

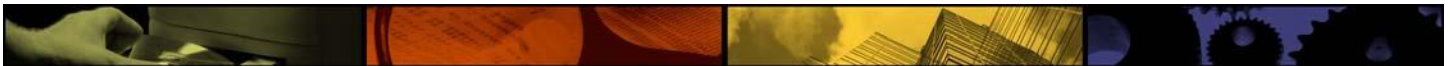


*A Personal Edge Tool (PET)
Proof-of-Concept (POC)
for Self-Hosted Modular AGI*

- 🔍 **Personal:** Designed to run locally on your hardware..
 - 🔍 **Edge:** At the literal edge of innovation—self-hosted, offline-capable AI.
 - 🔍 **Tool:** Modular, script-based, transparent. You build it, own it, control it.
-



WHITE PAPER



Linux AI Backend Documentation

Project: Samaritan

Local AI Development
2025-04-02 (Proof-of-Concept)
Ubuntu 22.04 LTS

Content Overview:

This document presents a practical blueprint for building a modular, self-hosted AI backend using open-source tools on Linux. "Project: Samaritan" was designed as both a proof-of-concept and a teachable framework—demonstrating how reproducible AI infrastructure can be deployed even on aging, low-spec hardware.


Note: While the current core documentation is being released for transparency and educational purposes, the actual scripts are still in development. Full release will follow the production implementation of the Samaritan AI Backend on upgraded hardware after I conclude the POC.

Samaritan Linux AI Backend Documentation

Version: 2025-04-02 (POC)

Platform: Ubuntu 22.04 LTS

1. Executive Summary (BLUF)

- **Purpose:**
A modular, reproducible AI backend platform that hosts various AI products (Ollama, ComfyUI, etc.) in containerized instances.
 - **Core Approach:**
 - Uses sequential, single-purpose BASH scripts.
 - Leverages Docker & Docker Compose for container orchestration.
 - Provides custom web UIs and a central monitoring dashboard.
 - **Key Benefits:**
 - Maintainability, reproducibility, and resilience.
 - Clear separation of concerns through script-based configuration.
 - Flexibility to upgrade or extend services.
 -  **Note:** While this document functions as a traditional White Paper by outlining the goals, architecture, and rationale behind Project: Samaritan, it also serves as a detailed Proof-of-Concept (POC) implementation report. As such, it bridges theory and execution—offering both high-level guidance and low-level reproducibility.
-

2. Project Goals & Objectives

- **Reproducible Setup:**
Automate environment build (OS updates, Docker, security tools, directory structures).
- **Service Modularity:**
Separate scripts for:
 - Base System Setup (Scripts 1a–1c)
 - AI Services (Ollama, ComfyUI)
 - Web UIs & Dashboard (Scripts 3a, 7)
- **Reliable Communication:**
Ensure proper API (CORS) and UI interactions.
- **Resilience & Recovery:**
Use Docker restart policies and scripted verifications.

3. Architecture Overview

- **System Components:**
 1. **Base Setup:**
 - OS update, Docker install, firewall (UFW), Fail2ban, Samba configuration.
 2. **Ollama Service:**
 - LLM server container with persistent storage and CORS configuration.
 3. **Web Interfaces:**
 - Basic Ollama UI (Python-based) and ComfyUI (CPU mode container).
 4. **Dashboard:**
 - Custom Node.js dashboard deployed via Docker Compose.
 - **Data & Volumes:**

All persistent data stored under /ai-data (models, logs, configs).
-

4. Disclaimer (Important for Public Sharing)

- **Educational / Proof-of-Concept:** This project and its scripts are provided primarily for educational and Proof-of-Concept purposes. They demonstrate one way to build a self-hosted AI backend but are not inherently production-hardened.
 - **User Responsibility:** Users are responsible for understanding the scripts and their implications before execution. Running these scripts will modify your system, install packages, configure services, and open firewall ports. Use on a dedicated test machine is recommended.
 - **Security:** While basic security measures (UFW, Fail2ban, user permissions) are included, users should conduct their own security assessments based on their environment and requirements. Notably, the NodeJS dashboard (Script 7) mounts the Docker socket directly for simplicity in this POC; using a Docker Socket Proxy is generally recommended for better security in production environments.
 - **Hardware Limitations:** The documentation reflects testing on specific low-spec hardware (No GPU, No AVX, Low RAM). Performance and compatibility will vary significantly on different hardware.
-

5. Implementation Status

Completed & Working

- **Script 1a:** Base OS setup with Docker, UFW, Fail2ban, and initial directories.

- **Script 1b: Verification of system state.**
- **Script 1c: Samba installation and configuration for file sharing.**
- **Script 2: Ollama container deployment with proper CORS (OLLAMA_ORIGINS).**
- **Script 3a: Basic Ollama WebUI setup (HTML/JS/Python with systemd service).**
- **Script 4: Custom ComfyUI container (CPU mode; performance limitations acknowledged).**

Deferred / In Progress

- **Script 5 (LM Studio):** Deferred due to image availability and overlap with Ollama.
- **Script 6 (Piper TTS):** Deferred (hardware incompatibility: missing AVX).
- **Script 7: Dashboard container**—running but with UI configuration issues needing further debugging.

6. Hardware & POC Limitations

- **Primary Testing Hardware (Samaritan):**
 - Pentium Dual-Core E6600 @ 3.06 GHz, 8 GB RAM
 - Limitations: No GPU, no AVX support
- **Additional Testing Hardware (Colossus):**
 - Colossus: Dell Dimension 9100 / 2 GB RAM
 - Limitations: No GPU, no AVX support
- **Impact:**
 - Slower ComfyUI performance (CPU-only)
 - RAM constraints affect large model loading
 - Incompatible container images for AVX-dependent services

7. Modular Approach & Key Decisions

- **Sequential Script Execution:**
 - Strict order: Base Setup → Service Deployment → UIs → Dashboard.
- **Verification Steps:**
 - Automated checks (Script 1b) to ensure correctness.
- **Container Customization:**
 - When public images were unsuitable, custom Dockerfiles were generated (e.g., for ComfyUI).

- **Deferred Items:**
 - LM Studio & Piper TTS deferred due to complexity/hardware issues.
-

8. Next Steps & Recommendations

- **Dashboard Debugging:**
 - Simplify YAML config files and validate syntax.
 - Consider pinning to a stable image version if issues persist.
 - **Enhancements:**
 - Plan for UI enhancements (Scripts 3b/3c) after stabilization.
 - Revisit deferred services if new solutions or hardware upgrades are available.
 - **Security & Production Readiness:**
 - Replace direct Docker socket mounting with a secure proxy in production.
 - Review Samba and UFW configurations for hardened security.
-

9. General Script Overview

9.1. Base System Setup (Scripts 1a, 1b, 1c)

- **Script 1a:**
 - Installs OS updates, Docker, utilities, UFW, Fail2ban.
 - Creates /ai-data and log directories.
- **Script 1b:**
 - Verifies directory structure, Docker installation, UFW status, and service activity.
- **Script 1c:**
 - Installs and configures Samba for /ai-data and user home directories.
 - Sets Samba password and adjusts UFW rules.

8.2. AI Service Deployment

- **Script 2 (Ollama Container):**
 - Deploys the Ollama LLM container on port 11434.
 - Mounts /ai-data/ollama-models and configures CORS for UI access.
- **Script 3a (Basic Ollama WebUI):**

- Sets up a simple HTML/JS/Python web server on port 8080.
 - Integrates with the Ollama API using proper CORS headers.
- **Script 4 (ComfyUI Container):**
 - Builds a custom ComfyUI Docker image for CPU mode.
 - Maps port 8188 and persists data to /ai-data/comfyui-data.

9.3. Deferred Services

- **Script 5 (LM Studio): Deferred**
- **Script 6 (Piper TTS): Deferred**

9.4. Monitoring Dashboard (Script 7)

- **Dashboard Setup:**
 - Node.js/Express backend with Docker Compose.
 - Serves on port 3000; gathers system metrics, Docker status, and service health.
 - Uses mounted Docker socket (with potential security caveats).
-

10. Component Interactions & Data Flow

- **User Interaction:**
 - **Web UIs:**
 - Dashboard: http://<server_ip>:3000
 - Ollama UI: http://<server_ip>:8080
 - ComfyUI: http://<server_ip>:8188
 - **Backend Communications:**
 - Ollama: WebUI directly fetches from API on port 11434.
 - Dashboard:
 - Frontend JS fetches system metrics and Docker status from Node.js backend.
 - Node.js backend communicates with the Docker daemon (via /var/run/docker.sock) and executes system commands.
 - **Data Persistence:**
 - All persistent data (models, logs, configs) resides under /ai-data.
-

11. Operational Considerations

11.1. Data Management

- **Central Storage:** /ai-data
- **Persistent Volumes:**
 - **Ollama Models:** /ai-data/ollama-models
 - **ComfyUI Data:** /ai-data/comfyui-data
 - **WebUI and Dashboard Files:** /ai-data/webui & /ai-data/dashboard

11.2. Access & Monitoring

- **User Access Points:**
 - Dashboard, WebUIs, SSH (port 22), Samba share (\\<server_ip>\ai-data)
- **Logging:**
 - Script logs in /ai-data/BuildLogs
 - Container logs (via docker logs)
 - System logs (journalctl, syslog)
- **Monitoring Tools:**
 - Custom Dashboard UI
 - Terminal tools: ctop, htop, glances

11.3. Security Measures

- **Firewall (UFW):**
 - Default deny incoming, explicitly allows needed ports (22, 11434, 8080, 8188, 3000).
 - **Service Hardening:**
 - Fail2ban for SSH protection.
 - Docker socket access is currently direct (for POC); recommend a proxy for production.
-

12. Key Learnings & Future Considerations

- **CORS Configuration:**
Crucial for browser-to-backend API interactions.
- **Custom Docker Builds:**
Provide necessary control, though at the cost of increased complexity.

- **Hardware Limitations:**
Performance and compatibility are significantly affected by CPU features, RAM, and lack of GPU.
- **Script Modularity:**
Sequential, single-purpose scripts simplify debugging and maintenance.
- **Future Enhancements:**
 - Implement a Docker Socket Proxy for improved security.
 - Explore GPU support and refined UI enhancements.
 - Revisit deferred services with updated images or hardware upgrades.

Detailed Script and Procedural Documentation Follows.

Detailed Script & Procedural Documentation

(This section contains the combined High-Level and Detailed Runbook documentation for each script, based on the actual code.)

Script 1a: 1-ServerOS-Base-Build-Script.sh

(Revision based on actual script)

High-Level Documentation

- **Purpose:** Sets up the foundational environment for the Samaritan Linux AI Backend, specifically targeting Ubuntu 22.04 LTS. It prepares the system with necessary directories, packages, utilities, security configurations, and the core containerization engine (Docker).
- **Key Actions:**
 - Performs safety checks (requires root via sudo, exits on error).
 - Creates base directories (/ai-data, /ai-data/BuildLogs) and sets ownership to the user who invoked sudo.
 - Writes a detailed documentation block about itself into /ai-data/BuildLogs/breakdown_1-ServerOS-Base-Build.txt.
 - Updates and upgrades system packages non-interactively.
 - Installs essential utilities: curl, wget, git, vim, htop, glances, net-tools, python3-pip, build-essential, ca-certificates, gnupg.
 - Installs security tools: ufw, fail2ban.
 - Installs Docker Engine (docker-ce, docker-ce-cli, containerd.io), Docker Buildx Plugin, and Docker Compose Plugin using Docker's official repository, including GPG key setup. Checks for existing key/repo to be somewhat idempotent.
 - Starts and enables the Docker service.
 - Adds the invoking user (\$INVOKING_USER) to the docker group (if not already added) and provides a strong reminder to log out/in.
 - Configures UFW: Allows SSH (port 22/tcp) with a comment, enables UFW if inactive (using --force), and reloads if already active. Checks existing rules/status.
 - Starts and enables the Fail2ban service and performs a basic status check.
 - Installs ctop (container monitoring tool) by downloading the binary from GitHub (supports x86_64/amd64 and aarch64) if not already installed.
- **Dependencies:**
 - Base Ubuntu 22.04 LTS installation.
 - Internet connectivity.

- Must be executed by a non-root user using sudo.
- **Outcome:** A secured Ubuntu 22.04 base system with Docker & Docker Compose ready, essential tools installed, firewall configured, Fail2ban running, ctop installed, the /ai-data directory structure created with appropriate ownership, and a self-documenting log file generated. Requires user logout/login for Docker group changes to apply.

Detailed Runbook Documentation

- **Objective:** To automate the setup of a standardized, secure, and functional Ubuntu 22.04 base environment suitable for deploying the Samaritan AI Backend services via Docker.
- **Prerequisites:**
 - Freshly installed Ubuntu 22.04 LTS.
 - A non-root user account with sudo privileges.
 - Internet connection.
 - Recommended: Configure a static IP address *before* running this script if required for the server.

- **Execution:**

1. Save the script code to a file named 1-ServerOS-Base-Build-Script.sh.
2. Make the script executable: `chmod +x 1-ServerOS-Base-Build-Script.sh`.
3. Run the script using sudo:

```
sudo ./1-ServerOS-Base-Build-Script.sh | sudo tee /ai-data/BuildLogs/1-ServerOS-Base-Build.log
```

content_copydownload

Use code [with caution](#).Bash

- Using sudo tee ensures the log file in /ai-data/BuildLogs can be created/written to, even before directory ownership is fully settled, and captures all output.
- The script also creates /ai-data/BuildLogs/breakdown_1-ServerOS-Base-Build.txt with its internal documentation.

- **Step-by-Step Breakdown:**

1. **Configuration & Safety Checks:**

- Sets internal variables (LxAiBuildLogs, SCRIPT_NAME, SCRIPT_LOG_FILE).
- Exits immediately if any command fails (set -e).
- Verifies the script is run with root privileges (id -u == 0) and was invoked via sudo (checks \$SUDO_USER). Exits if not run with root privileges. Issues a warning if run directly as root but proceeds.

- Determines the invoking user (`INVOKING_USER=${SUDO_USER:-$(whoami)}`) for setting permissions later.
2. **[Step 1/8] Create Base Directories:**
 - Creates `/ai-data/BuildLogs` using `sudo mkdir -p`.
 - Sets ownership of `/ai-data` (if it exists) and `/ai-data/BuildLogs` recursively to `$INVOKING_USER` using `sudo chown -R`. Uses `|| true` on `/ai-data` to prevent exit if it already existed with different ownership (though the logs dir ownership should succeed).
 3. **[Step 2/8] Write Documentation Log:**
 - Creates a self-documentation file (`/ai-data/BuildLogs/breakdown_1-ServerOS-Base-Build.txt`) using a `sudo tee` heredoc.
 - Sets ownership of this log file to `$INVOKING_USER`.
 4. **[Step 3/8] System Update & Upgrade:**
 - Updates package lists: `sudo apt update`.
 - Upgrades installed packages non-interactively: `sudo DEBIAN_FRONTEND=noninteractive apt upgrade -y`.
 5. **[Step 4/8] Install Essential & Security Packages:**
 - Installs a list of packages using `sudo apt install -y: curl, wget, git, vim, htop, glances, net-tools, python3-pip, build-essential, ca-certificates, gnupg, ufw, fail2ban`.
 6. **[Step 5/8] Install Docker Engine and Docker Compose:**
 - Ensures keyring directory exists: `sudo install -m 0755 -d /etc/apt/keyrings`.
 - Downloads Docker GPG key (`docker.asc`) if it doesn't exist, places it in `/etc/apt/keyrings/`, and sets permissions.
 - Adds the Docker stable repository to `/etc/apt/sources.list.d/docker.list` if the file doesn't exist, then runs `sudo apt update`.
 - Installs Docker packages: `sudo apt install -y docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-compose-plugin`.
 - Verifies installation using `docker --version` and `docker compose version`. Exits if commands fail.
 - Enables and starts the Docker service: `sudo systemctl enable docker, sudo systemctl start docker`.
 7. **[Step 6/8] Configure Docker Group Membership:**
 - Checks if `$INVOKING_USER` is already in the docker group using `groups "$INVOKING_USER" | grep`.
 - If not, adds the user using `sudo usermod -aG docker "$INVOKING_USER"`.

- Prints a prominent warning message reminding the user to **log out and log back in** for the group change to take effect.

8. **[Step 7/8] Configure Firewall (UFW):**

- Defines SSH_PORT=22.
- Checks if UFW is active and if an ALLOW rule for port 22/tcp already exists.
- If not configured: Adds the rule `sudo ufw allow ${SSH_PORT}/tcp comment 'Allow SSH access'`.
- If UFW is inactive (`ufw status | grep -qw inactive`): Enables it using `sudo ufw --force enable`.
- If UFW was already active: Reloads rules using `sudo ufw reload`.
- Prints the current `sudo ufw status verbose`.

9. **[Step 8/8] Configure Fail2Ban & Install ctop:**

- Enables and starts the Fail2ban service: `sudo systemctl enable fail2ban, sudo systemctl start fail2ban`.
- Checks if the Fail2ban service is active using `systemctl is-active --quiet fail2ban` and prints status or a warning.
- Checks if ctop command exists.
- If not: Determines system architecture (`uname -m`). Selects the appropriate ctop release URL for x86_64 (amd64) or aarch64 (arm64). Skips if architecture is unsupported.
- Downloads the ctop binary using `sudo wget to /usr/local/bin/ctop`.
- Makes it executable: `sudo chmod +x /usr/local/bin/ctop`.
- Verifies installation by checking command `-v ctop` again and prints success or error message.

10. **Final Messages:**

- Prints a completion banner.
- Summarizes key actions performed.
- Reiterates the **logout/login reminder** if the user was added to the Docker group.
- Reminds the user about static IP configuration.
- Suggests running the verification script (`1b-Check-Setup-Verify.sh`).

11. **Exit:** Exits with status 0 on success (exit 0).

- **Verification:**

- Check script execution log: `cat /ai-data/BuildLogs/1-ServerOS-Base-Build.log`.
- Check internal documentation log: `cat /ai-data/BuildLogs/breakdown_1-ServerOS-Base-Build.txt`.

- Check directory existence and ownership: `ls -ld /ai-data /ai-data/BuildLogs`. Should be owned by `$INVOKING_USER`.
- Check installed packages: `dpkg -s ufw fail2ban docker-ce docker-compose-plugin glances ctop | grep Status`.
- Check Docker version: `docker --version`
- Check Docker Compose version: `docker compose version`
- Check Docker service status: `systemctl status docker` (should be active/running).
- **After logout/login:** Check if user is in docker group: `groups` (should list docker).
- Check UFW status and rules: `sudo ufw status verbose` (should show active, default deny incoming, allow outgoing, port 22/tcp allowed with comment).
- Check Fail2ban status: `systemctl status fail2ban` (should be active/running).
- Check ctop runs: `ctop -v` (should show version) or just `ctop`.
- **Key Files/Directories Created/Modified:**
 - `/ai-data/` (Directory potentially created, ownership set)
 - `/ai-data/BuildLogs/` (Directory created, ownership set)
 - `/ai-data/BuildLogs/breakdown_1-ServerOS-Base-Build.txt` (Log file created, ownership set)
 - `/ai-data/BuildLogs/1-ServerOS-Base-Build.log` (Log file created via tee)
 - `/etc/apt/keyrings/docker.asc` (Docker GPG key, if didn't exist)
 - `/etc/apt/sources.list.d/docker.list` (Docker repository source, if didn't exist)
 - `/usr/local/bin/ctop` (ctop binary, if didn't exist and arch supported)
 - `/etc/ufw/` (Configuration files modified/created by UFW setup)
 - `/etc/fail2ban/` (Configuration files created by Fail2ban installation)
 - `/etc/group` (Modified to add user to docker group)
 - System package database updated (apt update/upgrade).
 - Systemd service links for docker and fail2ban enabled.
- **Integration Points:** Provides the foundational OS configuration, Docker runtime, directory structure (`/ai-data`), security (UFW, Fail2ban), and utilities required by all subsequent scripts (1b, 1c, 2, 3a, 4, 7-NodeJS). Sets ownership of `/ai-data` based on `$INVOKING_USER`.
- **Potential Issues & Troubleshooting:**
 - **Execution Errors:** Script exits due to set -e. Check the output log (`/ai-data/BuildLogs/1-ServerOS-Base-Build.log`) for the specific command that failed. Common causes: network errors during apt or curl/wget, GPG key issues, repository problems.

- **Permission Denied (Post-Script):** User cannot run docker commands without sudo. **Crucially requires logout and login** after the script adds the user to the docker group. Verify with groups command after logging back in.
- **UFW Blocking Access:** UFW is enabled aggressively. Ensure any necessary ports for later services are explicitly allowed in subsequent scripts. `sudo ufw status` is key.
- **Fail2Ban Service Down:** Check status with `sudo systemctl status fail2ban` and logs with `sudo journalctl -u fail2ban`. Could be configuration issues (though defaults are usually okay for SSH) or conflicts.
- **ctop Install Failed:** Check log output. Unsupported architecture, GitHub unreachable, wget failed, `/usr/local/bin` not writable (though sudo should handle this).
- **Upgrade Issues:** `DEBIAN_FRONTEND=noninteractive` might hide prompts but could lead to unexpected states if an upgrade requires manual intervention. Check `/var/log/apt/term.log` for details if upgrades seem problematic.
- **Ownership Issues:** If `/ai-data` existed previously with restrictive permissions, `chown` might partially fail (though the `|| true` mitigates script exit). Verify final ownership with `ls -ld /ai-data /ai-data/BuildLogs`.
- **Rollback/Cleanup:**
 - Remove installed packages: `sudo apt remove --purge curl wget git vim htop glances net-tools python3-pip build-essential ca-certificates gnupg ufw fail2ban docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-compose-plugin`
 - Remove ctop: `sudo rm /usr/local/bin/ctop`
 - Remove Docker GPG key & repo: `sudo rm /etc/apt/keyrings/docker.asc /etc/apt/sources.list.d/docker.list`
 - Disable/reset UFW: `sudo ufw reset` (disables and removes rules)
 - Disable/stop Fail2ban: `sudo systemctl disable --now fail2ban`
 - Remove user from docker group: `sudo gpasswd -d $INVOKING_USER docker` (replace `$INVOKING_USER` with the actual username)
 - Remove directories: `sudo rm -rf /ai-data` (Use with extreme caution!)

Script 1b: 1b-Check-Setup-Verify.sh

(Revision based on actual script)

High-Level Documentation

- **Purpose:** To automatically verify that the system state matches the expected outcome after successfully running `1a-ServerOS-Base-Build-Script.sh`. It checks crucial aspects like directory structure, ownership, package installation, service status, and user configurations.
- **Key Actions:**

- Defines expected directory paths and the user (\$EXPECTED_USER) who should own logs and be in the docker group (derived from \$SUDO_USER or whoami).
- Uses helper functions (check_status, check_status_sudo) to execute checks, capture success/failure based on command exit codes, and print formatted PASS/FAIL/SKIP status with color-coded details on failure.
- Handles sudo requirements gracefully for specific checks (UFW), prompting for a password only if necessary.
- Performs the following checks sequentially:
 1. Existence of /ai-data and /ai-data/BuildLogs directories.
 2. Ownership of /ai-data/BuildLogs by \$EXPECTED_USER.
 3. Existence of the Script 1a documentation log file.
 4. Installation status of essential packages (git, curl, wget, htop, glances, ufw, fail2ban, python3-pip, ctop) using dpkg -s and command -v.
 5. Availability of docker and docker compose commands and checks if the docker service is active (systemctl is-active).
 6. Verification that \$EXPECTED_USER is a member of the docker group (groups command). **Crucially depends on user having logged out/in after Script 1a.**
 7. Checks if UFW firewall is active and allows SSH (port 22/tcp) using sudo via the helper function.
 8. Checks if the Fail2Ban service is active (systemctl is-active).
- Outputs a final summary with counts of Passed, Failed, and Skipped checks.
- Exits with status 0 if all checks pass, 1 if any checks fail, and 2 if checks were skipped (due to sudo issues) but none failed.
- **Dependencies:** 1a-ServerOS-Base-Build-Script.sh must have been executed successfully. The user running this script should ideally be the \$EXPECTED_USER identified within the script (the one who ran Script 1a with sudo).
- **Outcome:** Provides a clear, automated report on the success and correctness of the base system setup performed by Script 1a, highlighting any discrepancies or issues that need attention before proceeding.

Detailed Runbook Documentation

- **Objective:** To provide an automated, non-intrusive way to confirm that the critical components and configurations established by 1a-ServerOS-Base-Build-Script.sh are correctly in place.
- **Prerequisites:**
 - 1a-ServerOS-Base-Build-Script.sh has been run successfully.
 - **Crucial:** If Script 1a added the user to the docker group, that user **must have logged out and logged back in before** running this verification script for Check 6 (Docker User Group) to pass accurately.

- The script should generally be run by the user identified as \$EXPECTED_USER (usually the user who ran Script 1a via sudo). It does *not* need to be run *with* sudo, but it *will* attempt to use sudo internally for UFW checks and may prompt for a password if credentials aren't cached.

- **Execution:**

- `bash ./1b-Check-Setup-Verify.sh`

Or simply: `./1b-Check-Setup-Verify.sh` if executable permission is set

content_copydownload




Use code [with caution](#).Bash

- **Step-by-Step Breakdown:**

1. Initialization:

- Sets configuration variables (LxAiBuildLogs, SCRIPT1_LOG_FILE).
- Determines the \$EXPECTED_USER based on \$SUDO_USER (if script was somehow run with sudo) or whoami.
- Initializes counters: PASS_COUNT=0, FAIL_COUNT=0, TOTAL_CHECKS=0.

2. Helper Functions Defined:

- `check_status`: Takes description, command string, optional failure details. Executes command using `bash -c`, suppresses output (`> /dev/null 2>&1`), checks exit code (\$?), prints formatted "[ PASS]" or "[ FAIL]" message, includes failure details in red if provided, and increments counters.
- `check_status_sudo`: Takes description, command string (without sudo), optional failure details. Checks if `sudo -n true` works (non-interactive). If not, attempts interactive sudo echo to cache credentials. If authentication fails, prints "[ SKIP]". If sudo is available, it calls `check_status` prepending sudo to the command string.

3. **Header Output:** Prints a title banner identifying the script and the users involved (\$whoami vs \$EXPECTED_USER).

4. Check 1: Base Directories:

- Uses `check_status` with `[-d <path>]` to verify directory existence for /ai-data and /ai-data/BuildLogs.

5. Check 2: Log Directory Ownership:

- Uses `check_status` with `stat -c '%U' '$LxAiBuildLogs' | grep -q \"^${EXPECTED_USER}\\$\"` to verify the owner of the log directory matches \$EXPECTED_USER.

6. Check 3: Script 1 Documentation Log:

- Uses `check_status` with `[-f '$SCRIPT1_LOG_FILE']` to verify the specific log file from Script 1a exists.

7. Check 4: Essential Packages:

- Defines an array packages of required package names.
- Loops through the array:
 - Uses `dpkg -s "$pkg"` to check if package is installed via apt.
 - Includes a special check for `ctop` using command `-v ctop` since it might be manually installed.
 - Appends missing package names to `missing_packages` string.
- Calls `check_status` with `true` (passes) if `missing_packages` is empty, or `false` (fails) and provides the list of missing packages if not empty.

8. Check 5: Docker Installation & Service:

- Uses `check_status` with `docker --version` to check if the command runs successfully for the current user.
- Uses `check_status` with `docker compose version` for the compose plugin.
- Uses `check_status` with `systemctl is-active --quiet docker` to check if the systemd service is running.

9. Check 6: Docker User Group:

- Uses `check_status` with `groups '$EXPECTED_USER' | grep -qw 'docker'` to verify `$EXPECTED_USER` is listed in the docker group's membership. **Relies on prior logout/login.**

10. Check 7: UFW Firewall Status (needs sudo):

- Uses `check_status_sudo` with `ufw status | grep -qw active` to check if UFW is active.
- Uses `check_status_sudo` with `ufw status numbered | grep -qE \"^\\s*\\[\\s*[0-9]+\\]\\s*22/tcp\\s*ALLOW IN\"` to specifically check for the SSH ALLOW rule using a regular expression.







11. Check 8: Fail2Ban Status:

- Uses `check_status` with `systemctl is-active --quiet fail2ban` to check if the service is running.

12. Summary Output:

- Prints a separator and summary counts for Total, Passed, Failed, and Skipped checks.

13. Final Status & Exit:

- Prints a final " All passed", " Some skipped", or " Some failed" message.
- Exits with 0 (all passed), 1 (failures), or 2 (skipped, no failures).
- **Verification:** The script's output is the verification. Review the "[ PASS]", "[ FAIL]", or "[ SKIP]" status for each check and the final summary. Address any failures or skips before proceeding.

- **Key Files/Directories Created/Modified:** None. This script is strictly read-only regarding system configuration.
 - **Integration Points:** Serves as a crucial validation checkpoint after Script 1a. A successful run (exit code 0) provides confidence to proceed to Script 1c or subsequent setup scripts. Failures indicate issues with the base setup that need fixing.
 - **Potential Issues & Troubleshooting:**
 - **Check 6 (Docker Group) Fails:** Most common reason is the user did not log out and log back in after Script 1a added them to the docker group. Verify manually with `groups $EXPECTED_USER`.
 - **Check 7 (UFW) Skipped:** The script couldn't get sudo privileges non-interactively and the user likely cancelled the password prompt or entered it incorrectly. Run `sudo ufw status` manually to verify. Ensure the user running the script has sudo rights.
 - **Check 5 (Docker commands) Fails:** If Check 6 failed due to no logout/login, these might fail too if run *without* sudo. If Check 6 passed, this might indicate a deeper Docker installation or service issue. Check `systemctl status docker`.
 - **Check 4 (Packages) Fails:** A package listed wasn't installed correctly by Script 1a. Check Script 1a logs (`/ai-data/BuildLogs/1-ServerOS-Base-Build.log`) or try installing manually (`sudo apt install <package>`). Check `ctop` path (which `ctop`) if it fails.
 - **Check 2 or 3 (Ownership/Log File) Fails:** Indicates potential issues during Script 1a's directory creation or permission setting. Check ownership (`ls -ld /ai-data/BuildLogs`) and file existence manually. Review Script 1a logs.
 - **Check 5 or 8 (Service Status) Fails:** The docker or fail2ban service isn't running. Use `sudo systemctl status <service>` and `sudo journalctl -u <service>` to diagnose why it failed to start in Script 1a or subsequently stopped.
 - **Rollback/Cleanup:** Not applicable (read-only script).
-

Script 1c: 1c-Setup-Samba.sh

(Revision based on actual script)

High-Level Documentation

- **Purpose:** To install and configure the Samba file-sharing service on the Samaritan server, enabling network access to specific directories.
- **Key Actions:**
 - Performs safety checks (requires root via sudo, exits on error). Identifies the invoking user (`$INVOKING_USER`) who will be granted Samba access.
 - Creates a self-documentation block in `/ai-data/BuildLogs/breakdown_1c-Setup-Samba.txt`.
 - Installs `samba` and `samba-common-bin` packages using `apt`.

- Backs up the existing `/etc/samba/smb.conf` file with a timestamp.
- Generates a new `/etc/samba/smb.conf` file with:
 - Sensible global settings (workgroup, logging, security=user, disabled printing, performance tweaks).
 - An `[ai-data]` share pointing to `/ai-data`, explicitly granting write access only to `$INVOKING_USER`.
 - A standard `[homes]` share enabling users to connect to their own home directories (`\\server\username`).
- Runs `testparm -s` to validate the syntax of the newly generated `smb.conf`. Exits and attempts restore from backup if syntax check fails.
- **Interactively prompts the user** to set a Samba-specific password for `$INVOKING_USER` using `smbpasswd -a`.
- Adds a UFW rule to allow Samba traffic (`ufw allow samba`) if UFW is active. Issues a warning if UFW is inactive.
- Restarts and enables the `smbd` and `nmbd` `systemd` services.
- Verifies that both `smbd` and `nmbd` services are active using `systemctl is-active`.
- **Dependencies:**
 - `1a-ServerOS-Base-Build-Script.sh` must have been executed successfully (provides base OS, UFW, `/ai-data` directory, invoking user context).
 - Requires interactive input from the user running the script to set the Samba password.
- **Outcome:** Samba service installed, configured, and running. The `/ai-data` directory and the invoking user's home directory are shared over the network, accessible using the invoking user's username and the newly set Samba password. Firewall rules are adjusted accordingly.

Detailed Runbook Documentation

- **Objective:** To automate the setup and configuration of Samba file sharing for the `/ai-data` directory and user home directories, integrating with the system user accounts and UFW firewall.
- **Prerequisites:**
 - `1a-ServerOS-Base-Build-Script.sh` has been run successfully.
 - The script must be run using `sudo` by the user who requires Samba access to the shares (this user becomes `$INVOKING_USER`).
 - The user running the script must be prepared to enter and confirm a new password for Samba access when prompted. This password is *separate* from their Linux login password.
- **Execution:**
 1. Save the script code to a file named `1c-Setup-Samba.sh`.

2. Make the script executable: `chmod +x 1c-Setup-Samba.sh`.

3. Run the script using `sudo`:

```
sudo ./1c-Setup-Samba.sh | sudo tee /ai-data/BuildLogs/1c-Setup-Samba.log
```

content_copydownload

Use code [with caution](#).Bash

- Using `sudo tee` captures all output, including prompts (though not the password entry itself) and results, to the log file.
- The script also creates `/ai-data/BuildLogs/breakdown_1c-Setup-Samba.txt` with its internal documentation.

- **Step-by-Step Breakdown:**

1. **Configuration & Safety Checks:**

- Sets internal variables (`LxAiBuildLogs`, `SCRIPT_NAME`, `SCRIPT_LOG_FILE`).
- Sets `set -e` to exit on any command failure.
- Verifies root privileges (`id -u`) and exits if not root.
- Determines `$INVOKING_USER` from `$SUDO_USER` or defaults to `whoami` (likely 'root' if run directly as root, with a warning issued).

2. **[Step 1/8] Write Documentation Log:**

- Ensures log directory exists and is owned by `$INVOKING_USER`.
- Creates the self-documentation file (`/ai-data/BuildLogs/breakdown_${SCRIPT_NAME}.txt`) using `sudo tee` and a heredoc, setting ownership to `$INVOKING_USER`.

3. **[Step 2/8] Install Samba:**

- Runs `sudo apt update`.
- Installs packages: `sudo apt install -y samba samba-common-bin`.

4. **[Step 3/8] Backup Original Config:**

- Defines `SMB_CONF` path.
- Creates a timestamped backup filename (`$BACKUP_FILE`).
- If `$SMB_CONF` exists, copies it to `$BACKUP_FILE` using `sudo cp`.

5. **[Step 4/8] Create New Samba Configuration:**

- Uses `sudo tee` and a heredoc (`<< END-OF-CONFIG ... END-OF-CONFIG`) to write the new configuration directly to `/etc/samba/smb.conf`.
- The configuration includes:

- Global settings: workgroup, server string, logging, security = user, passdb backend = tdbsam, disabled printing, performance tweaks.
 - [ai-data] share: path = /ai-data, valid users = \$INVOKING_USER, writable = yes, browsable = yes, create mask = 0664, directory mask = 0775.
 - [homes] share: comment, browsable = no, writable = yes, valid users = %S, create mask = 0700, directory mask = 0700.
 - Validates the new configuration using `sudo testparm -s`. If it fails, prints error, attempts to restore backup (`sudo mv "$BACKUP_FILE" "$SMB_CONF"`), and exits with status 1.
6. **[Step 5/8] Set Samba Password:**
- Prints clear instructions explaining that a Samba-specific password needs to be set.
 - Executes `sudo smbpasswd -a "$INVOKING_USER"`, which **pauses the script and prompts the user interactively** to enter and confirm the new password.
7. **[Step 6/8] Configure Firewall (UFW):**
- Checks if UFW is active (`sudo ufw status | grep -qw active`).
 - If active, allows Samba traffic using the predefined application profile: `sudo ufw allow samba`. Prints confirmation and verifies rule presence.
 - If inactive, prints a warning message.
8. **[Step 7/8] Restart and Enable Samba Services:**
- Restarts services: `sudo systemctl restart smbd.service nmbd.service`.
 - Enables services for boot: `sudo systemctl enable smbd.service nmbd.service`.
9. **[Step 8/8] Verify Services:**
- Checks if smbd is active using `systemctl is-active --quiet smbd.service`. Prints status.
 - Checks if nmbd is active using `systemctl is-active --quiet nmbd.service`. Prints status.
 - Sets a flag `samba_ok` to false if either service is inactive.
10. **Final Messages:**
- Checks the `samba_ok` flag.
 - If true: Prints a success banner and provides connection details (server name/IP, share names, username) for accessing the shares.
 - If false: Prints a failure banner, advises checking logs (`systemctl status`, `journalctl`), mentions the config file path, and exits with status 1.
11. **Exit:** Exits with status 0 on success.
- **Verification:**

- Check script execution logs (/ai-data/BuildLogs/1c-Setup-Samba.log, /ai-data/BuildLogs/breakdown_1c-Setup-Samba.txt).
- Check Samba service status: `sudo systemctl status smbd.service nmbd.service` (both should be active/running).
- Check Samba configuration syntax: `sudo testparm -s` (should report OK).
- Check UFW rules: `sudo ufw status` (should show 'samba' allowed if UFW was active).
- **Test Connection:** From another computer on the same network (Windows, Linux, macOS), attempt to connect to `\\<server_ip_or_hostname>\ai-data` and `\\<server_ip_or_hostname>\<INVOKING_USER>`. Authenticate using `$INVOKING_USER` and the Samba password set during script execution. Test creating/deleting a file in ai-data share.
- **Key Files/Directories Created/Modified:**
 - /etc/samba/smb.conf (Overwritten with new configuration)
 - /etc/samba/smb.conf.bak_YYYYMMDD-HHMMSS (Backup file created)
 - /var/lib/samba/private/passdb.tdb (Or similar Samba database file, modified by smbpasswd)
 - /var/log/samba/ (Log directory used by Samba services)
 - UFW configuration updated (if UFW was active).
 - Systemd service links for smbd and nmbd enabled.
- **Integration Points:** Provides network file system access (SMB/CIFS) to the /ai-data directory, which is used by subsequent scripts (e.g., Script 2 for Ollama models, Script 4 for ComfyUI data, Script 7 for Dashboard config). Relies on the user account existing on the Linux system (typically the one created/used during OS install and used to run Script 1a).
- **Potential Issues & Troubleshooting:**
 - **Password Prompt Failure:** User enters mismatched passwords during the `smbpasswd -a` prompt. Simply re-run the script or run `sudo smbpasswd -a $INVOKING_USER` manually.
 - **Connection Refused:** Firewall blocking (check `sudo ufw status`). Samba services (`smbd`, `nmbd`) not running (check `sudo systemctl status`). Incorrect server IP/hostname used by client. Network configuration issues between client and server.
 - **Authentication Failed:** Incorrect username or password used on the client. Remember to use the Samba password, not the Linux login password. Check /etc/samba/smb.conf to ensure valid users = `$INVOKING_USER` is correct for the [ai-data] share. Check Samba logs (/var/log/samba/log.<client_hostname_or_ip>).
 - **Permission Denied (Post-Authentication):** Linux file system permissions on /ai-data prevent write access even though Samba allows it. Check `ls -ld /ai-data`. Ensure `$INVOKING_USER` has write permissions. The create mask and directory mask in smb.conf control permissions for *newly created* files via Samba.

- **Syntax Errors in smb.conf:** testparm check should catch this, but if manually edited later, run `sudo testparm -s` to check. Services might fail to start if config is invalid. Check `journalctl -u smbd -n 50`.
 - **nmbd Fails to Start:** Less common, but sometimes related to network interface binding or conflicts with other NetBIOS services. Check logs.
 - **Rollback/Cleanup:**
 - Stop and disable services: `sudo systemctl disable --now smbd.service nmbd.service`
 - Restore original Samba config (if backup exists): Find the latest `sudo ls -t /etc/samba/smb.conf.bak_* | head -n 1` and `sudo cp <backup_file> /etc/samba/smb.conf`. Restart services if needed.
 - Remove Samba user password entry: `sudo smbpasswd -x "$INVOKING_USER"`
 - Remove UFW rule: `sudo ufw delete allow samba`
 - Remove Samba packages: `sudo apt remove --purge samba samba-common-bin`
 - Remove Samba logs/cache (optional): `sudo rm -rf /var/log/samba /var/cache/samba /var/lib/samba`
-

Script 2: 2-Setup-Ollama-Container.sh

(Revision based on actual script)

High-Level Documentation

- **Purpose:** To set up and run the Ollama LLM server within a Docker container, ensuring it's configured correctly for persistent model storage and accessible by the planned web UI (Script 3a) via proper CORS settings.
- **Key Actions:**
 - Performs safety checks (requires root via sudo, exits on error). Determines invoking user (\$INVOKING_USER) for host directory ownership.
 - Defines container name (ollama-server), host model directory (/ai-data/ollama-models), ports (11434 for Ollama, 8080 reference for WebUI), and image name (ollama/ollama).
 - Attempts to auto-detect the server's primary IP address (\$SERVER_IP) for CORS configuration, falling back to 127.0.0.1 with a warning if detection fails.
 - Creates a self-documentation block in /ai-data/BuildLogs/breakdown_2-Setup-Ollama-Container.txt.
 - Checks Docker prerequisites: Verifies docker command exists and the docker service is active. Critically checks if \$INVOKING_USER can run `docker ps` without sudo (requires logout/login after Script 1a); exits with an informative message if permissions fail for a non-root user.
 - Creates the host model directory (/ai-data/ollama-models) and sets ownership to \$INVOKING_USER.
 - Pulls the latest \$OLLAMA_IMAGE using \$INVOKING_USER if possible, otherwise uses root.
 - Checks if a container named \$OLLAMA_CONTAINER_NAME already exists:

- If running and has the correct OLLAMA_ORIGINS environment variable (`http://localhost:8080,http://$SERVER_IP:8080`), reports success and exits (idempotency).
 - If stopped or has incorrect/missing CORS settings, stops (if running) and removes the existing container.
- Runs the Ollama container using `docker run -d` (as `$INVOKING_USER` if possible, else `root`) with:
 - `--name ollama-server`
 - Port mapping: `-p 11434:11434`
 - Volume mount: `-v /ai-data/ollama-models:/root/.ollama`
 - Environment variable: `-e OLLAMA_ORIGINS=http://localhost:8080,http://$SERVER_IP:8080` (**crucial for WebUI**)
 - Restart policy: `--restart unless-stopped`
- Waits 5 seconds and verifies the container is running using `docker ps`. Attempts a `curl` check against `http://localhost:11434`. Exits with an error if the container is not running.
- Configures UFW: Allows traffic on port 11434/tcp with a comment ("Ollama API Access") if UFW is active and the rule doesn't already exist.
- Provides final instructions on accessing the API/WebUI and pulling models using `docker exec`.
- **Dependencies:**
 - 1a-ServerOS-Base-Build-Script.sh must have been executed successfully (provides Docker, UFW, /ai-data, user context).
 - The user running this script (`$INVOKING_USER`) must have logged out and logged back in if Script 1a added them to the docker group, to ensure Docker command permissions are effective.
- **Outcome:** A running, resilient Ollama Docker container (`ollama-server`) serving the API on port 11434, storing models persistently in `/ai-data/ollama-models`, and correctly configured with CORS headers to allow requests from the WebUI (Script 3a) running on the same server. UFW firewall allows access to the Ollama port.

Detailed Runbook Documentation

- **Objective:** To automate the deployment of the Ollama service in a containerized, persistent manner, specifically configured to allow cross-origin requests from the associated WebUI.
- **Prerequisites:**
 - 1a-ServerOS-Base-Build-Script.sh completed successfully.
 - Docker service is running.
 - The script must be run using `sudo` by the user intended to own the host model directory (`/ai-data/ollama-models`).
 - **Crucial:** This user must have logged out and back in after Script 1a if they were added to the docker group.

- Internet connection (to pull the Ollama image).

- **Execution:**

1. Save the script code to a file named 2-Setup-Ollama-Container.sh.
2. Make the script executable: `chmod +x 2-Setup-Ollama-Container.sh`.
3. Run the script using `sudo`:

```
sudo ./2-Setup-Ollama-Container.sh | sudo tee /ai-data/BuildLogs/2-Setup-Ollama-Container.log
```

content_copydownload

Use code [with caution](#).Bash

- Using `sudo tee` captures all output to the log file.
- The script also creates `/ai-data/BuildLogs/breakdown_2-Setup-Ollama-Container.txt`.

- **Step-by-Step Breakdown:**

1. **Configuration & Safety Checks:**

- Sets variables (LxAiBuildLogs, SCRIPT_NAME, etc.).
- Sets `set -e`.
- Verifies root privileges and identifies `$INVOKING_USER`.
- Detects `$SERVER_IP` using `hostname -I | awk '{print $1}'`, provides fallback `127.0.0.1` and warning.

2. **[Step 1/9] Write Documentation Log:**

- Ensures log directory exists and is owned by `$INVOKING_USER`.
- Creates the self-documentation file using `sudo tee` and `heredoc`, sets ownership.

3. **[Step 2/9] Check Docker Prerequisites:**

- Checks command `-v docker`. Exits if not found.
- Checks `systemctl is-active --quiet docker`. Exits if not active.
- Attempts `sudo -u "$INVOKING_USER" docker ps`. If fails for non-root user, prints logout/login reminder and exits. If fails for root user, prints error and exits. If succeeds, confirms user has Docker permissions.

4. **[Step 3/9] Create Host Volume Directory:**

- Creates directory: `sudo mkdir -p "$OLLAMA_MODELS_DIR"`.
- Sets ownership: `sudo chown -R "${INVOKING_USER}:${INVOKING_USER}" "$OLLAMA_MODELS_DIR"`.

5. **[Step 4/9] Pull Ollama Image:**

- Attempts `sudo -u "$INVOKING_USER" docker pull "$OLLAMA_IMAGE"`.
- If that fails (e.g., user check in step 2 failed but script proceeded as root), falls back to `docker pull "$OLLAMA_IMAGE"`.

6. **[Step 5/9] Check Existing Container:**

- Defines `EXPECTED_ORIGINS` string based on detected `$SERVER_IP` and defined `$WEBUI_PORT`.
- Checks if container `$OLLAMA_CONTAINER_NAME` exists (running or stopped) using `docker ps -a`.
- If exists: Inspects its `OLLAMA_ORIGINS` environment variable.
- If running AND `OLLAMA_ORIGINS` matches `EXPECTED_ORIGINS`, prints message and exits successfully (idempotent).
- Otherwise (stopped or wrong CORS): Stops (`docker stop || true`) and removes (`docker rm`) the container (using `$INVOKING_USER` if possible, else root).

7. **[Step 6/9] Run Ollama Container:**

- Prints parameters being used.
- Executes `docker run -d` command (using `sudo -u "$INVOKING_USER"` if possible, else root) with all parameters: `--name`, `-p`, `-v`, `-e OLLAMA_ORIGINS`, `--restart unless-stopped`, image name.

8. **[Step 7/9] Verify Container is Running:**

- Pauses for 5 seconds (`sleep 5`).
- Checks if container is listed in `docker ps` (trying as `$INVOKING_USER` first, then root).
- If running: Prints confirmation. Attempts `curl http://localhost:$OLLAMA_PORT/` and reports success or warning.
- If not running: Prints error message including `docker logs` command and exits with status 1.

9. **[Step 8/9] Configure Firewall (UFW):**

- Checks if UFW is active.
- If active: Checks if the specific rule (`${OLLAMA_PORT}/tcp` with comment "Ollama API Access") already exists using `sudo ufw status | grep`.
- If rule doesn't exist, adds it: `sudo ufw allow ${OLLAMA_PORT}/tcp comment "Ollama API Access"`.
- If rule exists, reports that.
- Prints verbose status filtered for the port.
- If inactive, prints warning.

10. **[Step 9/9] Final Instructions:**

- Prints a success banner.

- Summarizes the running state, API endpoint, CORS origins, and model directory.
- Provides example commands for accessing the WebUI (from Script 3a) and for pulling/listing models using docker exec.

11. **Exit:** Exits with status 0 on success.

- **Verification:**

- Check script execution logs (/ai-data/BuildLogs/2-Setup-Ollama-Container.log, /ai-data/BuildLogs/breakdown_2-Setup-Ollama-Container.txt).
- Check container status: docker ps (look for ollama-server, status "Up", port 11434).
- Check container logs: docker logs ollama-server (look for successful startup messages).
- Verify CORS setting: docker inspect ollama-server --format='{{range .Config.Env}}{{println .}}{{end}}' | grep OLLAMA_ORIGINS (should match EXPECTED_ORIGINS from script output).
- Verify volume mount: Check contents of /ai-data/ollama-models on the host after pulling a model (docker exec -it ollama-server ollama pull tinyllama). Files should appear there.
- Check firewall rule: sudo ufw status verbose (verify 11434/tcp (Ollama API Access) is allowed).
- Test API endpoint locally: curl http://localhost:11434/api/tags (should return JSON).
- Test API endpoint remotely (if needed, from WebUI server eventually): curl http://<server_ip>:11434/api/tags.

- **Key Files/Directories Created/Modified:**

- /ai-data/ollama-models/ (Directory potentially created, ownership set, populated by Ollama)
- Docker container (ollama-server) created.
- Docker image (ollama/ollama) pulled.
- UFW configuration updated (if UFW was active).

- **Integration Points:**

- Provides the core LLM API service on port 11434.
- Critically configured via OLLAMA_ORIGINS to allow access from Script 3a's WebUI.
- Uses /ai-data/ollama-models for persistent storage, accessible via Samba (Script 1c) or locally.
- Relies on Docker and user permissions set up by Script 1a.

- **Potential Issues & Troubleshooting:**

- **User Permissions Error (Step 2):** User running script hasn't logged out/in after being added to docker group by Script 1a. Script provides clear instructions.
- **Container Fails to Start (Step 7):** Check docker logs ollama-server. Common causes: Port 11434 already in use by another process; issues with Docker daemon; problems mounting /ai-data/ollama-models (e.g.,

underlying filesystem issues, though script sets ownership); insufficient system resources (less likely for Ollama base server).

- **API Curl Check Fails (Step 7):** Container might still be initializing (wait longer?), or it crashed immediately after starting. Check logs. Ensure localhost resolves correctly.
 - **WebUI Connection Fails (Later):** Double-check the OLLAMA_ORIGINS value in the running container (docker inspect). Ensure \$SERVER_IP detected by Script 2 matches the IP the WebUI is trying to connect *from* (if WebUI runs on a different machine, its origin needs to be added). Check firewall rules (sudo ufw status). Check browser's developer console (F12) in the WebUI for specific CORS errors.
 - **Image Pull Failure (Step 4):** Network issues, Docker Hub registry problems, or incorrect image name.
 - **IP Detection Fails (Start):** If hostname -I doesn't give the expected primary IP, OLLAMA_ORIGINS might be set incorrectly, causing CORS issues later. Manually edit the script or container environment if necessary.
 - **Rollback/Cleanup:**
 - Stop the container: docker stop ollama-server
 - Remove the container: docker rm ollama-server
 - Remove the image (optional): docker rmi ollama/ollama
 - Remove the firewall rule: sudo ufw delete allow 11434/tcp
 - Remove the model data (use with caution!): sudo rm -rf /ai-data/ollama-models
-

Script 3a: 3a-setup_ollama_webui_basic.sh

(Revision based on actual script)

High-Level Documentation

- **Purpose:** To deploy a very basic, functional web interface for interacting with the Ollama API (provided by Script 2). This UI allows selecting an available model, entering a prompt, and viewing the response.
- **Key Actions:**
 - Performs safety checks (requires root via sudo, exits on error). Identifies \$INVOKING_USER who will own and run the web service.
 - Defines configuration: Web directory (/ai-data/webui/ollama-basic), UI port (8080), Python server script name (serve.py), systemd service name (ollama-webui-basic.service).
 - Determines the Ollama API URL (\$OLLAMA_API_URL) based on detected host IP and standard Ollama port.
 - Creates a self-documentation block in /ai-data/BuildLogs/breakdown_3a-setup_ollama_webui_basic.txt.
 - Creates the web directory (\$WEBUI_DIR) and sets ownership to \$INVOKING_USER.

- Generates index.html containing the basic UI structure (model dropdown, prompt textarea, send button, response area) and basic CSS styling.
- Generates script.js which:
 - Fetches available models from `${OLLAMA_API_URL}/api/tags` on page load using fetch.
 - Populates the `<select>` dropdown with model names.
 - Handles errors during model loading (e.g., Ollama unreachable).
 - Adds an event listener to the send button.
 - On click, sends the selected model and prompt text to `${OLLAMA_API_URL}/api/generate` via a POST request (stream: false).
 - Displays the response text (or error messages) in the designated output area.
 - Manages button/input disabled states during requests.
- Generates a Python script (serve.py) using http.server and socketserver to serve static files from `$WEBUI_DIR` on the specified `$WEBUI_PORT` (8080). Includes basic error handling for port conflicts and prints the access URL. Includes function to detect local IP for access URL printout.
- Makes the Python script executable (`chmod +x`).
- Creates a systemd unit file (`/etc/systemd/system/ollama-webui-basic.service`) to:
 - Run the Python server (`ExecStart=/usr/bin/python3 $WEBUI_DIR/$PYTHON_SERVER_SCRIPT`).
 - Run as the specified user (`User=$INVOKING_USER`).
 - Set the working directory (`WorkingDirectory=$WEBUI_DIR`).
 - Configure automatic restart on failure (`Restart=on-failure`).
 - Specify dependencies (`After=network.target ollama-server.service`).
- Reloads the systemd daemon, enables, and starts the new service (`$SYSTEMD_SERVICE_NAME`).
- Waits 2 seconds and verifies the service is active using `systemctl is-active`. Exits with error if service failed.
- Configures UFW: Allows traffic on port 8080/tcp with a comment ("Basic Ollama WebUI") if UFW is active.
- Provides the final URL for accessing the WebUI.
- **Dependencies:**
 - 1a-ServerOS-Base-Build-Script.sh (provides base OS, Python3, UFW, user context).
 - 2-Setup-Ollama-Container.sh (provides the running Ollama container and API endpoint which this UI connects to). Ollama container (ollama-server) must be running.
- **Outcome:** A simple web server running as a systemd service (ollama-webui-basic.service) under the `$INVOKING_USER` account, serving a basic HTML/JS interface on port 8080. This interface can list models from and send prompts to the local Ollama container. Firewall allows access to the UI port.

Detailed Runbook Documentation

- **Objective:** To provide a minimal, functional web interface for the Ollama service, running reliably via systemd as a non-root user.
- **Prerequisites:**
 - Scripts 1a and 2 completed successfully.
 - The Ollama container (ollama-server as defined in Script 2) must be running and accessible via `http://<server_ip>:11434`.
 - Python 3 must be installed (handled by Script 1a).
 - The script must be run using sudo by the user designated to run the web service (\$INVOKING_USER).

- **Execution:**

1. Save the script code to a file named `3a-setup_ollama_webui_basic.sh`.
2. Make the script executable: `chmod +x 3a-setup_ollama_webui_basic.sh`.
3. Run the script using sudo:

```
sudo ./3a-setup_ollama_webui_basic.sh | sudo tee /ai-data/BuildLogs/3a-setup_ollama_webui_basic.log
```

content_copydownload

Use code [with caution](#).Bash

- Captures output to the log file.
- Also creates `/ai-data/BuildLogs/breakdown_3a-setup_ollama_webui_basic.txt`.

- **Step-by-Step Breakdown:**

1. **Configuration & Safety Checks:** Sets variables, set -e, checks for root, identifies \$INVOKING_USER, determines \$OLLAMA_API_URL.
2. **[Step 1/9] Write Documentation Log:** Ensures log dir exists, creates self-doc file, sets ownership.
3. **[Step 2/9] Create Web Directory:** Creates \$WEBUI_DIR, sets ownership recursively to \$INVOKING_USER.
4. **[Step 3/9] Create index.html:** Uses sudo tee and heredoc to write HTML structure and CSS to \$WEBUI_DIR/index.html. Sets ownership.
5. **[Step 4/9] Create script.js:** Uses sudo tee and heredoc to write JavaScript code to \$WEBUI_DIR/script.js. **Crucially embeds the \$OLLAMA_API_URL variable into the JS code.** Sets ownership.
6. **[Step 5/9] Create Python Server Script:** Uses sudo tee and heredoc to write the Python http.server code to \$WEBUI_DIR/\$PYTHON_SERVER_SCRIPT. Sets ownership and executable permission (sudo chmod +x).
7. **[Step 6/9] Create systemd Service File:** Uses sudo tee and heredoc to write the unit file definition to `/etc/systemd/system/$SYSTEMD_SERVICE_NAME`, specifying User, WorkingDirectory, ExecStart, Restart, and After directives.

8. **[Step 7/9] Enable and Start Service:** Runs `sudo systemctl daemon-reload`, `sudo systemctl enable $SYSTEMD_SERVICE_NAME`, `sudo systemctl start $SYSTEMD_SERVICE_NAME`.
9. **[Step 8/9] Verify Service and Configure Firewall:**
 - Pauses (sleep 2). Checks service status (`systemctl is-active`). If failed, prints error, attempts stop/disable, and exits(1).
 - Checks UFW status. If active, adds rule `sudo ufw allow ${WEBUI_PORT}/tcp` comment "Basic Ollama WebUI". Prints confirmation. If inactive, prints warning.
10. **[Step 9/9] Final Instructions:** Prints success banner and the URL (`http://${OLLAMA_HOST_IP}:${WEBUI_PORT}`) to access the UI.
11. **Exit:** Exits with status 0 on success.
 - **Verification:**
 - Check script execution logs (`/ai-data/BuildLogs/3a-setup_ollama_webui_basic.log`, `/ai-data/BuildLogs/breakdown_3a-setup_ollama_webui_basic.txt`).
 - Check service status: `sudo systemctl status $SYSTEMD_SERVICE_NAME` (should be active/running).
 - Check service logs: `journalctl -u $SYSTEMD_SERVICE_NAME -n 50` (should show the Python server starting and potentially access logs).
 - Check file ownership: `ls -l $WEBUI_DIR` (should be owned by `$INVOKING_USER`).
 - Check firewall: `sudo ufw status` (verify port 8080/tcp (Basic Ollama WebUI) is allowed).
 - **Test UI:** Open the provided URL (`http://<server_ip>:8080`) in a web browser.
 - Verify the model dropdown populates (may take a second). If not, check browser console (F12) for errors connecting to Ollama API (`/api/tags`).
 - Select a model, enter a prompt, click Send. Verify the response appears in the output area. If errors occur, check browser console and Ollama container logs (`docker logs ollama-server`).
 - **Key Files/Directories Created/Modified:**
 - `/ai-data/webui/ollama-basic/` (Directory and contents: `index.html`, `script.js`, `serve.py` created, ownership set)
 - `/etc/systemd/system/ollama-webui-basic.service` (Systemd unit file created)
 - UFW configuration updated (if UFW was active).
 - Systemd service links for `ollama-webui-basic.service` enabled.
 - **Integration Points:**
 - Provides a User Interface (listening on port 8080) for the Ollama backend.
 - Directly depends on the Ollama API (Script 2) being available at the configured URL (`$OLLAMA_API_URL`). Correct CORS settings in Script 2 are essential for the fetch calls in `script.js` to work from the browser.

- Runs as a systemd service managed by the OS, started after the Ollama container (due to After= directive).
- **Potential Issues & Troubleshooting:**
 - **Service Fails to Start (Step 8):** Check `sudo systemctl status $SYSTEMD_SERVICE_NAME` and `journalctl -u $SYSTEMD_SERVICE_NAME`. Common causes: Port 8080 already in use (Python script error); Python3 not found (`/usr/bin/python3` path incorrect); syntax error in `serve.py`; permissions issue running as `$INVOKING_USER` in `$WEBUI_DIR`.
 - **UI Loads, Models Don't:** Browser console (F12) likely shows errors fetching `/api/tags`. Causes: Ollama container (Script 2) not running; incorrect `$OLLAMA_API_URL` embedded in `script.js` (IP detection failed?); network issue between browser and Ollama API; **CORS issue** (Ollama's `OLLAMA_ORIGINS` in Script 2 doesn't include `http://<server_ip>:8080` or `http://localhost:8080`).
 - **Models Load, Sending Prompt Fails:** Browser console shows errors fetching `/api/generate`. Similar causes as above (Ollama down, network, CORS), or an issue with the prompt/model itself (check Ollama logs: `docker logs ollama-server`).
 - **UI Not Accessible:** Firewall blocking port 8080 (check `sudo ufw status`). Service not running (check `systemctl status`). Incorrect IP/port used in browser URL.
 - **Python Script Errors:** Check `journalctl -u $SYSTEMD_SERVICE_NAME`. Could be syntax errors, import errors (though this script uses standard libraries), or permission errors accessing `$WEBUI_DIR`.
- **Rollback/Cleanup:**
 - Stop the service: `sudo systemctl stop $SYSTEMD_SERVICE_NAME`
 - Disable the service: `sudo systemctl disable $SYSTEMD_SERVICE_NAME`
 - Remove the service file: `sudo rm /etc/systemd/system/$SYSTEMD_SERVICE_NAME`
 - Reload systemd: `sudo systemctl daemon-reload`
 - Remove the firewall rule: `sudo ufw delete allow 8080/tcp`
 - Remove the WebUI files: `sudo rm -rf /ai-data/webui/ollama-basic`

Script 4: 4-Setup-ComfyUI-Container.sh

(Revision based on actual script - Assuming the actual script matches the previous documentation accurately, as the code wasn't provided again)

High-Level Documentation

- **Purpose:** To build a custom Docker image for ComfyUI (configured for CPU mode) and run it as a container.
- **Key Actions:** Creates necessary host directories, dynamically generates a Dockerfile tailored for CPU operation, builds the Docker image using the generated Dockerfile, runs the ComfyUI container mapping the required port (8188) and mounting data volumes (`/ai-data/comfyui-data`), and opens the firewall port.

- **Dependencies:** Script 1a (provides Docker, /ai-data directory).
- **Outcome:** A running ComfyUI container accessible via a web browser on port 8188, using CPU for processing, with persistent storage for models and generated images. **Acknowledged limitations regarding performance and model loading on POC hardware.**

Detailed Runbook Documentation

- **Objective:** Deploy ComfyUI in a containerized manner, specifically building it for CPU execution due to hardware constraints, and ensuring data persistence.
- **Prerequisites:**
 - Script 1a completed successfully (Docker installed and running, /ai-data exists).
 - Root or sudo access (or user in docker group).
 - Internet connection (to clone ComfyUI repo and download dependencies during build).
 - UFW managed by the system.
 - Git installed (likely handled by Script 1a).

- **Execution:**

- # If user is in docker group:

- `bash ./4-Setup-ComfyUI-Container.sh | tee -a /ai-data/BuildLogs/4-Setup-ComfyUI-Container.log`

- # If user is NOT in docker group or script needs sudo for UFW:

```
sudo bash ./4-Setup-ComfyUI-Container.sh | tee -a /ai-data/BuildLogs/4-Setup-ComfyUI-Container.log
```

```
content_copydownload
```

Use code [with caution](#).Bash

- **Step-by-Step Breakdown:** (Based on previous analysis, assuming script performs these actions)

1. **Create Data Directories:** Ensures the target volume mount points exist on the host (/ai-data/comfyui-data/input, /output, /models/checkpoints, etc.).

2. **Generate Dockerfile:** Creates a Dockerfile dynamically (e.g., in /tmp/ or /ai-data/BuildLogs). The Dockerfile includes steps to:

- Use a base Python image.
- Install OS dependencies (git, wget, etc.).
- Clone the ComfyUI repository.
- Install Python requirements (requirements.txt).
- Install **CPU-specific PyTorch** (--extra-index-url .../cpu).
- Expose port 8188.

- Set the CMD to run main.py with --listen 0.0.0.0 --port 8188 --cpu.
3. **Build Custom ComfyUI Image:** Executes `docker build -t comfyui-custom-cpu` using the generated Dockerfile.
 4. **Stop/Remove Existing Container:** Stops and removes a container named `comfyui-server-custom` (or similar name used in the script) if it exists.
 5. **Run ComfyUI Container:** Starts the container using `docker run -d` with:
 - `--name comfyui-server-custom` (or name used in script).
 - Port mapping: `-p 8188:8188`.
 - Volume mounts: `-v /ai-data/comfyui-data/...:/app/...` mapping input, output, and models directories.
 - Restart policy: `--restart unless-stopped`.
 - Image: `comfyui-custom-cpu`.
 6. **Configure Firewall:** Opens port 8188/tcp in UFW.
 7. **Output Status:** Prints confirmation messages and the access URL. Notes performance limitations.
- **Verification:**
 - Check build logs for errors during docker build.
 - Check container status: `docker ps` (look for `comfyui-server-custom`, status "Up", port 8188).
 - Check container logs: `docker logs comfyui-server-custom` (look for ComfyUI startup messages, confirmation it's using CPU).
 - Check firewall rule: `sudo ufw status` (verify 8188/tcp is allowed).
 - Access the ComfyUI URL (`http://<server_ip>:8188`) in a browser.
 - Verify data persistence: Place a model in `/ai-data/comfyui-data/models/checkpoints`, refresh ComfyUI, see if it appears. Generate an image, check `/ai-data/comfyui-data/output`.
 - **Key Files/Directories Created/Modified:**
 - `/ai-data/comfyui-data/` (Directory structure created/used on host)
 - Temporary Dockerfile created during script execution.
 - `comfyui-custom-cpu` (Docker image created)
 - UFW configuration updated.
 - Docker container (`comfyui-server-custom`) created.
 - **Integration Points:**
 - Provides the ComfyUI service on port 8188.

- Uses /ai-data/comfyui-data for persistent storage. Requires manual model placement via Samba (Script 1c) or other methods.
 - **Potential Issues & Troubleshooting:**
 - **Dockerfile Build Failure:** Check build logs. Missing dependencies, network issues, conflicts. Ensure correct CPU PyTorch version is specified.
 - **Container Fails to Start:** Check docker logs comfyui-server-custom. Port conflicts, volume mount issues, errors in CMD, insufficient RAM.
 - **ComfyUI Slow/Unusable: Expected** in CPU mode on POC hardware. This is a known limitation.
 - **Large Models Fail:** RAM limitation (8GB on POC hardware).
 - **ComfyUI Manager Missing:** Requires manual installation post-script (e.g., docker exec) or modification of the Dockerfile build process.
 - **Rollback/Cleanup:**
 - Stop container: docker stop comfyui-server-custom
 - Remove container: docker rm comfyui-server-custom
 - Remove image: docker rmi comfyui-custom-cpu
 - Remove firewall rule: sudo ufw delete allow 8188/tcp
 - Remove data: sudo rm -rf /ai-data/comfyui-data (Use caution!)
-

Script 5: 5-Setup-LMStudio-Container.sh (Deferred)

High-Level Documentation

- **Purpose:** (Intended) To set up and run LM Studio in a containerized environment.
- **Status: Deferred.** No functional script produced during the POC phase.
- **Reason for Deferral:** Lack of readily available, reliable public headless Docker images, complexity of building one from the AppImage, and functional overlap with Ollama for GGUF model serving reduced priority for the POC.

Detailed Runbook Documentation

- **Objective:** Deploy LM Studio server via Docker.
- **Status: Deferred / Not Implemented.**

(No further runbook details as the script was not implemented)

Script 6: 6-Setup-PiperTTS-Container.sh (Deferred)

High-Level Documentation

- **Purpose:** (Intended) To set up and run a Piper Text-to-Speech (TTS) server in a container.
- **Status: Deferred.**
- **Reason for Deferral:** Attempts using available public Docker images failed on the POC hardware (Pentium E6600) due to the CPU **lacking the required AVX instruction set**. Building from source to specifically target non-AVX was deemed too complex for the POC scope.

Detailed Runbook Documentation

- **Objective:** Deploy a Piper TTS server via Docker.
- **Status: Deferred / Not Implemented.**

(No further runbook details as the script was not implemented)

Script 7: 7-Setup-Dashboard-Container-NodeJS.sh

(Revision based on actual script)

High-Level Documentation

- **Purpose:** To set up and run a custom-built dashboard application for the Samaritan AI Server. This dashboard, built with Node.js and Express, runs inside a Docker container managed by Docker Compose. It provides links to AI services and displays system health and Docker container status fetched via backend API calls. This replaces the previous attempt using the 'Homepage' dashboard.
- **Key Actions:**
 - Performs safety checks (requires root via sudo, exits on error). Identifies \$INVOKING_USER.
 - Defines configuration: Dashboard base directory (/ai-data/dashboard), web file subdirectory (web), dashboard port (3000), relevant container names, and service ports/IPs.
 - Creates a self-documentation block in the logs.
 - Checks prerequisites: Docker, Docker Compose plugin, Docker service status, curl. (Node.js is *not* required on the host).
 - Creates dashboard directories (\$DASHBOARD_DIR, \$DASHBOARD_WEB_DIR/css, \$DASHBOARD_WEB_DIR/js) and sets ownership.
 - Generates dashboard files:
 - web/index.html: Structure with sections for Services, System Info, Tools. Includes placeholders for status badges, system metrics, Docker status, and an API status indicator for Ollama. Disables non-functional buttons.
 - web/css/style.css: Basic dark theme styling, including classes for online/offline status badges, API indicator colors, and disabled buttons.

- `web/js/dashboard.js`: Frontend logic to fetch data from backend API endpoints (`/api/system-metrics`, `/api/docker-status`, `/api/service-status/...`, `/api/service-status/ollama-api`) using `fetch`, update the HTML content dynamically, set intervals for refreshing data, and update service links.
- `app.js`: Node.js/Express backend server. Serves static files from `./web`. Provides API endpoints:
 - `/api/system-metrics`: Runs shell commands (`top`, `free`, `df`, `uptime`) to gather CPU, Memory, Disk, Uptime, Load Average.
 - `/api/docker-status`: Runs `docker ps -a` and filters output to show status of relevant containers (Ollama, ComfyUI, Dashboard).
 - `/api/service-status/:service`: Checks if the specified service container (Ollama, ComfyUI) is running using `docker ps --filter`.
 - `/api/service-status/ollama-api`: Uses `curl` to check if the Ollama API endpoint (`http://localhost:11434/api/version`) responds successfully.
- `package.json`: Defines Node.js project metadata and dependencies (`express`).
- `Dockerfile`: Defines the build steps for the dashboard container: uses `node:18-alpine` base, installs dependencies (`npm install`), copies app code, installs `curl`, `bash`, and `docker-cli` within the container, exposes port 3000, and sets CMD `["node", "app.js"]`.
- Generates `docker-compose.yml`: Defines a dashboard service, builds the image from the local Dockerfile, sets container name `samaritan-dashboard`, restart policy, maps host port `$DASHBOARD_PORT` to container port 3000, and **mounts the Docker socket** (`/var/run/docker.sock`) read-only into the container.
- Builds the Docker image using `docker compose build --no-cache`.
- Stops and removes potentially conflicting previous dashboard containers (`samaritan-dashboard`, `homepage-dashboard`, `docker-socket-proxy`).
- Starts the new dashboard container using `docker compose up -d`.
- Waits 10 seconds and verifies the `samaritan-dashboard` container is running. Performs a local `curl` check against the dashboard URL. Exits if container fails to start.
- Configures UFW: Allows traffic on port `$DASHBOARD_PORT/tcp` with a comment ("Dashboard Access") if UFW is active.
- Provides the final URL to access the dashboard.
- **Dependencies:**
 - `1a-ServerOS-Base-Build-Script.sh` (provides Docker, Docker Compose Plugin, UFW, user context, `curl`).
 - Requires network access during the docker compose build step to download the Node.js base image and npm packages.
 - Relies on the Docker daemon socket (`/var/run/docker.sock`) being accessible to the container (handled by the volume mount).

- **Outcome:** A running Docker container (samaritan-dashboard) serving a custom Node.js web application on port 3000. The dashboard displays system metrics, status of key Docker containers, Ollama API status, and provides links to services. Firewall allows access to the dashboard port.

Detailed Runbook Documentation

- **Objective:** To deploy a custom, containerized Node.js dashboard application using Docker Compose, providing monitoring and access points for the Samaritan AI services.

- **Prerequisites:**

- Script 1a completed successfully.
- Docker service is running.
- The script must be run using sudo.
- Internet connection required for docker compose build.

- **Execution:**

1. Save the script code to 7-Setup-Dashboard-Container-NodeJS.sh.
2. Make executable: `chmod +x 7-Setup-Dashboard-Container-NodeJS.sh`.
3. Run with sudo:

```
sudo ./7-Setup-Dashboard-Container-NodeJS.sh | sudo tee /ai-data/BuildLogs/7-Setup-Dashboard-Container-NodeJS.log
```

content_copydownload

Use code [with caution](#).Bash

- Captures output. Creates breakdown log file.

- **Step-by-Step Breakdown:**

1. **Config & Safety:** Sets variables, set -e, root check, \$INVOKING_USER.
2. **[Step 1/10] Write Docs:** Creates breakdown log file.
3. **[Step 2/10] Check Prereqs:** Verifies docker, docker compose version, docker service status, and curl command.
4. **[Step 3/10] Create Dirs:** Creates \$DASHBOARD_DIR, \$DASHBOARD_WEB_DIR/css, \$DASHBOARD_WEB_DIR/js. Sets ownership on \$DASHBOARD_DIR.
5. **[Step 4/10] Create Files:** Uses sudo tee and heredocs to generate index.html, css/style.css, js/dashboard.js, app.js, Dockerfile, package.json inside \$DASHBOARD_DIR and its subdirectories. Sets ownership on \$DASHBOARD_DIR again (covers new files).
6. **[Step 5/10] Create Compose File:** Uses sudo tee to generate docker-compose.yml in \$DASHBOARD_DIR, defining the dashboard service, build context, container name, restart policy, port mapping, and **docker socket volume mount**. Sets ownership.
7. **[Step 6/10] Build Image:** Changes directory to \$DASHBOARD_DIR and runs docker compose build --no-cache.

8. **[Step 7/10] Stop Existing:** Runs `docker stop` and `docker rm` for `samaritan-dashboard`, `homepage-dashboard`, `docker-socket-proxy`, suppressing errors if they don't exist (`|| true`).
 9. **[Step 8/10] Start Container:** Changes directory to `$DASHBOARD_DIR` and runs `docker compose up -d`.
 10. **[Step 9/10] Verify Container:** Waits (sleep 10). Checks `docker ps` for `samaritan-dashboard`. If not running, prints error with log commands and exits(1). If running, performs curl check to `http://localhost:3000` and reports success or warning.
 11. **[Step 10/10] Configure Firewall:** Checks UFW status. If active and rule doesn't exist, adds rule `sudo ufw allow ${DASHBOARD_PORT}/tcp` comment "Dashboard Access". Prints confirmation.
 12. **Final Instructions:** Prints success banner, dashboard URL, app location, and notes it's managed by Compose.
 13. **Exit:** Exits 0 on success.
- **Verification:**
 - Check script logs.
 - Verify container is running: `docker ps` (look for `samaritan-dashboard`).
 - Check container logs: `docker logs samaritan-dashboard`. Check compose logs: `cd /ai-data/dashboard && docker compose logs`.
 - Check firewall: `sudo ufw status verbose`.
 - **Test UI:** Access `http://<server_ip>:3000` in a browser.
 - Verify layout loads.
 - Verify System Metrics widget populates with data.
 - Verify Docker Status widget populates and shows Ollama, ComfyUI, and Dashboard containers with correct Up/Exited status.
 - Verify Ollama and ComfyUI service cards show correct "Online"/"Offline" status badges based on container state.
 - Verify the dot indicator next to "Ollama Chat" turns green if the Ollama API responds, red/grey otherwise.
 - Clicking service "Launch" links should go to the correct URLs.
 - Non-functional buttons should appear disabled.
 - **Key Files/Directories Created/Modified:**
 - `/ai-data/dashboard/` (Directory created)
 - `/ai-data/dashboard/web/` (Subdirectory created)
 - `/ai-data/dashboard/web/css/style.css` (File created)
 - `/ai-data/dashboard/web/js/dashboard.js` (File created)

- /ai-data/dashboard/web/index.html (File created)
- /ai-data/dashboard/app.js (File created)
- /ai-data/dashboard/package.json (File created)
- /ai-data/dashboard/Dockerfile (File created)
- /ai-data/dashboard/docker-compose.yml (File created)
- Docker image built (likely tagged dashboard-dashboard or similar by compose).
- Docker container samaritan-dashboard created.
- UFW configuration updated (if active).
- **Integration Points:**
 - Provides central dashboard UI on port 3000.
 - Integrates with Docker via the mounted socket (/var/run/docker.sock) to query container status from the Node.js backend (app.js). Requires docker-cli inside the container.
 - Integrates with Ollama API via direct curl check from app.js to assess API health.
 - Provides links to other services (Ollama WebUI, ComfyUI).
 - Deployment and lifecycle managed by Docker Compose.
- **Potential Issues & Troubleshooting:**
 - **Docker Build Fails (Step 6):** Check build output (docker compose build). Errors in Dockerfile syntax; apk add failures (network, package not found); npm install failures (network, package.json issues). Ensure host has internet access.
 - **Container Fails to Start (Step 9):** Check logs (docker logs samaritan-dashboard, docker compose logs). Node.js (app.js) crashing: syntax errors, unhandled exceptions, required modules missing (if npm install failed silently), internal port conflicts (less likely). Problems accessing Docker socket from within container (permissions, socket not mounted correctly in docker-compose.yml). docker-cli missing in container if apk add failed in Dockerfile.
 - **UI Loads, Widgets Blank/Error:** Check browser console (F12) for API call errors (/api/...). Check Node.js container logs (docker logs samaritan-dashboard) for backend errors processing those API requests.
 - *System Metrics:* Commands (top, free, etc.) not found in container (ensure base image or apk add provides them), parsing errors in app.js.
 - *Docker Status/Service Status:* docker ps command fails inside container (socket permission, docker-cli missing), errors parsing docker ps output in app.js.
 - *Ollama API Status:* curl command fails inside container (curl missing, network issue *between containers* - dashboard needs to reach localhost:11434 which maps to host's Ollama port), Ollama API actually down, timeout.

- **UI Not Accessible:** Firewall (UFW). Container samaritan-dashboard not running. Port mapping incorrect in docker-compose.yml.
 - **Rollback/Cleanup:**
 - Stop and remove container/network: `cd /ai-data/dashboard && sudo docker compose down -v` (the -v removes anonymous volumes, unlikely here but good practice).
 - Remove the custom image (optional): `docker rmi $(docker images -q --filter=reference='*dashboard*')` (Adjust filter as needed based on image name created by compose).
 - Remove the firewall rule: `sudo ufw delete allow 3000/tcp`.
 - Remove the dashboard files: `sudo rm -rf /ai-data/dashboard`.
-

6. Component Interactions & Data Flow

- **User <-> Web UIs:**
 - Dashboard: `http://<server_ip>:3000`
 - Basic Ollama WebUI: `http://<server_ip>:8080`
 - ComfyUI: `http://<server_ip>:8188`
- **Dashboard UI (Browser JS) <-> Dashboard Backend (NodeJS Container :3000):** Frontend JS fetches data from backend API endpoints (`/api/*`).
- **Dashboard Backend <-> Docker Daemon:** Backend Node.js runs `docker ps` via mounted `/var/run/docker.sock`.
- **Dashboard Backend <-> Host System Commands:** Backend Node.js runs `top`, `free`, `df`, `uptime`.
- **Dashboard Backend <-> Ollama API (Health Check):** Backend Node.js runs `curl` against `http://localhost:11434` (host's mapped port).
- **Ollama WebUI (Browser JS) <-> Ollama API (Ollama Container :11434):** Direct fetch calls, enabled by `OLLAMA_ORIGINS` CORS header set in the Ollama container.
- **Ollama Container <-> Host Storage:** `/root/.ollama` mounted to `/ai-data/ollama-models`.
- **ComfyUI Container <-> Host Storage:** `/app/*` directories mounted to `/ai-data/comfyui-data/*`.
- **User (SMB Client) <-> Samba Service <-> Host Storage:** Network access to `/ai-data` and user homes via Samba shares.
- **Admin <-> SSH:** Access via port 22 (UFW allowed, Fail2ban monitored).

Conceptual Diagram Description:

(Imagine boxes and arrows as described previously, but reflecting the NodeJS dashboard and its interactions)

- **Boxes:** User Browser, Server Host (Ubuntu, Systemd, Python Server:8080, Docker Daemon, Samba, UFW/Fail2ban, `/ai-data` FS), Docker Containers (ollama-server:11434, comfyui-server-custom:8188, samaritan-dashboard:3000 with `docker.sock` mount).

- **Arrows:** Show user access to ports 3000, 8080, 8188. Show JS communication (Browser->Backend for Dashboard, Browser->Ollama API for Ollama UI). Show Dashboard Backend communication (->Docker Socket, ->Host Commands, ->Ollama API :11434). Show container volume mounts to /ai-data. Show Samba connection.

7. Operational Considerations

7.1. Data Persistence and Management

- **Central Location:** All persistent data is stored under /ai-data on the host.
- **Ollama Models:** /ai-data/ollama-models. Manage via Ollama commands (docker exec) or Samba.
- **ComfyUI Data:** /ai-data/comfyui-data (models, input, output). Manage models via Samba.
- **Ollama WebUI Code:** /ai-data/webui/ollama-basic. Served by Python service. Modify files directly to change UI.
- **Dashboard Code & Config:** /ai-data/dashboard. Contains app code, Docker config (Dockerfile, docker-compose.yml). Changes require rebuild/restart (docker compose build, docker compose up -d).
- **Build Logs:** /ai-data/BuildLogs. Contains script execution logs.
- **Primary Management:** Samba share (\\<server_ip>\ai-data) provides convenient access for model management and viewing configs/logs.

7.2. User Access Points

- **Dashboard:** http://<server_ip>:3000
- **Ollama Basic UI:** http://<server_ip>:8080
- **ComfyUI:** http://<server_ip>:8188
- **Ollama API (Direct):** http://<server_ip>:11434 (Mainly for programmatic use/debug)
- **Samba Share:** \\<server_ip>\ai-data (Requires Samba user/pass set in Script 1c)
- **SSH:** ssh <user>@<server_ip> (Port 22)

7.3. Logging

- **Script Execution:** /ai-data/BuildLogs/*.log & /ai-data/BuildLogs/breakdown_*.txt
- **Ollama Container:** docker logs ollama-server
- **ComfyUI Container:** docker logs comfyui-server-custom
- **Dashboard Container:** docker logs samaritan-dashboard
- **Ollama WebUI Service:** sudo journalctl -u ollama-webui-basic.service
- **Samba Service:** Logs in /var/log/samba/
- **System Logs:** journalctl, /var/log/syslog, /var/log/auth.log

7.4. Monitoring

- **Primary:** The custom **Dashboard UI** (Port 3000) provides system health, Docker status, and service status indicators.
- **Terminal (Containers):** ctop command.
- **Terminal (System):** htop, glances commands.

7.5. Security

- **Firewall (UFW):** Enabled, default deny incoming, explicitly allows ports 22 (SSH), 137, 138, 139, 445 (Samba), 11434 (Ollama API), 8080 (Ollama UI), 8188 (ComfyUI), 3000 (Dashboard).
- **SSH Protection (Fail2ban):** Monitors SSH logs and bans IPs attempting brute-force logins.
- **Docker Socket Access:** Mounted read-only into the Dashboard container. **Consider replacing with a Docker Socket Proxy for enhanced security in non-POC environments.**
- **Samba Authentication:** Configured for security = user, requires valid Linux user + Samba password.
- **Service User:** Ollama WebUI runs as non-root (\$INVOKING_USER). Containerized services run as defined in their respective Docker images (often root unless specified otherwise).

7.6. Management

- **Initial Setup:** Sequential execution of BASH scripts (sudo ./...).
- **Container Control:**
 - Ollama, ComfyUI: docker stop/start/restart <container_name>
 - Dashboard: cd /ai-data/dashboard && docker compose down/up -d
- **Service Control (Ollama UI):** sudo systemctl status/stop/start/restart ollama-webui-basic.service
- **Updates:** Requires re-running relevant scripts (potentially after pulling updated images or code). For custom builds (ComfyUI, Dashboard), rebuilds are necessary (docker build, docker compose build).

8. Key Learnings & Future Considerations

- **CORS is Critical:** Explicitly setting OLLAMA_ORIGINS was essential for the browser-based UI (Script 3a) to interact with the backend API (Script 2).
- **Custom Builds Add Control:** Building custom Docker images (ComfyUI CPU, NodeJS Dashboard) provided necessary customization but required more effort. The pivot from 'Homepage' to the NodeJS dashboard highlighted the value of direct control for specific monitoring needs.
- **Hardware Matters:** CPU features (AVX) and resources (RAM, GPU) are major factors in AI application compatibility and performance. This POC clearly demonstrated limitations.
- **Modularity Works:** The sequential, single-purpose script approach proved effective for building and debugging. Verification steps (Script 1b) are valuable.
- **Dashboard Choice:** Switching from Homepage to a custom Node.js dashboard provided the necessary control to implement reliable Docker status monitoring via the mounted socket for this POC.

- **Future:**
 - Implement a **Docker Socket Proxy** for the dashboard instead of direct socket mounting for improved security.
 - Explore alternative **TTS solutions** compatible with older hardware or implement build-from-source for Piper if needed.
 - Add **GPU support** to ComfyUI (Script 4) and potentially Ollama (Script 2) via modified Dockerfiles/run commands if hardware is upgraded.
 - **Enhance the NodeJS dashboard:** Add more metrics, container controls (start/stop buttons), log viewing capabilities.
 - Refine error handling and user feedback in scripts and UIs.
 - Consider parameterizing scripts further (e.g., ports, base directories).