

WHITEPAPER

Configuration Management at Platform Scale with Ansible

Repository layout, role design, change discipline and pipeline practice for teams that run their own infrastructure across more than one site.

PUBLISHED BY
ZSoftly Technologies Inc.

DATE
September 2026

VERSION
1.0

AUDIENCE
Platform, infrastructure
and operations
engineers

CONTENTS

Executive summary	Page 2
Why configuration as code, and a repository layout	Page 3
Inventories and variable layering	Page 4
Role design rules	Page 5
Change discipline and safe pruning	Page 6
Running from CI, testing and secrets	Page 7
Drift and lessons from production	Page 8
Readiness checklist and how ZSoftly can help	Page 9
Sources	Page 10

Ansible and GitHub Actions facts in this paper were checked against their documentation on 29 September 2026. The practice and lessons come from the Ansible repository that ZSoftly uses to run its own cloud platform. Names, addresses and values are left out on purpose.

EXECUTIVE SUMMARY

Make the repository the only way a change reaches production

Teams that run their own servers, networks and storage accumulate hand-made changes. Each one is small. Together they make a platform nobody can rebuild and nobody can safely change. Ansible fixes that only when the repository is laid out and run with discipline. This paper describes the layout, rules and habits we use, and the incidents that taught us each one.

The problem

- Configuration lives in consoles, shell history and one engineer's memory.
- A variable meant for one site silently applies to another.
- A playbook that removes what it does not declare deletes something another tool owns.
- Check mode reports no changes for tasks it cannot model, so a dry run looks safer than it is.
- Unpinned images and packages upgrade production during an unrelated deploy.

The approach

Keep one repository with shared roles, numbered playbooks and one inventory per site. Put values in layered `group_vars` and secrets in Ansible Vault. Write roles that are idempotent, fail closed and preview their changes. Run them from a pipeline that validates and dry-runs every change, and deploys on an explicit trigger, from a runner inside the target site. Make one change, verify it, then make the next.

One repository

Roles, playbooks, inventories and runbooks in one place.

Scoped by site

Each run loads one site's inventory and uses that site's runner.

Safe by default

Pruning and risky phases are off until named.

Pinned versions

An upgrade is a reviewed diff, never a side effect.

The outcomes to aim for

- Any host can be rebuilt from the repository and a vault password.
- Every change is a pull request with a dry-run result attached.
- Nothing is deleted unless the code owns it and a person asked for the deletion.
- Drift is listed in one place and shrinks over time.

Who should read this

Engineering leaders should read this page, the change discipline on page 6 and the lessons on page 8. **Platform engineers** will find the layout and role rules on pages 3 to 5.

Operations teams should read the pipeline, testing and secrets chapter on page 7. Anyone starting a new repository should work through the checklist on page 9 first.

What this paper is not. It is not an Ansible tutorial. It assumes you can write a playbook and a role. It covers what changes when one repository runs many services across several sites, and many people change it every week.

CHAPTER 1

Why configuration as code, and a repository layout that scales

Cloud providers hide most host configuration behind an API. When you own the hardware, you own every hypervisor, firewall, load balancer, database and DNS server. Each one has settings that matter and a console that lets anyone change them. Configuration as code gives you a record of intent, a review step, a way to rebuild and a way to see what changed. It only works if the repository is the normal path for change, not a document of what someone did last month.

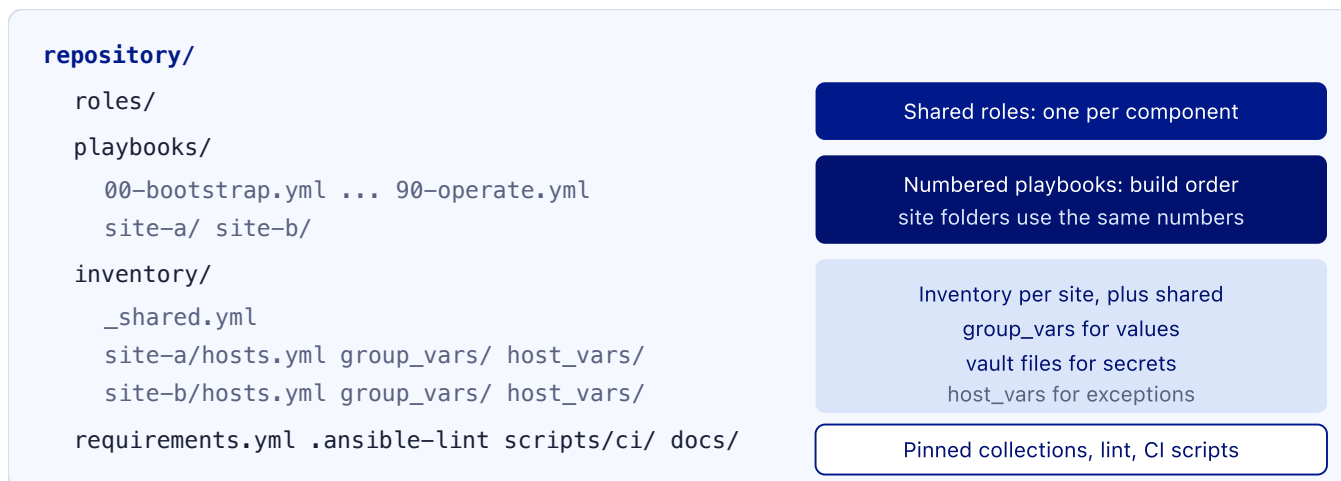


Figure 1. The layout we use. Roles hold behaviour. Playbooks hold order and targeting. Inventories hold every value.

Layout rules

- **Number the playbooks.** A prefix such as `15-provision`, `16-bootstrap`, `85-deploy` shows the build order at a glance. A new site reuses the same numbers in its own folder.
- **Roles hold behaviour, never site values.** Role defaults are safe fallbacks. Extend an existing role before you add a new one.
- **Pin dependencies.** List collections and external roles in `requirements.yml`. Vendor third-party roles you depend on for production.
- **One-time work stays out of the tree.** Renames, repairs and migrations run as a reviewed command sequence recorded in a runbook. Remove any temporary playbook when the change is done.
- **Docs sit next to code.** Each role has a README. Upgrades are one appended row in a version log per service.

What belongs in configuration management

In the repository

- OS baseline, hardening and access
- Service versions and configuration
- Firewall, NAT, load balancer and DNS
- Alert rules, dashboards, platform settings

Kept out, or handled another way

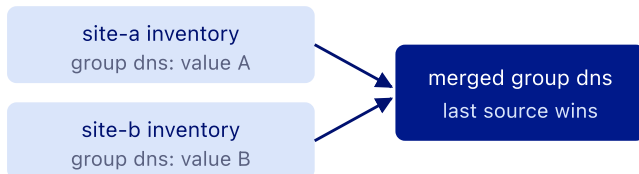
- One-time migrations, kept in runbooks
- Application data and backups
- Secret values in plain text
- Records of intent, such as IPAM

CHAPTER 2

Inventories and variable layering

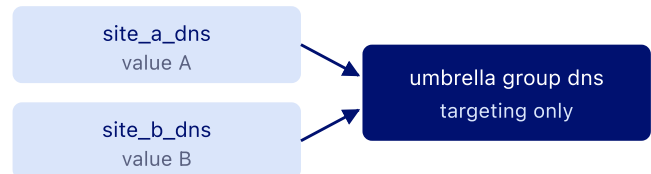
Most Ansible mistakes at scale are variable mistakes. Ansible has 22 levels of variable precedence, and extra vars passed with `-e` always win [4]. Within the inventory, a child group beats its parent and a host beats both. Groups at the same level merge in alphabetical order, and the last one loaded wins [3]. When you pass several inventory sources, the last source wins for a conflicting value [3]. That last rule is the one that bites multi-site teams.

UNSAFE: SHARED GROUP NAME



Every host in dns gets value B, including site A hosts. A host limit does not change it, because precedence follows source order.

SAFE: SITE-PURE GROUPS



Site values live on groups that exist in one site only. The shared name stays for targeting and carries values that are the same everywhere.

Figure 2. The cross-site collision and its fix. The shared group still lets one playbook target every DNS host.

Kind of value	Where it lives	Why
Same everywhere	<code>group_vars/all</code> or a shared group	One place to change, no chance of collision
Differs by site	A group that exists in one site only	Cannot bleed into another site's hosts
Differs by host	<code>host_vars</code> or inline in <code>hosts.yml</code>	Host scope beats every group
Secret	A vault file beside the plain values	Encrypted at rest, diffable by name
Required with no safe default	Role argument, marked mandatory	The run fails before it touches a host

Table 1. Where each kind of value belongs.

After any change to a site value, print the resolved variable for every host in the group and confirm each shows its own site's value. A quick scan of `ansible-inventory --list` for any host holding another site's gateway or address range catches the rest.

Scoping a run

- **A limit narrows hosts, not values.** `--limit` picks which hosts run. It does not change which inventory source supplied a variable.
- **Watch child groups too.** A group that spans sites, placed as a child of one site's group, pulls the other site's hosts into it. They then inherit that site's values. Give each site a wrapper group that holds only its own hosts.
- **Target site groups for cluster work.** Any operation against a cluster, a replicated database or storage names a site-scoped group and a matching admin host, never a shared parent.
- **One site per pipeline job.** A job loads exactly one site's inventory. A change to two sites is two jobs.

CHAPTER 3

Role design rules

Ansible defines an operation as idempotent when running it once gives exactly the same result as running it repeatedly [11]. That is the first rule of every role. A second run that reports changes is a bug, even if nothing broke. The rules below follow from it.

Rule	In practice
Use modules, not shell	Modules know the current state and report real changes. The shell module supports check mode only partly, through <code>creates</code> and <code>removes</code> [2]. Wrap unavoidable commands with those, or with a read step and a <code>changed_when</code>
Templates over inline scripts	Any script longer than a few lines becomes a Jinja template in the role, deployed to the host and called by name. It gets reviewed, linted and versioned like the rest
Restart through handlers	Handlers run only when notified, once per play, however many tasks notify them [6]. Use <code>flush_handlers</code> when a later task needs the restarted service
Fail closed	Check identity before acting: the right cluster, the writable database node, the expected site. Stop unless exactly one target matches
No silent defaults for targets	Variables that choose hosts, clusters or zones have no default. The run fails if the caller forgets them
Reject unknown keys	Validate declared data before writing. A typo in a field name becomes an error, not a setting that is ignored
Back up before editing	Copy any file edited outside a module before the write, so a rollback does not depend on memory
Pin versions	Every image tag and package version is a variable. Upgrades change that variable in a pull request

Table 2. The rules every role in the repository follows.

Modes a role should offer

Precheck only

A flag that runs every read and assertion and stops before the first write. It answers "would this run succeed" without the partial state a failed run leaves behind.

Opt-in operations

Operational phases such as restart, purge, delete or rollback carry the `never` tag. Ansible skips them unless you request them by tag [5].

Confirmed destruction

A phase that deletes data also needs a confirmation variable that names the phase and an explicit target list. The value `all` is refused where it would be dangerous.

Upgrade on request

Package roles install with state `present`, the default of the apt module [8]. A patch run passes `latest` for that run only, so routine deploys never upgrade by accident.

Report in a fixed vocabulary

Have roles print outcomes with a small set of prefixes, such as `[OK]`, `[KEEP]`, `[PRUNE-PLAN]` and `[FAIL]`. Reviewers can then read a dry run in seconds, and a pipeline can search the log for the lines that need a person.

CHAPTER 4

Change discipline and safe pruning

One change, then verify

Run one change against one scope. Check the result on the system itself. Report it. Only then start the next step. This feels slow in a sequence of steps that each look safe. It is how you catch the second-order effect before the second change doubles it. In a cluster, a pause after the first node is what surfaces a quorum or link alarm before the second node joins the problem.

Declared state against live state

Some playbooks enforce a full list: firewall rules, NAT forwards, DNS overrides. They add what is missing and remove what is not declared. Before such a playbook runs for the first time, or after anyone has touched the target by hand, diff the declared list against the live configuration, item by item. Comparing two high-availability peers with each other proves only that they agree. It does not prove that either matches your list.

WHAT A PRUNING ROLE DOES WITH EACH LIVE ITEM

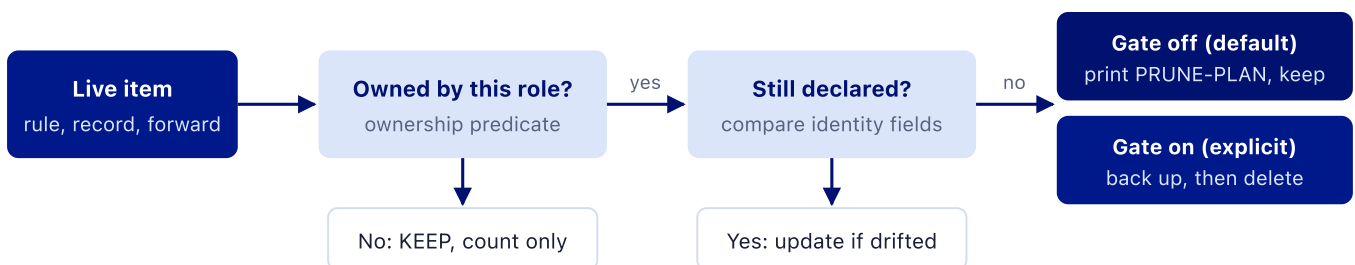


Figure 3. A pruning role can only delete items it owns, and only when a person turns the prune gate on for that run.

Prune gates with an ownership predicate

- **Ownership first.** The role decides which live items it owns, for example by a description prefix or a tag. Everything else is reported as kept and never touched.
- **Prune off by default.** An owned item that is no longer declared is printed as a planned removal. It is removed only when a `prune_unmanaged` style variable is set true for that run.
- **Claim a namespace without rules.** An empty declaration with an ownership match is valid. The role then reports what it finds there, which is how you adopt an existing area safely.

Fixes made by hand must land in code

During an incident, engineers change live systems to restore service. That is correct. The fix is not finished until the same change is in the repository. A runtime rule that exists only in memory disappears at the next reload or sync, and the outage returns with no recent change to blame.

Around every change

- 1 Read the live state of the target with a read-only command and save it.
- 2 Run the playbook in check and diff mode, scoped to one host or one site.
- 3 Apply to that scope only. Watch alerts and the service itself as well as the play recap.
- 4 Confirm the result on the system, then record what changed.
- 5 Stop. Start the next scope only after someone has looked at the first.

CHAPTER 5

Running from CI, testing and secrets

A laptop is a poor control point. Run production playbooks from a pipeline, on self-hosted runners that sit inside each site. One reusable workflow does the work, and one small workflow per service calls it with the playbook, inventory and runner labels.

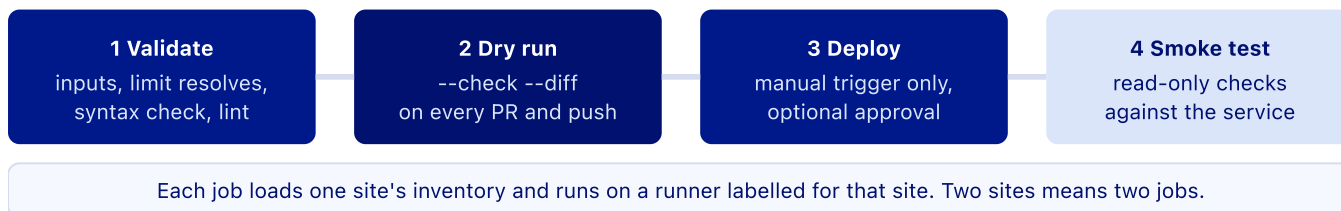


Figure 4. The reusable pipeline. Stages 1 and 2 run automatically. Stage 3 needs a person.

Pipeline rules

- **Route by label.** A self-hosted runner must carry every label a job lists to be eligible [10]. Give each runner a site label and require it in every job, so a job cannot land on the wrong site's runner.
- **Gate deploys.** Environment protection rules can require a named reviewer to approve a job before it runs [12]. Some playbooks, such as those that need root on a firewall, are marked engineer-only and never deploy from CI.
- **Keep scripts out of YAML.** Steps longer than a few lines move to numbered scripts in the repository. Pass inputs through `env:`, which GitHub recommends over inline expressions to block script injection [13].

Testing: what each tool proves

Syntax checks prove the files parse. `ansible-lint` checks content against a profile. We use `production`, the last of its six ordered profiles [9]. Check mode makes no changes, and modules that do not support it report nothing and do nothing [1]. Diff mode shows what a supported module would change [1]. A dry run therefore says nothing about shell tasks or scripts that edit files directly. Treat `changed=0` from such a playbook as unknown, not safe, and diff live state by hand first.

Secrets

Ansible Vault encrypts sensitive values such as passwords, either whole files or single variables [7]. Keep vault files beside the plain `group_vars`, with variable names in the clear so reviews still make sense. The pipeline receives the vault password and an SSH key as repository secrets and loads the key into an agent for the job only. Never pass a secret as a command-line argument, where any user on the host can read it in the process list. Use a file the tool reads instead.

Pipeline failure modes we have seen

- **A job on the wrong site's runner.** Command execution over SSH worked, then module transfers hung for minutes and ended unreachable. A required site label on every job stopped it.
- **A public registry as a hard dependency.** A deploy that pulled images on every run hit a rate limit and stopped halfway, leaving two sites on different configs. With pinned tags, pull only when an image is missing.
- **A reload that reloaded nothing.** A config file mounted into a container as a single file kept its old copy after the template wrote a new one. Verify changes inside the running service.

CHAPTER 6

Drift and lessons from production

Drift is any gap between the repository and the running system. Detect it three ways. Run scheduled dry runs of idempotent playbooks and alert on any reported change. Diff live objects against declared lists for anything a playbook prunes. Keep an unmanaged-configuration register: a table of every setting no playbook owns, with its risk, added when someone changes a console and removed when a role takes ownership. The register is honest about the gap, and the gap shrinks.

What happened	Cause	What we changed
A pruning run deleted forwarding rules and cut a console service for minutes	The playbook removed every rule matching a broad pattern that it did not declare. Two of those rules belonged to another playbook. Peer-to-peer checks passed, and check mode reported no changes because the edit ran in a script	An ownership predicate, a prune gate that defaults off, and a live diff before any first run
One site's hosts used another site's values and a monitoring scrape failed	A site value sat in <code>group_vars</code> of a group name used in both sites. The last inventory source won for every host	Site-pure groups for site values, a resolved-variable check after each edit, one site per CI job
A cluster operation wrote one site's config onto the other site's hosts	The role defaulted its target to a group that spanned both sites	A mandatory target with no default, and a guard that compares cluster identity and fails closed. A backup made the recovery instant
A routine config deploy upgraded production monitoring	Images were tagged <code>latest</code> and the deploy pulled on every run	Every image pinned behind a variable. Upgrades are now a reviewed diff
Hosts reported as patched had not rebooted in weeks	The reboot ran only when the package manager flagged one. Hosts with a held kernel never raised the flag	A force-reboot variable for held hosts, and a report of hosts running stale libraries
A firewall rule added during an incident vanished hours later	It existed only in the running ruleset. The next high-availability sync reloaded the saved config	Incident fixes land in code the same day, on the node that syncs

Table 3. Anonymised lessons from our postmortems. Each fix is now a role change, a guard or a pipeline rule.

What the lessons share

- **Scope is the risk.** Most incidents came from a run touching more than its author meant: a broader pattern, a merged group, a default target.
- **Green is not proof.** A clean check, a successful reload or a patched status each described the tool, not the system.
- **Defaults decide outcomes.** Safe defaults such as prune off, no default target and pinned versions prevent more incidents than careful operators do.
- **Memory is not a system.** Every fix that lived only in a running process or a person's head came back as a second incident.

Write it up, then code it. Each lesson above started as a short postmortem with a timeline, a cause and follow-ups. We close a postmortem only when its fixes are merged and have run once in production.

WORKING WITH US

Readiness checklist and how ZSoftly can help

Check your readiness

- | | |
|--|---|
| ☐ One repository with roles, numbered playbooks and per-site inventories. | ☐ Every pruning role has an ownership predicate and a prune gate. |
| ☐ No site value in a group that spans sites. | ☐ Validate and dry run on every pull request, deploy on a trigger. |
| ☐ Collections, roles, images and packages pinned. | ☐ One site per CI job, on a runner in that site. |
| ☐ Secrets in Vault, never on a command line. | ☐ Lint at a stated profile, enforced in CI. |
| ☐ A second run of every playbook reports no changes. | ☐ An unmanaged-configuration register that someone reviews. |
| ☐ Risky phases tagged <code>never</code> and confirmed by name. | ☐ Postmortem fixes merged before the postmortem closes. |

If more than three boxes are empty, start with an assessment before you add new automation.

How we work

ZSoftly Technologies Inc. is a cloud engineering company. We design, build and run cloud platforms for clients in Canada, the United States and the United Kingdom. The engineers who design your automation are the ones who build it and hand it over.

Assess

Review your repository, inventories, pipelines and recent incidents.

Design

Agree layout, variable rules, pipeline gates and ownership in writing.

Build

Refactor or write roles and pipelines, one service at a time.

Operate

Run changes with your team until the new habits hold.

Hand over

Runbooks and access move to your team, which ships a change unaided.

Where we usually start

- 1 Discovery call.** We review your current automation and goals. Your first hour with our engineers is free.
- 2 Assessment.** A written report on layout, variable scoping, pruning risk, pipeline gaps and drift.
- 3 Pilot.** One service moved to the patterns in this paper, from lint to a gated deploy.
- 4 Roll-out.** Remaining services in priority order, each with a dry run and a rollback.

What you keep

- All roles, playbooks, inventories and pipeline code, in your own repository.
- Runbooks for deploys, upgrades, rollbacks and incident follow-up.
- Vault passwords, runner hosts and keys that belong to you from the first day.

Talk to an engineer. Book a discovery call at zcp.zsoftly.ca/contact or call +1 (647) 771-1065.

REFERENCES

Sources

All sources were checked on 29 September 2026.

1. Ansible. Validating tasks: check mode and diff mode.
https://docs.ansible.com/ansible/latest/playbook_guide/playbooks_checkmode.html
2. Ansible. ansible.builtin.shell module, check mode support.
https://docs.ansible.com/ansible/latest/collections/ansible/builtin/shell_module.html
3. Ansible. How to build your inventory, including how variables are merged.
https://docs.ansible.com/ansible/latest/inventory_guide/intro_inventory.html
4. Ansible. Using variables, including variable precedence.
https://docs.ansible.com/ansible/latest/playbook_guide/playbooks_variables.html
5. Ansible. Tags, including the special tags always and never.
https://docs.ansible.com/ansible/latest/playbook_guide/playbooks_tags.html
6. Ansible. Handlers: running operations on change.
https://docs.ansible.com/ansible/latest/playbook_guide/playbooks_handlers.html
7. Ansible. Protecting sensitive data with Ansible Vault.
https://docs.ansible.com/ansible/latest/vault_guide/index.html
8. Ansible. ansible.builtin.apt module, state parameter.
https://docs.ansible.com/ansible/latest/collections/ansible/builtin/apt_module.html
9. Ansible Lint. Profiles.
<https://docs.ansible.com/projects/lint/profiles/>
10. GitHub. Using self-hosted runners in a workflow.
<https://docs.github.com/en/actions/hosting-your-own-runners/managing-self-hosted-runners/using-self-hosted-runners-in-a-workflow>
11. Ansible. Glossary, idempotency.
https://docs.ansible.com/ansible/latest/reference_appendices/glossary.html
12. GitHub. Managing environments for deployment, required reviewers.
<https://docs.github.com/en/actions/managing-workflow-runs-and-deployments/managing-deployments/managing-environments-for-deployment>
13. GitHub. Security hardening for GitHub Actions, script injection.
<https://docs.github.com/en/actions/security-for-github-actions/security-guides/security-hardening-for-github-actions>

Notices

This paper is for general information. It does not create a warranty or a commitment from ZSoftly Technologies Inc. The lessons describe our own systems with identifying details removed. Ansible is a registered trademark of Red Hat, Inc. GitHub is a trademark of GitHub, Inc. Other names may be trademarks of their respective owners.

© 2026 ZSoftly Technologies Inc. Version 1.0, September 2026.