Shell Scripting

Due to 8 November 2025

يعد الغش مخالفة أكاديمية وفقا للوائح والقوانين المعمول بها في جامعة قطر، وقد تصل عقوبة هذه المخالفة في بعض الحالات إلى الفصل النهائي من الجامعة وعلى الطلاب تجنب القيام أو المشاركة في أي عمل يخالف ميثاق النزاهة الأكاديمية وإجراءات الاختبارات المعمول بها في جامعة قطر

Cheating is an academic violation according to Qatar University rules and regulations, and in some cases, it may result in final dismissal from the university. Students should not under any circumstances commit or participate in any cheating attempt or any act that violates student code of conduct.

It is the right of the instructor to test the student's undersetting of the project in any way during the demo and discussion session. So, a project that is 100% working might be may be graded (-100%) due to student not being able to explain a functionality they have implemented.

Project Objectives:

1. **Automated Client–Server Tasks:** Develop scripts on both client and server to exchange files, validate access, and perform basic analysis.
2. **System Monitoring & Logging:** Collect system metrics, record results, and maintain logs with rotation for long-term storage.
3. **Secure File & Permission Management:** Apply correct permissions, compress files, and ensure safe handling of shared data.
4. **Error Handling & Recovery:** Write scripts that handle invalid input, connection issues, and produce meaningful error messages.
5. **Practical Administration:** Automate routine tasks such as connectivity checks, reporting, and log management across multiple VMs.
6. **Implement simple IPC:** Use a pipe between parent and child so the child sends a task/project name, and the parent executes the matching shell script, logging the interaction.

Environment Setup:

- **Virtual Machines:** Use **Ubuntu Server 22.04 LTS** for three VMs:
  - **VM1 (Server):** (4GB RAM, 4 CPUs) Hosts monitoring services, receives tokens and files from clients, maintains logs, and provides administrative scripts.
  - **VM2 (Client):** (2GB RAM, 2 CPU each) Responsibilities: Runs scripts for sending tokens to the server, monitors file system changes, and transfers compressed payloads.
    Client must be able to SSH/SCP to the Server.

Note: [If it is not possible to install it on the same machine, feel free to install it on different physical machines].

Tasks (First Part):

Write a script docs/setup.sh must be provided. This script is responsible for creating the required directory structure and setting up permissions.

```
S25-01/
  server/
    srv_access_accept.sh
    srv_analyze.sh
    srv_stats.sh
    srv_net_health.sh
  client/
    cli_submit.sh
    cli_fetch.sh
    cli_push_metrics.sh
    cli_fix_perms.sh
    cli_watchdog.sh
  docs/
    sample_input.txt
    setup.sh
    README.md
```

- **On Server VM**, the setup script must:
  - Create the directories: ~/tokens, ~/queue, ~/results, ~/logs, ~/archive
  - Create an allowlist file: ~/tokens/allowlist.txt
  - Ensure all shell scripts under S25-01/ are marked executable (chmod +x).
- **On Client VM**, the setup script must:
  - Ensure the S25-01/ repository exists.
  - Create local directories: results/ and tmp/.
  - Ensure all shell scripts under S25-01/ are marked executable (chmod +x).

Server Side Tasks (VM1):

Server-Setup requirement:

- Install MySQL server, configure MySQL server to start automatically on boot. Checks and displays the status of the MySQL service.

1. `srv_access_accept.sh:` Write a script that validates access codes and creates request files for incoming jobs. The server keeps valid codes in `~/tokens/allowlist.txt`

Example:

```
CS405TOKEN
TEAM2025
DEVACCESS
```

- When srv_access_accept.sh runs, it receives:
  - CLIENT_USER (the requester)
  - ACCESS_CODE (to check against the allowlist)
  - FNAME (the filename being submitted)

- If the access code is valid:
  - A unique **Request ID** is generated (e.g., REQ_20251018_140311_0007).
  - A request file is created in ~/queue/ with details:
    - ID, CLIENT_USER, CREATED_UTC, PAYLOAD_FILE, FILENAME

Example:

```
ID=REQ_20251018_140311_0007
CLIENT_USER=qustudent
CREATED_UTC=2025-10-18T14:03:11Z
PAYLOAD_FILE=/home/server/queue/REQ_20251018_140311_0007.payload
FILENAME=sample_input.txt.gz
```

- The script prints:

```
ACCEPT REQ_20251018_140311_0007
```

- If the access code is invalid, the script prints:

```
REJECT invalid_token
```

Explanation of terminology - Payload:

When you run `gzip file.txt` in Linux, the program **compresses the content of `file.txt`** and produces a new file named `file.txt.gz`.

Here's what happens in terms of terminology:

- **Input file**: `file.txt` (the original, uncompressed file).
- **Output file**: `file.txt.gz` (the compressed file).
- **Payload** inside the `.gz`: the compressed version of the *original file's data*.

So, the **payload file** in this context is simply the **original file's data that gets stored (in compressed form) inside the `.gz` archive**.

sample_input.txt – Client VM will send the file to the server VM

Operating Systems lab project Fall 2025. This file is submitted by qustudent for testing. It contains three lines of text total.

The server has to give these statistics:

Lines: 3 Words: 17 Bytes: 119

2. `srv_analyze.sh`: Request processor

Run continuously. When a *.req file and its matching *.payload exist in ~/queue/, analyze the payload and write a result file in ~/results/.

- **Input files (in ~/queue/):**
  - <ID>.req containing: ID, CLIENT_USER, CREATED_UTC, PAYLOAD_FILE, FILENAME
  - <ID>.payload (the uploaded file: plain text **or** gzip-compressed text)
- **Behavior:**
  - Detect payload type:
    - Plain text → analyze directly.
    - Gzip-compressed (.gz) → decompress, then analyze.
  - Compute and write to ~/results/RES_<ID>.txt:
    - RESULT for <ID>
    - CLIENT_USER=<…>
    - LINES=<…>
    - WORDS=<…>
    - BYTES=<…>
    - PROCESSED_UTC=<UTC timestamp>
  - Append a one-line summary to ~/logs/worker.log with:
    - UTC_timestamp, ID=<…>, CLIENT_USER=<…>, DURATION_MS=<…>, PAY LOAD=<path>
  - Remove the processed .req and .payload from ~/queue/.

Example (result file ~/results/RES_REQ_20251018_140311_0007.txt):

```
RESULT for REQ_20251018_140311_0007
CLIENT_USER=qustudent
LINES=3
WORDS=20
BYTES=138
PROCESSED_UTC=2025-10-18T14:03:12Z
```

Example (log entry in ~/logs/worker.log):

```
2025-10-18T14:03:12Z ID=REQ_20251018_140311_0007 CLIENT_USER=qustudent
DURATION_MS=7
PAYLOAD=/home/server/queue/REQ_20251018_140311_0007.payload
```

3. `srv_stats.sh`: Log summary & rotation

Summarize ~/logs/worker.log and rotate old logs.

- **Input:** ~/logs/worker.log (one line per job with DURATION_MS= and CLIENT_USER=).
- **Behavior:**
    - Print a one-line summary to stdout:

```
jobs=<count> avg_time_ms=<avg> top_user=<name-or-none>
```

    - Compress and move logs **older than 7 days** to ~/archive/.
- **Sample Output:**

```
jobs=5 avg_time_ms=6 top_user=qustudent
```

4. `srv_net_health.sh`: Connectivity snapshot

Check reachability and SSH port status of a target host; append result to `~/logs/health.log`.

- **Input (arg):** target hostname (e.g., `client-vm`).
- **Behavior:**
  - Ping the target (1 probe, short timeout) → `ping=OK|FAIL`.
  - Check TCP port **22** on the target → `ssh_port=OPEN|CLOSED`.
  - Append a single line to `~/logs/health.log` with UTC timestamp, target, ping, ssh_port.
- **Sample Log Entry:**

```
2025-10-18T14:10:00Z target=client-vm ping=OK ssh_port=OPEN
```

Client side Tasks (VM2):

Client-Side Scripts

1. `cli_submit.sh`: Submit work

Validate inputs, compress the file, set permissions, request an ID, upload the payload.

- **Inputs (args/env):** server hostname, path to input file, access code, requester name.
- **Behavior:**
  - Create a gzip copy of the input (keep .gz).
  - Change the compressed file mode to **rw for user+group, r for others**.
  - Invoke the server's srv_access_accept.sh remotely with CLIENT_USER, ACCESS_CODE, FNAME=<basename.gz>.
  - On ACCEPT <ID>, upload the compressed file to ~/queue/<ID>.payload on the server.
- **Sample Output:**

```
ACCEPT REQ_20251018_140311_0007
```

2. `cli_fetch.sh`: Retrieve results by ID

Check the job state on the server and download the result if ready.

- **Inputs (args):** server hostname, `<ID>`.
- **Behavior:**
  - If `~/results/RES_<ID>.txt` exists on the server → copy to `S25-01/results/<ID>.txt` and print:
  - `READY results/<ID>.txt`
  - Else if `~/queue/<ID>.req` or `~/queue/<ID>.payload` exists → print:

    - `PROCESSING`
    - Otherwise → print:
    - `MISSING`
- **Sample Output:**

```
READY results/REQ_20251018_140311_0007.txt
```

3. `cli_push_metrics.sh`: Client metrics report (independent)

Collect a small system snapshot and deliver it to the server.

- **Inputs (arg):** server hostname.
- **Behavior:**
    - Create `tmp/metrics_<timestamp>.txt` containing hostname, UTC time, load average, memory usage, and top 5 CPU processes.
    - Copy the file to server `~/results/`.
- **Sample Output:**

```
PUSHED ~/results/metrics_20251018_141200.txt
```

4. `cli_fix_perms.sh`: Permissions hygiene (independent)

Find and remediate overly permissive files.

- **Inputs (arg, optional):** root directory (default: `$HOME`).
- **Behavior:**
    - Find files with mode **777**, change to **700**, append each change to `perm_changes.log`.
    - Print a one-line summary at the end.
- **Sample Output:**

```
fixed:/home/qustudent/tmp/run.sh 777->700
summary: changed=1
```

5. `cli_watchdog.sh`: Local periodic health (independent)

Emit brief health snapshots locally.

- **Inputs (arg, optional):** number of loops `N` (default: 2).
- **Behavior:**
  - Once per minute, print a line with UTC timestamp, load, and memory usage.
  - After `N` loops, print completion line.
- **Sample Output:**

```
[2025-10-18T14:12:00Z] load=0.12 0.07 0.02
mem_used=1024/7948MB
[2025-10-18T14:13:00Z] load=0.09 0.06 0.02
mem_used=1030/7948MB
 done (loops=2)
```

This is a summary of what each one of these scripts does.

Server Side (VM1)

- **srv_access_accept.sh**
  This script is the *entry point* for client requests. It checks whether the client's access code is valid, generates a unique request ID, and creates a small "request file" containing the details (user, filename, timestamp, etc.).
- **srv_analyze.sh**
  This runs continuously in the background. It looks for new request files and their corresponding uploaded payloads. Once found, it analyzes the payload (counts lines, words, bytes), writes the results to the results folder, logs the processing, and removes the processed request.
- **srv_stats.sh**
  This script looks at the server's worker log, summarizes how many jobs were processed, what the average processing time was, and who used the system most. It also compresses and archives old logs.
- **srv_net_health.sh**
  This is like a connectivity check. It pings a target machine (e.g., the client VM) and checks whether SSH (port 22) is open. It then appends the results with a timestamp into a health log.

Client Side (VM2)

- **cli_submit.sh**
  This is how the client sends work. It compresses the input file, sets safe permissions, asks the server for approval (via access code), and if accepted, uploads the payload to the server's queue.
- **cli_fetch.sh**
  This script retrieves results. It checks if the job is done on the server. If results are ready, it downloads them; if still processing, it reports that; if missing, it warns.
- **cli_push_metrics.sh**
  This collects system information from the client (CPU, memory, load, etc.) and sends a snapshot to the server.
- **cli_fix_perms.sh**
  This is a housekeeping script. It scans the client's files for unsafe permissions (like 777) and fixes them, recording changes in a log.
- **cli_watchdog.sh**
  This is a local monitoring script. It runs for a specified number of loops (default 2), every minute prints load and memory usage, then stops.

Second Part

Determine the Number of Processor Cores: The server should ascertain the number of available processor cores. This information is essential for optimizing parallel processing, as it allows the server to tailor its process creation to the capabilities of the hardware.

Create siblings Based on Core Count: Leveraging the information about the number of cores, the server should create an equal number of siblings. This ensures that the system's processing capabilities are fully utilized, with each core handling a separate sibling process.

For simplicity: For process creation always consider half of the cores.

Each sibling will be assigned randomly one of the following tasks:

1. **Create File**
2. **Delete File**
3. **Copy File**
4. **Display File Information**
5. **Search for a Word in File**
6. **List Directory Contents**

Pass all arguments through the command line (CLI).

The Parent creates a log file.
Siblings will be reporting to the parent by logging their successful task along with the time.
Once done, parent will print the log file contents on the screen.

Instructions & Deliverables: Please read carefully

1. **Submission Requirements:**
    - **Word Document:**
        - Cover page with group member details (Name, ID).
        - Task distribution table (tasks assigned per member + contribution %).
    - **Scripts:** All scripts must be error-free and include inline comments explaining logic.
    - **C files:** All C file(s) must be error-free and include inline comments explaining logic.
    - **Zip File:** Named as Student1_Student2_Student3_Student4.zip.
2. **Anti-Plagiarism Measures:**
    - **Demo Session:** Randomly assigned tasks must be executed live.
    - **Script Defense:** Students must explain every line of code during the demo.
3. **Penalties:**

- Copying and/or plagiarism (-100%) which includes:
    - **Inappropriate interaction with any other student, outside agency, website, or software that generates assessment responses.**
- Shell Script should be error free (Errors) (-25%).
- In case of late submission, (-10%) for each day of delay (Max 3 days delay).
- A group of three to four students can work on the project.
- Team members are required to meet regularly for discussion and workload distribution.
- It is the right of the instructor to use any way of testing the student in the discussion and demo session, and according to that in some cases (100%) graded project may be down to (-100%) graded project.
- Discussion & Demo 50%. + Student Work 50%.
- One demo after the submission of the second project phase.