SBSeg 2026 Armação dos Búzios - RJ
VEJA OS SELOS DE ARTEFATOS

AdminForge: Declarative Privileged-Identity Management for Linux Server Fleets

Rui de Quadros Ribeiro[1,2], **Cristhian Kapelinski**[1], **Diego Kreutz**[1]

[1]AI Horizon Labs | PPGES, UNIPAMPA [2]CPD, UFRGS
{ruiribeiro.aluno, cristhianavila.aluno, diegokreutz}@unipampa.edu.br

Abstract. *Stolen credentials are a leading cause of security breaches, yet on Linux server fleets the accounts, SSH keys, and permissions behind them are still often managed by hand. Centralized tools (Teleport, Boundary, FreeIPA) add a network-facing service that itself must be protected, and configuration-management tools have no notion of granting and revoking access. We present AdminForge, an open-source command-line tool that manages accounts, keys, and permissions from one operator machine: the operator edits the declared state, previews the changes, and applies them over SSH, recording every action in a local history and installing nothing on the hosts. Five professional Linux administrators completed its nine-task workflow without training, rating usefulness, ease of use and intention to use at a median of* ***6/7****. A parallel apply, verify or audit over 50 hosts finishes in tens of seconds, and an unchanged re-apply answers locally in* ***under 0.3 s****, faster than an Ansible re-run and with no listening service added.*

1. Introduction

Unix-like systems run 91.9% of the websites whose server operating system is known,[1] so the accounts and keys that guard administrative access to Linux servers protect a large part of production infrastructure. Verizon's 2026 Data Breach Investigations Report finds stolen credentials involved in 36% of breaches,[2] and IBM puts the average cost of a breach that starts with compromised credentials at USD 4.67 million.[3] On a Linux fleet, these credentials are the user accounts, SSH keys, and access permissions (shell and `sudo`) that administrators often manage by hand, server by server. Doing it by hand produces stale privileges, inconsistent access and incomplete auditing. These are exactly the problems that least privilege, economy of mechanism (keeping the design small and simple) [Saltzer and Schroeder 1975] and modern access controls [Joint Task Force 2020] exist to prevent. These pain points are documented: SSH keys go unmanaged at scale [Ylonen 2019], infrastructure-as-code scripts carry security smells like hard-coded credentials [Rahman et al. 2019], and operators struggle with access-control misconfigurations [Dietrich et al. 2018]. This paper presents AdminForge, a tool that manages these accounts, keys and permissions from a single operator machine.

Three kinds of tools address this problem, and each has a built-in cost. Centralized identity platforms (FreeIPA,[4] OpenLDAP,[5] Kerberos,[6] Active Directory[7]) simplify authentication and authorization through shared identity repositories, but require a resident directory service and bind every host to it. Privileged access management (PAM) systems (Teleport,[8] Boundary,[9] CyberArk,[10] JumpServer,[11] BeyondTrust,[12] Delinea[13]) add governance and auditing for administrative access through a network-facing controller and

[1] W3Techs, Usage statistics of operating systems for websites, July 2026 [2] Verizon DBIR, 2026 [3] IBM Security, Cost of a Data Breach Report 2025 [4] FreeIPA Project [5] OpenLDAP Documentation [6] MIT Kerberos [7] Microsoft Active Directory Domain Services [8] Teleport Documentation [9] Boundary Documentation [10] CyberArk Privileged Access Manager [11] JumpServer [12] BeyondTrust Documentation [13] Delinea Documentation

proxy that must themselves be protected. Configuration-management tools (Ansible,[14] Salt,[15] Puppet,[16] Chef,[17] CFEngine[18]) treat access as just one part of general automation. Ansible needs no agent, while Salt, Puppet, Chef, and CFEngine run an agent on each host by default. None of them has a built-in notion of grants, sudo profiles, and revocation, nor a tamper-evident record of who granted what to whom: keeping the playbooks in Git tracks edits to files, not grant and revocation actions tied to an operator. The open problem, then, is to manage accounts, keys, and permissions across a Linux fleet from one place, with an auditable record of who granted what, without adding a service that itself has to be protected.

To close this gap we built *AdminForge*,[19] an open-source command-line tool for declarative privileged-identity management on Linux server fleets. The operator edits the declared (desired) access state, made of users, servers, their groups, sudo profiles, and the permissions between them, and then pushes it to the hosts over SSH with an explicit command; Section 3 shows the workflow as typed at the terminal. The design keeps things simple and the attack surface small by avoiding resident components and third-party libraries. Because security mechanisms depend on users correctly performing operational tasks [Whitten and Tygar 1999, Cranor and Garfinkel 2005], we evaluated the tool in a usability study with five professional Linux administrators, each completing the full nine-task workflow without prior training (Section 5).

AdminForge borrows from two of these families and departs from the third: a declared model of the access state, as in configuration management; a record of who did what, as in PAM; and, unlike identity platforms and PAM systems, no resident service on the hosts. Table 1 makes this positioning concrete against representative tools of each family, along five dimensions: authentication, the absence of a resident network service, declarative state, an auditable record of grants, and revocation.

Table 1. AdminForge against representative privileged-access tools.

Tool	Authentication	No resident network service	Declarative state	Auditable grants	Revocation
FreeIPA[4]	Kerberos, LDAP	✗	✗	✓	Directory edit
Teleport[8]	Short-lived certificates	✗	(✓)†	✓	Certificate expiry, locks
Boundary[9]	Credentials issued per session	✗	(✓)†	✓	Grant revocation, session cancel
Ansible[14]	SSH keys	✓	✓	✗	Manual edit
AdminForge (this work)	**SSH keys**	✓	✓	✓	**Key removal on `apply`**

† Declarative roles, but enforced by a runtime controller; no offline reconciliation of a declared state.

AdminForge is the only tool in Table 1 that meets the three middle columns at once. The identity and PAM tools keep an audit trail and, in part, a declarative model, but only through a network-facing service that must itself be run and protected. Ansible needs no such service and is declarative, but records task runs, not grants: neither its logs nor the playbooks' Git history capture who was granted or revoked what access.

We make three contributions: (i) AdminForge itself,[19] released as open source: client-side operation, no resident network-facing service on managed hosts, a local operation history, and a standard-library-only implementation (Sections 2 and 3); (ii) a qualitative positioning against representative tools along five dimensions (Table 1); and (iii) a

[14] Ansible Documentation [15] Salt Documentation [16] Puppet Documentation [17] Chef Documentation [18] CFEngine Documentation [19] `https://doi.org/10.5281/zenodo.22664962`

two-part evaluation: a usability study with five professional administrators (Section 5) and a performance evaluation on a Docker fleet showing a constant per-host cost, a no-change apply faster than Ansible, and no added runtime attack surface (Section 6).

2. The AdminForge Architecture

AdminForge manages a fleet's users, keys, and permissions from one operator machine, with no central directory and no resident service. Its target is a fleet of up to about a thousand Linux hosts administered by one operator; it does not aim to replace a full identity platform, to broker interactive sessions, or to govern what a user does after being granted access. The design choices in this section are properties of the released code.

Conceptual model. The model has eight entities: *User* (any managed account, privileged or not), *SSH Credential* (a key registered for a user), *User Group*, *Server*, *Server Group*, *Permission*, *Sudo Profile*, and *History*. Access exists only when a permission links a user group to a server group; there is no direct link from a user to a server, which keeps policies reusable. A permission grants shell access or `sudo`, and the `sudo` case can be limited to a set of commands through a reusable *sudo profile*. The tool separates its operator (recorded as `superadmin` in the history) from the users it manages, and records only the operator's actions. Figure 1 lists the operator's use cases; each is one CLI command, and each command that changes state adds one history entry. Appendix A shows the full data model.

Design principles. We aimed for a simple tool with no permanent infrastructure and little code. Following the *economy of mechanism* principle [Saltzer and Schroeder 1975], AdminForge runs from the command line, stores its data in plain files, and uses no resident service or database. The first version used the usual libraries: paramiko for SSH, click for the command line, PyYAML for storage, and cryptography. Counting the lines of code that actually run (the tool's own source plus every library it loads, ignoring blank lines and comments), that version came to roughly 58,000 lines. Dropping every external library for the standard library instead — `subprocess` (calling the system `ssh`), `argparse`, and `json` — brings the tool to about 3,900 lines by the same count, over 90% less code running. Shell completion is an optional add-on (`argcomplete`, about 2,200 more lines when installed).

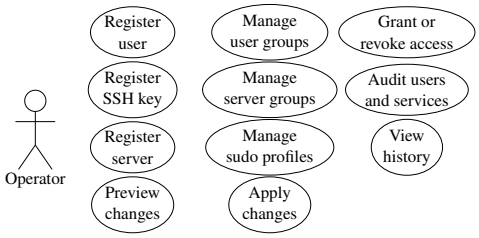


Figure 1. Use cases.

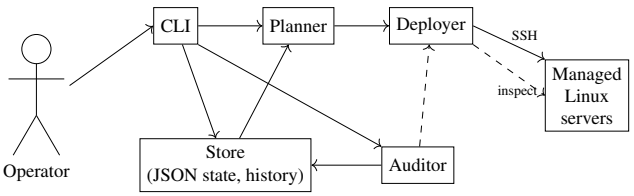


Figure 2. AdminForge architecture.

Figure 2 shows the five parts and how one operation moves between them.

Operator side. The operator drives everything through the command-line interface (*CLI*), the single entry point. It parses each command with `argparse`, offers per-command help and shell completion, and, because every change goes through it, lets the Auditor keep a complete log. The *Store* holds the declared state and the history as plain JSON files on the operator's machine, written under a file lock with restrictive permissions; plain files are easy to read and to version, and avoid a database.

Planning and deployment. The *Planner* compares the declared state with what was last applied and computes only the difference, combining overlapping grants into the highest access each user has on each server. `preview` shows that difference; `apply` installs it, changing only what is needed. The *Deployer* carries that difference to each host over SSH, using the system `ssh` client. It creates missing accounts and updates each user's `authorized_keys` and `sudoers`, sending public keys only and pinning each host's key on first contact, so hosts need no agent. It records per host what it installed: an interrupted run is reported as partial, and re-running `apply` retries only the missing hosts. Applying the same state twice changes nothing. Editing the configuration changes only the local state; it reaches the hosts only on `apply`, optionally after `preview`.

Heterogeneous fleets. A managed host needs only OpenSSH, `sudo`, and the standard account tools: the Deployer creates accounts with `useradd`, reads users and groups with `getent`, and validates every sudoers drop-in on the host with `visudo -cf` before installing it under `/etc/sudoers.d/`. No package manager or interpreter is invoked on the host, so the same commands serve Debian, Red Hat and SUSE derivatives; the audit guards its service listing with `command -v systemctl`, so hosts without systemd still return the rest of the report. Two assumptions are distribution-specific: accounts are created with `useradd -m -s /bin/bash`, and keys are written under `/home/`*`user`*`/.ssh` rather than at the home directory the host reports. We validated Debian hosts only.

Audit. The *Auditor* keeps a local history. Each command becomes one entry (operator, timestamp, result, affected servers), linked by a SHA-256 hash chain [Schneier and Kelsey 1999] that `history verify` checks. For an audit it inspects a host's users, sudo rules, and running services, read-only, over the same SSH path. Because the history lives only on the operator's machine, it is a structured log, not an integrity guarantee if that machine is fully taken over (Section 4). Appendix A traces one `apply` end to end.

3. Usage

AdminForge installs as a normal Python package and provides the `adminforge` command, also called `af`. Every command has `--help`, and the shell can complete commands, flags, and names (completion is an optional add-on; the base install uses only the standard library). The commands below follow the study scenario of Section 5: adding a user and her public key, adding the three lab servers (`--auto` records the host key on first connection; `web-02` and `db-03` follow the same pattern), and putting them into groups (`app` holds the web hosts, `db` holds `db-03`).

```
af user add --username alice --name "Alice Souza" --key-file alice.pub --email "a@co.com"
af server add --hostname web-01 --ip 10.0.0.10 --auto
af user-group create --name sre
af user-group add-member --group sre --username alice
af server-group create --name app
af server-group create --name db
```

Access is granted from a user group to a server group, at level `shell` or `sudo`; a reusable sudo profile limits a grant to named commands. Nothing has reached the servers yet: `preview` shows the pending changes, `apply` sends them over SSH after confirmation, and `apply verify` checks the declared state against the real `authorized_keys` and `sudoers` on each host.

```
af permission grant --user-group sre --server-group app --level sudo
```

```
af sudo-profile create --name db-maint --command /bin/journalctl
af permission grant --user-group sre --server-group db --level sudo --profile db-maint
af preview && af apply && af apply verify
```

Offboarding, investigation, and checking close the workflow, in that order. `af user disable alice` followed by `af apply` removes her keys from every host; `af audit server` reads users, groups, `sudoers`, and services over read-only SSH to reveal changes made outside the tool. Finally, `af history verify` checks the history, reporting `chain intact (last hash: 3f9c...)` or the first altered entry.

4. Threat Model

We consider three adversaries: an *external attacker* with no valid credentials; a *malicious or compromised managed user* holding a legitimate grant; and a *compromised operator workstation*, where the declared state, SSH keys, and history reside. The operator is trusted. We assume the declared state and history sit on an operator-controlled host with the restrictive permissions the tool sets, that SSH distributes public keys only, and that host keys are pinned on first use.

Because every change goes through one declared state, drift on a single server (a host whose real state no longer matches the declared one) and manual key edits show up as differences from that state; per-command sudo profiles limit what a misbehaving user can run; the history records who granted what; and no service that listens on the network is added to the hosts (Table 1). Two limits follow. The tool controls who is *granted* access, not what they do afterward, which is left to the operating system. The local history records changes but cannot survive a full takeover of the operator's machine; simple measures at deployment (copying entries off the machine, append-only file flags, hardening the machine) reduce this risk without adding any service to the hosts.

5. Usability Evaluation

We evaluated AdminForge in a moderated laboratory study with five professional Linux administrators, organized around three research questions. **RQ1**: can experienced administrators learn AdminForge's operational model and complete its core workflow without prior training? **RQ2**: which parts of the model cause difficulty or risk of operational error? **RQ3**: how do administrators rate the tool's usefulness, ease of use, and the security and confidence it conveys, and do they intend to use it?

Every session followed one narrative scenario on the same laboratory infrastructure of three managed Linux servers administered from an AdminForge workstation. Acting as the operator, the participant performed nine tasks in a fixed order, one row each in Table 4 (Appendix A): the full workflow, from registering a user and her SSH key to auditing a divergence we planted outside the tool and inspecting the history. A task counted as completed when the participant reached its success criterion in the facilitator guide (for the audit, reporting the unmanaged `sudoers` entry it flags). Graduated verbal hints were allowed and logged, with no step run by the facilitator.

Sessions were individual and moderated. Participants received no prior training, only the scenario and the tool's built-in help, and thought aloud [Nielsen 1993]. All five signed an informed-consent form; responses are pseudonymized (P1–P5) and the replication package contains no personal data. After each task the participant rated confidence and ease on 7-point scales. After the last task they answered a single questionnaire, also

on 7-point scales: four items on perceived usefulness, four on perceived ease of use, three on intention to use [Davis 1989, Davis et al. 1989], and four on security and confidence.

Five IT professionals with prior Linux administration experience took part: three with more than seven years of experience, one with three to seven years, and one with one to three years. Every participant rated familiarity with command-line administration and `sudo` at 4 or 5 on the 1–5 profile scale, and reported lower familiarity with declarative infrastructure tools, matching the population the study targets. Access to this specialized population is a documented obstacle in security research, reported for system operators [Dietrich et al. 2018] and in the usability evaluation of Certbot, whose authors state that recruiting enough professional administrators was not feasible [Tiefenau et al. 2019].

Given the ordinal scales and small sample, we favor medians over means [Boone and Boone 2012, Jamieson 2004]. Construct statistics pool each construct's item-level responses across participants (perceived usefulness, perceived ease of use, security and confidence: 20 responses each; intention to use: 15); we report the median, the interquartile range (IQR), the top-box rate (share of responses of 6 or 7), the mean, and the standard deviation (SD). Per-task statistics use the five responses to that task. The full replication package ships in the artifact.[19]

Task performance (RQ1). All five participants completed all nine tasks, and reported high confidence overall, with a median of 7 in seven of the nine tasks. Table 4 (Appendix A) reports the ratings per task.

Sources of difficulty (RQ2). Difficulty concentrated on two tasks. Restricted `sudo` configuration had the lowest ease (median 5, mean 4.6), mainly because participants struggled to discover the *sudo profile* abstraction. The audit task had the lowest confidence (median 5, mean 5.0). The audit's alerts section surfaces the planted divergence, and reporting it was the task's success criterion. Two open responses explain the low confidence: one participant did not find the audit command at first, and another ran the audit but was unsure whether the flagged `sudoers` entry was outside the tool's control.

Acceptance (RQ3). Perceived usefulness, perceived ease of use, and intention to use all reached a median of 6, with a top-box rate between 60% and 73%; Table 2 reports the pooled construct results. Security and confidence also reached a median of 6, although with greater dispersion. Open responses highlighted the usefulness of the help system and the separation between configuration and deployment, while suggesting better discoverability of advanced features.

Table 2. Aggregated construct results (1–7 scale); Top-box: share of responses of 6 or 7.

Construct	Median	IQR	Top-box (%)	Mean	SD
Perceived usefulness	6	3.8–7.0	60	5.40	1.76
Perceived ease of use	6	4.5–7.0	70	5.55	1.61
Intention to use	6	4.5–7.0	73	5.47	1.77
Security and confidence	6	3.0–7.0	65	5.30	1.81

Discussion. Participants understood AdminForge without training. Adding users and servers, grouping them, and granting or revoking access all scored high on confidence and ease, and separating definition from application gave a sense of control, since changes could be reviewed before reaching the servers. The RQ2 difficulties concentrated on restricted `sudo` and auditing, both central to the tool: participants struggled to discover the right command and interpret its output, pointing to better contextual help and audit presentation. A related gap between confidence in outcomes and perceived protection

against dangerous errors points to clearer communication of a command's impact in high-privilege operations.

Threats to validity. With only five participants, the study cannot claim statistical generalization, and its findings should be read as evidence about this tool with this population rather than a population-level claim. Small expert cohorts are the norm here: a study of operators' security misconfigurations at ACM CCS used six interviews [Dietrich et al. 2018], and a USENIX Security study of HTTPS deployment interviewed seven auditors [Krombholz et al. 2017], both because access to this population is a documented obstacle [Kotulic and Clark 2004]. The laboratory setting is a second limitation, since production adds operational pressure, organizational processes, and infrastructure heterogeneity. The questionnaire adapts the Technology Acceptance Model [Davis 1989] rather than a validated instrument, and the sample is too small for scale-reliability statistics, so the ratings should be read together with the qualitative observations. Finally, the study covers only initial use, not long-term adoption.

6. Performance Evaluation

Administrative operations run a few times per day or week, not many times per minute, so speed was never a design goal: we chose security, ease of use, and a small dependency footprint instead (Section 2). We therefore ask only whether the tool is fast enough for how it is used and whether its cost grows predictably with the fleet.

Everything ran on one machine (AMD Ryzen 5 8600G, 6 cores, 32 GB RAM, Linux 6.17, Docker 29.4). The fleet consists of Debian containers running only `sshd`, each standing in for a managed host reached over SSH. Fleet and tool share the machine, so every container's account creation runs on the operator's CPU; on a real fleet that work runs on each host and only network latency is added, so these figures are, if anything, a heavy case. Fleet sizes are $N \in \{1, 5, 10, 25, 50\}$; each declared state has 10 users in 2 groups and one sudo profile with access to the whole fleet, over 5 repetitions on fresh containers.

Cold apply is the first `apply` to a fresh fleet; it opens an SSH session to each host, creates the accounts, and writes the keys and sudo rules, so it does the most work. *No-change apply* is a second `apply` right after, when nothing has changed; it is resolved locally and contacts no host. *Verify* (`apply verify`) reads each host's `authorized_keys` and `sudoers` and compares them against the declared state. *Audit* (`audit server`) reads each host's users, groups, sudo rules, and services to reveal changes made outside the tool. The three host-touching operations run one host at a time or several at once with `--jobs`; the no-change apply is always local. Every number in Table 3 is a median over the 5 repetitions, which are stable (cold `apply` min-max spread under 2% at every size). The harness and raw results ship in the artifact.[19]

Table 3. Cost per operation by fleet size, sequential and parallel (`--jobs 25`).

	Cold apply (s)		Verify (s)		Audit (s)		
N	**seq**	**par**	**seq**	**par**	**seq**	**par**	**No-change (s)**
5	70.5	15.2	23.7	5.1	1.2	0.34	0.09
10	141.0	15.9	47.6	5.4	2.4	0.40	0.11
25	350.7	19.6	118.3	6.7	5.9	0.49	0.16
50	703.6	38.7	234.9	13.2	11.6	0.87	0.27

Scalability. Done one host at a time, the three host-touching operations cost a fixed

amount per host, so their total grows with the fleet; cold `apply` about 14 s per host, `verify` about 4.7 s, and `audit` about 0.23 s. Since the hosts are independent, `--jobs` (we used 25) runs several at once, making these almost flat with fleet size; Table 3 reports both modes for $N \geq 5$ (at $N{=}1$ they coincide). At $N{=}50$ the three fall by 13.3 to $18.2\times$ (Table 3), to tens of seconds for a 50-host fleet. Because the per-host cost is constant, the total is predictable at scale; extrapolating 14 s per host, a parallel cold `apply` to a thousand hosts is on the order of ten minutes, bounded by the operator's cores and network, not the tool. Peak memory stays near 20 MiB, because `--jobs` bounds how many hosts run at once.

Comparison with Ansible. We wrote an equivalent Ansible playbook and ran it on the same fleet (ansible core 2.21.2, same SSH key and hosts). At $N{=}50$, with Ansible's 25 parallel connections matched to `--jobs 25`, the first run is comparable (Ansible 59.4 s, AdminForge 38.7 s). The two answer "is anything pending?" differently; Ansible must connect to every host (17.4 s at $N{=}10$, 57.4 s at $N{=}50$), keeping no record of what it applied, while AdminForge holds this record and answers in under 0.3 s without contacting a host. To check the hosts against the declared state, AdminForge's parallel `audit server` and the stricter `apply verify` (0.87 s and 13.2 s at $N{=}50$) still come in below Ansible's re-run. Writing the same state took three Ansible files (playbook, generated inventory, and per-user vars; 78 YAML lines at $N{=}10$) versus 29 AdminForge commands. Two limits bound this comparison: all figures come from a local Docker fleet, not a production network, and Ansible is the only tool measured, being the agentless member of its family (Table 1).

7. Demonstration and Conclusion

Planned demonstration. We start from an empty state, register a user, her SSH key and three servers, grant `sudo` to a group, run `preview` to show the pending changes, then `apply` and `apply verify` against the live hosts. We disable the user and `apply` again, so the audience sees her keys disappear from every host. We close with `af audit server` on a divergence planted outside the tool, and `af history verify`. We need only the presenters' notebook and a projector; the managed hosts are local containers, so no network is required. The tool, replication package, and a demonstration video[20] are available in the open-source repository under the GNU AGPL-3.0 license.[19]

Conclusion. We presented AdminForge, which manages privileged identities on Linux server fleets from a declared state. All five administrators finished the nine tasks and rated usefulness, ease of use, and intention to use at a median of 6/7, while a parallel operation over 50 hosts takes under a minute. Two directions follow. The study's gap between confidence in outcomes and perceived protection against dangerous errors (Section 5) calls for clearer impact warnings before high-privilege operations, a more discoverable sudo profile, and an audit report that separates what the tool manages from what it merely found. The threat model (Section 4) leaves the operator workstation as a single point of trust: it holds the declared state, the keys and the history. N-of-M approval (at least N of M operators must sign off), backed by a Git source of truth, would distribute it. AdminForge shows that a fleet's privileged identities can be governed from one workstation, with an

[20] Demonstration video (installation and features): `https://youtu.be/6rs2qtIuMvs`

attributable record of grants and nothing new to defend on the hosts.[21]

References

Boone, H. N. and Boone, D. A. (2012). Analyzing Likert data. *J. Extension*, 50(2):1–5.

Cranor, L. F. and Garfinkel, S., editors (2005). *Security and Usability*. O'Reilly.

Davis, F. D. (1989). Perceived usefulness, perceived ease of use, and user acceptance of information technology. *MIS Q.*, 13(3):319–340.

Davis, F. D., Bagozzi, R. P., and Warshaw, P. R. (1989). User acceptance of computer technology: A comparison of two theoretical models. *Management Science*, 35(8):982–1003.

Dietrich, C., Krombholz, K., Borgolte, K., and Fiebig, T. (2018). Investigating system operators' perspective on security misconfigurations. In *ACM CCS*.

Jamieson, S. (2004). Likert scales: How to (ab)use them. *Medical Education*, 38(12):1217–1218.

Joint Task Force (2020). Security and privacy controls for information systems and organizations. Technical Report SP 800-53 Rev. 5, NIST.

Kotulic, A. G. and Clark, J. G. (2004). Why there aren't more information security research studies. *Inf. Manag.*, 41(5):597–607.

Krombholz, K., Mayer, W., Schmiedecker, M., and Weippl, E. (2017). "I have no idea what I'm doing" – on the usability of deploying HTTPS. In *USENIX Security Symp.*

Nielsen, J. (1993). *Usability Engineering*. Academic Press.

Rahman, A., Parnin, C., and Williams, L. (2019). The seven sins: Security smells in infrastructure as code scripts. In *41st International Conference on Software Engineering (ICSE)*. IEEE.

Saltzer, J. H. and Schroeder, M. D. (1975). The protection of information in computer systems. *Proc. IEEE*, 63(9):1278–1308.

Schneier, B. and Kelsey, J. (1999). Secure audit logs to support computer forensics. *ACM Trans. Inf. Syst. Secur.*, 2(2):159–176.

Tiefenau, C. et al. (2019). A usability evaluation of Let's Encrypt and Certbot: Usable security done right. In *ACM CCS*.

Whitten, A. and Tygar, J. D. (1999). Why Johnny can't encrypt: A usability evaluation of PGP 5.0. In *USENIX Security Symp.*

Ylonen, T. (2019). SSH key management challenges and requirements. In *10th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*. IEEE.

A. Data Model, `apply` Flow, and Per-Task Ratings

Figure 3 shows the eight entities exactly as the released code defines them. They are stored as plain JSON files, one directory per record type, with each user's SSH credentials nested in its file, permissions in a single file, and the append-only `history.jsonl` carrying the hash chain. Underlined attributes are identifiers. References use natural keys (username, hostname, name) rather than foreign-key IDs, so there are no ID-based joins; the dashed line marks a reference the files do not enforce; history IDs are sequential (OP-nnnn). Figure 4 traces one `apply`, from the operator's command to the history entry. The only step not visible in the diagram is the per-host record of what was installed, which is what lets an interrupted run resume.

[21] **Acknowledgments.** Partially supported by CAPES (Código de Financiamento 001), CNPq (409743/2025-9) and FAPERGS (24/2551-0001368-7, 24/2551-0000726-1). Generative AI tools assisted with writing and language revision; the authors reviewed and take full responsibility for the content.

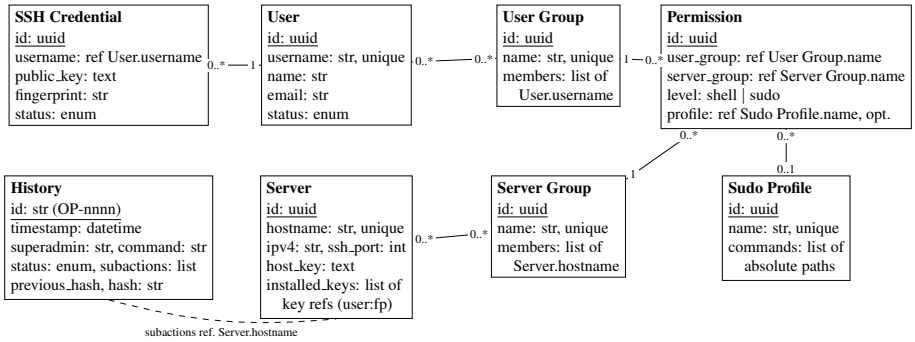


Figure 3. Data model of the declared state and the operation history.

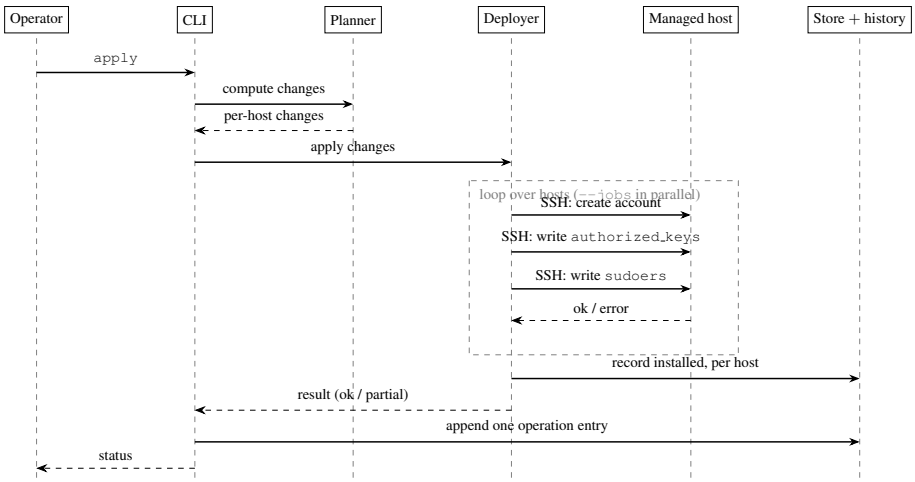


Figure 4. Sequence of one `apply`.

Table 4 completes the study of Section 5 with the median confidence and ease for each of the nine tasks, the mean in parentheses. Its rows follow the fixed order the participants performed them, down the left half and then the right.

Table 4. Median rating per task (1–7 scale), in the order performed; mean in parentheses.

Task	Confidence	Ease	Task	Confidence	Ease
Register user and SSH key	7 (6.6)	6 (6.0)	Configure restricted sudo	7 (6.2)	5 (4.6)
Register servers	7 (6.6)	6 (6.2)	Revoke access	7 (6.4)	6 (5.8)
Organize groups	7 (6.4)	7 (6.6)	Run audit	5 (5.0)	6 (5.0)
Grant access	7 (6.6)	6 (6.4)	Inspect history	7 (6.6)	6 (6.4)
Apply changes	6 (5.8)	6 (6.0)			

B. Parallel Speedup

Table 5 names the speedup behind the 13.3–18.2× range of Section 6: each divides the sequential median of Table 3 by the parallel one at the largest fleet measured, alongside the constant per-host cost. The no-change `apply` is local and has no entry.

Table 5. Parallel speedup per operation at N=50 (`--jobs 25`), from Table 3.

Operation	Sequential (s)	Parallel (s)	Speedup	Per host, seq. (s)
Cold `apply`	703.6	38.7	18.2×	14
`apply verify`	234.9	13.2	17.8×	4.7
`audit server`	11.6	0.87	13.3×	0.23