting in DevOps

- DevOps involves automating repetitive tasks to improve software delivery and infrastructure management.
- Shell scripting is commonly used for:
  - Automating server configuration.
  - Managing files and directories.
  - Installing and configuring software.
  - Building and deploying applications.
  - Managing Docker containers and images.
  - Monitoring system resources.
  - Executing automated tests.
  - Integrating commands into CI/CD pipelines.

Shell Scripting in DevOps

- Shell scripts can support almost every stage:

DevOps stage	Shell scripting application
Code	Execute Git commands
Build	Compile applications
Test	Run automated tests
Release	Prepare release packages
Deploy	Deploy applications
Operate	Manage servers and containers
Monitor	Check services and resources

Creating and executing a script

Example: `hello.sh`

```
#!/bin/bash
# My first DevOps script
echo "Welcome to DevOps!"
echo "Starting automation..."
```

Execution:

```
chmod +x hello.sh
./hello.sh
```

Expected output:

```
Welcome to DevOps!
Starting automation...
```

Variables and user input

```
#!/bin/bash
APP_NAME="MyApplication"
VERSION="1.0"

echo "Application: $APP_NAME"
echo "Version: $VERSION"

read -p "Enter the server name: " SERVER
echo "Deploying $APP_NAME on $SERVER"
```

- Environment variables such as `$HOME`, `$PATH`, and `$USER`.
- Set a Temporary Variable (Current Session): `export KEY=value`

Arguments

- DevOps scripts frequently receive parameters from other tools or pipelines.

```
#!/bin/bash
echo "Script name: $0"
echo "Application: $1"
echo "Environment: $2"
echo "Number of arguments: $#"
```

Execution:

```
./deploy.sh myapp production
```

Conditions

Example: checking whether a service is running.

```
#!/bin/bash

SERVICE="docker"

if systemctl is-active --quiet "$SERVICE";
  then
    echo "$SERVICE is running"
else
    echo "$SERVICE is not running"
fi
```

Loops

Example: checking several services.

```
#!/bin/bash

for service in docker ssh cron
do
    if systemctl is-active --quiet "$service";
  then
        echo "$service: RUNNING"
    else
        echo "$service: STOPPED"
    fi
done
```

Functions

- Functions allow us to organize scripts into reusable components.

```
#!/bin/bash
check_service() {
    SERVICE=$1
    if systemctl is-active --quiet "$SERVICE"; then
        echo "$SERVICE is running"
    else
        echo "$SERVICE is stopped"
    fi
}

check_service docker
check_service ssh
```

File and directory management

```
#!/bin/bash

BACKUP_DIR="$HOME/backup"
SOURCE_DIR="$HOME/project"

mkdir -p "$BACKUP_DIR"

cp -r "$SOURCE_DIR" "$BACKUP_DIR/"

echo "Backup completed successfully"
```

Monitoring disk space

```
#!/bin/bash

THRESHOLD=80

USAGE=$(df / | awk 'NR==2 {gsub("%","",$5);
  print $5}')
echo "Disk usage: $USAGE%"
if [ "$USAGE" -ge "$THRESHOLD" ]; then
    echo "WARNING: Disk usage is too high!"
else
    echo "Disk usage is normal"
fi
```

Monitoring disk space

`df /`: Runs the df (disk free) utility to check space on the root directory (/).

`| (pipe)`: Sends the output of df / directly into the awk command.

`awk 'NR==2 {gsub("%","",$5); print $5}'`: Processes the text stream:

`NR==2`: Targets only the second line of output (skipping the header line on NR==1).

`gsub("%","",$5)`: Finds the percent sign (%) in the 5th column (the "Use%" column) and replaces it with an empty string.

`print $5`: Prints the cleaned-up numeric value (e.g., 45 instead of 45%).

Monitoring memory

```
#!/bin/bash


free -h
```

- It displays the total amount of free and used physical memory (RAM) and swap space on your system.
- `-h` flag converts the numbers into readable units like megabytes (MB) or gigabytes (GB)

Automating tasks with Cron

- Cron allows scripts to execute automatically at scheduled times.

Example:

```
crontab -e
```

Schedule a backup every day at 2:00 AM:

```
0 2 * * * /home/user/backup.sh
```

Automating tasks with Cron

```
* * * * * <command to execute>

| | | | |
| | | | └── Day of the week (0-6, Sunday=0)
| | | └──── Month (1-12)
| | └────── Day of the month (1-31)
| └──────── Hour (0-23)
└────────── Minute (0-59)
```

Automating tasks with Cron

Special Characters

`*` — Any value (every minute, hour, day, etc.)

`,` — List of values (e.g., 1,15,30 in the minute field)

`-` — Range of values (e.g., 1-5 for Monday through Friday)

`/` — Step values (e.g., */15 to run every 15 minutes)

Automating tasks with Cron

- Common Crontab Commands

Command	Description
`crontab -e`	Edit or create the current user's crontab file
`crontab -l`	List all scheduled cron jobs for the current user
`crontab -r`	Remove the current user's crontab filecrontab -u
`username -l`	List another user's crontab (requires root/sudo)

Automating Docker operations

Example: `docker-management.sh`

```
#!/bin/bash
echo "Docker Management Script"
echo "1. List containers"
echo "2. List images"
echo "3. Show running containers"
echo "4. Stop all running containers"
read -p "Choose an option: " OPTION
case $OPTION in
    1) docker ps -a ;;
    2) docker images ;;
    3) docker ps ;;
    4)    docker stop $(docker ps -q) ;;
    *)    echo "Invalid option" ;;
esac
```

Building and tagging Docker images

Example: `build-image.sh`

```
#!/bin/bash
APP_NAME="myapp"
VERSION="1.0"
echo "Building Docker image..."
docker build -t "$APP_NAME:$VERSION" .
if [ $? -eq 0 ]; then
    echo "Image built successfully"
else
    echo "Image building failed"
    exit 1
fi
```

Automating container deployment

```
#!/bin/bash
CONTAINER="web-server"
IMAGE="nginx:latest"
PORT="8080"
docker stop "$CONTAINER" 2>/dev/null
docker rm "$CONTAINER" 2>/dev/null
docker run -d \
    --name "$CONTAINER" \
    -p "$PORT:80" \
    "$IMAGE"
echo "Application deployed on port $PORT"
```

Advanced Shell Scripting in DevOps

Advanced Shell Scripting

Advanced Shell Scripting Techniques focus on developing robust, reusable, secure, and automated scripts for managing Linux systems and implementing DevOps workflows:

- Develop modular and reusable Bash scripts.
- Implement advanced error handling and debugging.
- Manage background processes and parallel execution.
- Process logs, JSON, and YAML files.
- Implement secure configuration management.
- Automate Docker deployment operations.
- Develop scripts suitable for CI/CD pipelines.
- Apply security and reliability best practices.

Advanced Functions and Modular Scripting

Functions allow developers and system administrators to organize scripts into reusable components.

Important concepts:

- Local variables using local.
- Function parameters using $1, $2, etc.
- Return codes.
- Code reusability.
- Modular script organization.

Advanced Functions and Modular Scripting

```
#!/bin/bash
log_message() {
    local level="$1"
    local message="$2"
    echo "[$(date '+%Y-%m-%d %H:%M:%S')] [$level] $message"
}
 check_service() {
    local service="$1"
    if systemctl is-active --quiet "$service"; then
        log_message "INFO" "$service is running"
    else
        log_message "ERROR" "$service is not running"
        return 1
    fi
}
 check_service docker
```

Advanced Input and Argument Handling

Shell scripts can accept command-line arguments and options to make them flexible.

DevOps applications:

- Deployment configuration.
- Environment selection.
- Automated server administration.
- CI/CD pipeline parameterization.

Advanced Input and Argument Handling

```
#!/bin/bash
 while getopts "u:p:e:" opt; do
    case "$opt" in
        u) USERNAME="$OPTARG" ;;
        p) PORT="$OPTARG" ;;
        e) ENVIRONMENT="$OPTARG" ;;
        *) echo "Usage: $0 -u username -p port -e environment"
            exit 1
            ;;
    esac
done
echo "Username: $USERNAME"
echo "Port: $PORT"
echo "Environment: $ENVIRONMENT"
```

Example execution: `./deploy.sh -u admin -p 8080 -e production`

Error Handling and Debugging

Reliable scripts must detect errors and stop execution when critical operations fail.

Option	Purpose
`set -e`	Exit on many unhandled command failures
`set -u`	Detect use of unset variables
`set -o pipefail`	Detect failures within pipelines
`set -x`	Display commands during execution

Error Handling and Debugging

```
#!/bin/bash


set -Eeuo pipefail
trap 'echo "Error on line $LINENO"; exit 1' ERR


echo "Starting deployment..."


mkdir -p /tmp/deployment
cp application.conf /tmp/deployment/


echo "Deployment completed successfully"
```

Working with Arrays and Associative Arrays

Arrays allow scripts to manage collections of values.

DevOps applications:

- Managing multiple containers.
- Service configuration.
- Server inventories.
- Deployment environments.

Indexed arrays:

```
#!/bin/bash
SERVICES=("nginx" "docker" "ssh" "mysql")
for service in "${SERVICES[@]}"; do
    echo "Checking $service"
done
```

Working with Arrays and Associative Arrays

Associative arrays:

```
#!/bin/bash


declare -A PORTS
PORTS[frontend]=3000
PORTS[backend]=5000
PORTS[database]=5432

for service in "${!PORTS[@]}"; do
    echo "$service uses port ${PORTS[$service]}"
done
```

Advanced File Processing and Text Manipulation

Shell scripting provides powerful utilities for processing logs, configuration files, and application data.

DevOps applications:

- Log analysis.
- Security monitoring.
- Detecting failed authentication attempts.
- Extracting application errors.
- Generating operational reports.

Advanced File Processing and Text Manipulation

Essential commands:

Command	Purpose
`grep`	Search for patterns
`sed`	Modify text streams
`awk`	Process structured text
`cut`	Extract columns
`sort`	Sort data
`uniq`	Remove repeated adjacent lines
`tr`	Translate characters

Advanced File Processing and Text Manipulation

This script for monitoring failed login attempts.

It extracts and counts IP addresses associated with failed SSH login attempts.

The exact log format can vary between Linux distributions.

```
#!/bin/bash

LOG_FILE="/var/log/auth.log"

if [[ -r "$LOG_FILE" ]]; then
    grep "Failed password" "$LOG_FILE" |
    awk '{print $11}' |
    sort |
    uniq -c |
    sort -nr
else
    echo "Cannot read authentication log"
    exit 1
fi
```

Process Management and Background Execution

Shell scripts can execute multiple tasks simultaneously and manage their processes.

This technique is useful for parallel testing, backups, and deployment operations.

Important concepts:

- `&`: Execute a process in the background.
- `$!`: Obtain the PID of the last background process.
- `wait`: Wait for a process to finish.
- `jobs`: Display active background jobs.
- `kill`: Send signals to processes.

Process Management and Background Execution

```
#!/bin/bash


echo "Starting backup..."
tar -czf backup.tar.gz ./data &


BACKUP_PID=$!
echo "Backup process ID: $BACKUP_PID"


wait "$BACKUP_PID"
echo "Backup completed"
```

Process Management and Background Execution

```
#!/bin/bash
for i in {1..5}; do
    (
        echo "Task $i started"
        sleep 3
        echo "Task $i completed"
    ) &
done
wait
echo "All tasks completed"
```

Signals and Traps

Signals allow scripts to respond to events such as interruption or termination.

Common signals:

Signal	Description
`SIGINT`	Interrupt (Ctrl+C)
`SIGTERM`	Termination request
`SIGKILL`	Force termination
`SIGHUP`	Hangup
`EXIT`	Shell exit trap

Signals and Traps

The trap command is a built-in shell utility in Linux used to catch system signals and special events, executing a specified command or function before the script continues or terminates.

It is most commonly used in shell scripts to perform cleanup tasks, such as deleting temporary files or stopping background processes, even if the script crashes or is interrupted by the user.

DevOps applications:

- Cleaning temporary resources.
- Stopping services gracefully.
- Handling interrupted deployments.
- Managing temporary files.

Signals and Traps

```
#!/bin/bash
TEMP_FILE=$(mktemp)

cleanup() {
    echo "Cleaning temporary files..."
    rm -f "$TEMP_FILE"
}

trap cleanup EXIT
trap 'exit 130' INT

echo "Working..."
sleep 10
```

Advanced Redirection and Process Substitution

Redirection allows scripts to control standard input, output, and error streams.

Operator	Meaning
`>`	Redirect standard output
`>>`	Append standard output
`2>`	Redirect standard error
`&>`	Redirect output and errors
`2>&1`	Redirect errors to standard output
`<`	Redirect input

Advanced Redirection and Process Substitution

Both standard output and errors are saved in the same log file.

```
#!/bin/bash
./deploy.sh > deployment.log 2>&1
```

Process substitution: This compares two files after sorting their contents without creating intermediate files.

```
diff <(sort file1.txt) <(sort file2.txt)
```

Preventing simultaneous execution with flock

This prevents two instances of the same deployment script from executing concurrently.

```
#!/bin/bash
LOCK_FILE="/tmp/deployment.lock"
exec 200>"$LOCK_FILE"
if ! flock -n 200; then
    echo "Another deployment is running"
    exit 1
fi
echo "Deployment started"
sleep 5
echo "Deployment completed"
```

Environment Variables and Configuration Management

Environment variables allow scripts to adapt to different deployment environments.

Environment variables using :

- `export`.
- `.env` configuration files.
- Default values.
- Separating configuration from application code.

Security consideration: Never store passwords, API keys, or cloud credentials directly in scripts or commit them to Git. Use protected CI/CD secrets or an appropriate secrets manager.

Environment Variables and Configuration Management

```
#!/bin/bash
APP_ENV="${APP_ENV:-development}"
APP_PORT="${APP_PORT:-3000}"

echo "Environment: $APP_ENV"
echo "Application port: $APP_PORT"
```

In Execution:

```
APP_ENV=production APP_PORT=8080 ./start.sh
```

Working with JSON and YAML in DevOps

Modern DevOps environments frequently exchange configuration and monitoring data in JSON and YAML formats.

`jq` is a command-line JSON processor. It acts like `sed` or `awk` for structured JSON data, allowing you to parse, filter, format, and transform output directly in the terminal.

Working with JSON and YAML in DevOps

```
#!/bin/bash

RESPONSE='{
  "application": "backend",
  "status": "running",
  "port": 5000
}'

STATUS=$(echo "$RESPONSE" | jq -r '.status')
PORT=$(echo "$RESPONSE" | jq -r '.port')

echo "Status: $STATUS"
echo "Port: $PORT"
```

Using curl to test an API:

- REST API testing.
- Docker health checks.
- Kubernetes-related automation.
- Cloud service integration.
- Monitoring dashboards.

```
#!/bin/bash

URL="http://localhost:5000/health"

HTTP_CODE=$(curl -s -o /tmp/health.json \

    -w "%{http_code}" "$URL")

if [[ "$HTTP_CODE" == "200" ]]; then

    echo "Application is healthy"

else

    echo "Application health check failed: $HTTP_CODE"

    exit 1

fi
```

Shell Script Security Best Practices

Security is particularly important when shell scripts automate infrastructure and deployment processes.

Essential for production DevOps:

- Validate all user inputs.
- Quote variables to prevent word splitting and glob expansion.
- Avoid executing untrusted input with eval.
- Use absolute paths for critical commands when appropriate.
- Avoid hardcoded passwords and tokens.
- Use temporary files securely with mktemp.
- Avoid running scripts as root unless necessary.
- Check the exit status of critical commands.

Shell Script Security Best Practices

Essential for production DevOps:

- Set restrictive file permissions.

```
chmod 700 deploy.sh
```

- Scan scripts using ShellCheck

```
shellcheck deploy.sh
```

`ShellCheck` detects common errors and potentially unsafe coding practices in shell scripts.

Recommended Advanced Tools

Tool	Purpose
`ShellCheck`	Shell script static analysis
`jq`	JSON processing
`yq`	YAML processing
`curl`	HTTP/API requests
`ssh`	Remote server administration
`rsync`	File synchronization
`flock`	Process locking
`cron`	Task scheduling
`git`	Version control
Docker CLI	Container automation