



SYNAPSE CORTEX

Administrator Guide

Everything a Global Admin or Tenant Admin needs to run Synapse-Cortex v2.

Version **2.0.0** · **Confidential** · 6 July 2026

Synapse IronNode product family

This guide covers everything a **Global Admin** or **Tenant Admin** needs to run Synapse-Cortex v2: initial setup, user management, SLA configuration, MFA enforcement, the NetscanXi asset-import integration, the audit trail, and the new **AI Remediation Module**.

Synapse-Cortex v2 is a React single-page application backed by a pure-JSON FastAPI API (the server-rendered pages from v1 are gone — everything below still describes the same admin workflows, just running in the new SPA).

1. First-Run Setup

The first time Synapse-Cortex starts against an empty database, every page redirects to `/setup` until an admin account exists. There is no default login — you create it yourself.

1. Run `docker compose up --build` from the project root (see the top-level README/`.env.example` for `POSTGRES_*`, `SECRET_KEY`, and the AI module variables covered in Section 9).
2. Visit `http://<host>:8000/`. The SPA calls the `setup-status` endpoint on load and routes you to `/setup` automatically if no admin exists yet.
3. Enter your **Tenant Name**, plus your own name, email, and password. This creates the tenant, seeds default SLA policies for all four priorities, and creates you as the tenant's **Global Admin**. The **AI Remediation Module is disabled by default** for every new tenant — see Section 9 to turn it on.
4. You're immediately logged in and land on the Dashboard.

There is one tenant per Synapse-Cortex deployment in this release; all users, tickets, assets, SLA policies, playbooks, and vaulted credentials belong to it.

2. Roles

Four roles, from most to least privileged:

Role	Can do
<code>global_admin</code>	Everything, including managing other global admins and the AI module toggle
<code>tenant_admin</code>	Everything except manage global admin accounts and the AI module toggle
<code>agent</code>	Work tickets: view/update status, priority, assignment, description; use AI Remediation on tickets
<code>requester</code>	Create tickets, view their own

Only `global_admin` and `tenant_admin` see the **Admin** menu (Users, SLA Policies, Asset Types, Integrations, Audit Trail, Playbooks). Only `global_admin` sees **AI Settings** and **Vault** — the tenant-wide AI on/off switch, the Claude API key, and the credential vault's one-time creation are all deliberately kept out of `tenant_admin`'s reach. A `tenant_admin` cannot create, edit, or delete a `global_admin` account — only another `global_admin` can.

Deleting a ticket is also admin-only: the **Delete Ticket** button on a ticket's detail page only appears for `global_admin` and `tenant_admin`. Deletion is permanent (it also removes that ticket's logged actions and remediation history) and is recorded in the Audit Trail.

3. Managing Users

Admin → **Users** lists every account in the tenant.

- **Create:** click + *New User*, fill in name/email/password/role. Passwords must be at least 8 characters.
- **Change role:** use the inline role dropdown on any row (subject to the role-hierarchy rule above).
- **Enable/disable:** click the status badge to toggle an account on or off without deleting it.
- **Delete:** click *Delete* on any row except your own — you can't delete yourself.
- **Require MFA:** click the MFA badge to toggle "required" for that user. See below for what happens next.

4. Multi-Factor Authentication (MFA)

MFA is TOTP-based (any standard authenticator app — Google Authenticator, Authy, 1Password, etc.).

- **Self-service enrollment:** any user can go to **Settings** → **Profile & MFA** and click *Enable MFA* to scan a QR code and confirm with a 6-digit code.
- **Admin-required MFA:** if you flag a user as "MFA required" from the Users page and they haven't enrolled yet, their **next login redirects straight to the enrollment screen** — they can't reach the dashboard until they finish setting it up.
- **Disabling:** a user can turn MFA off themselves from Settings by re-entering their password. There is no tenant-wide "force MFA for everyone" switch in this release — it's set per user.

5. SLA Policies

Admin → **SLA Policies** shows the four priority tiers (Critical/High/Medium/Low) with their response and resolution time targets in minutes. Defaults are seeded at setup; edit the numbers inline and click *Save* per row. Changes apply to tickets created afterward — existing tickets keep the due dates computed under the old policy.

6. Managing Asset Types

Admin → **Asset Types** controls the list of categories available in the asset **Type** field, both for assets created manually and for assets imported from NetscanXi. Every new tenant starts with seven defaults (Server, Workstation, Laptop, Network Device, Mobile, Printer, Other), but these are just a starting point — rename, add, or remove them to match how your organization actually categorizes hardware.

- **Add a type:** click + *New Type* and give it a name (unique within the tenant).
- **Rename a type:** edit the name inline in the table and click away (or tab out) to save. Any asset already using that type is unaffected — it keeps pointing at the same category, now shown under its new name.
- **Delete a type:** click *Delete*. If any asset currently uses that type, deletion is blocked — reassign those assets to a different type first.

Renaming or adding types also changes what NetscanXi's imports match against: on each import, NetscanXi's free-text device-type guess is matched against your tenant's *current* type names (see Section 8), so keeping familiar defaults like "Server" or "Printer" around helps that matching stay accurate.

7. Audit Trail

Admin → **Audit Trail** is a running log of every meaningful action in the tenant: ticket and asset creates/updates/deletes, ticket actions logged, asset type changes, asset imports, user management, SLA policy changes, logins (success and failure), MFA enrollment/disable, API key generation/revocation, playbook create/update/toggle/delete, AI module toggling, and every remediation investigate/approve/reject. Filter by action type with the dropdown; results are paginated 25 at a time. Asset imports from NetscanXi log **one row per push batch** (not per asset), so a 500-device scan doesn't flood the log — the row's detail shows the created/updated counts for that batch.

8. NetscanXi Integration (Assets + Tickets)

This is the integration that keeps your CMDB and ticket queue in sync with what NetscanXi actually finds on the network. One API key covers two feeds: **asset inventory** (asset ID, MAC, IP, device type, OS, software) and **tickets** (auto-created from NetscanXi's Remediation Tracking items).

How it works

NetscanXi **pushes** to Cortex; Cortex never reaches out to NetscanXi. The asset ID NetscanXi assigns to a device is used **directly** as that asset's ID in Cortex too, so the same device shows the same ID in both tools. (Assets you create manually in Cortex keep the existing `AST-xxxxxxxxxx` ID scheme; only NetscanXi-imported assets get NetscanXi's raw ID.)

Setup steps

1. In Cortex, go to **Admin** → **Integrations**.
2. Click **Generate API Key**. The key is shown exactly once — copy it now along with the **API URL** shown above it (this is the *base* address of the Cortex instance, with no path after it — NetscanXi appends its own ingest paths automatically). If you lose the key, you'll need to generate a new one (the old one stops

working immediately).

3. In NetscanXi, open the **Synapse Cortex** panel (toolbar, tenant-admin only).
4. Paste the API URL and API Key, check **Enable sending asset data to Synapse Cortex**, and click **Save**. This one toggle and key pair governs both feeds below.
5. Click **Test Connection** — you should see "OK Connected". If you see "Failed to Connect", double-check the URL and key (and that Cortex is reachable from NetscanXi's network).
6. Click **Send latest scan now** to push the asset inventory, and/or **Send remediation items as tickets** (further down the same panel) to push tickets. Either can be triggered independently, whenever you like.

What gets imported — Assets

Each pushed asset creates or updates a Cortex Asset record:

- **Asset ID** — NetscanXi's ID, used as-is.
- **MAC / IP** — as discovered. Devices without a MAC (e.g. some Wi-Fi-only or virtual hosts) are still imported; Cortex only enforces "no duplicate MAC per tenant" for assets that actually have one.
- **Asset Type** — NetscanXi's free-text device-type guess is matched on a best-effort basis against your tenant's *current* list of asset types (see Section 6), not a fixed set — so if you've renamed or removed a default category, the match adapts accordingly, falling back to "Other" (or whatever type is available) when nothing matches. The original NetscanXi label is always preserved and shown on the asset's detail page as "as reported by NetscanXi," even if the match guessed wrong.
- **OS and Software** — shown on the asset's detail page under **Discovered OS & Software**, and can be corrected there if needed. This field also feeds the AI Remediation Module's target-OS guardrail (Section 9).

Re-pushing the same scan updates existing asset records in place — it never creates duplicates, matching first on NetscanXi's asset ID and falling back to MAC address. It's also change-aware: if a re-push finds a device whose data hasn't actually changed since last time, that asset is left completely untouched (no write, no "last updated" bump) rather than being marked updated just because a push happened — only assets that are genuinely new or have genuinely changed data count toward the created/updated totals shown in the Audit Trail.

What gets imported — Tickets

Each pushed item from NetscanXi's **Remediation Tracking** module creates a Cortex ticket the first time it's seen:

- **Title / Description** — taken from the remediation item; the CVE (if any) is appended to the title.
- **Priority** — passed straight through (NetscanXi's low/high/medium/critical scale matches Cortex's exactly).
- **Status** — mapped from NetscanXi's remediation status: `open`→New, `in_progress`→In Progress, `blocked`→Awaiting User, `resolved`→Resolved.

- **Asset link** — if the remediation item names an asset NetscanXi has already imported, the ticket is linked to it automatically. A ticket with a linked asset is exactly the kind of ticket the AI Remediation Module is built for (Section 9) — the asset's discovered OS feeds directly into playbook selection and guardrails.

Tickets are matched by a stable reference id (`netscanxi:<tenant>:remediation:<id>`) shown on the ticket's detail page as **External Ref**. **Unlike assets, ticket import is create-only**: once a ticket exists for a given remediation item, later pushes leave it alone entirely — title, status, priority, whatever an agent has since changed in Cortex is never overwritten. If NetscanXi later marks the same item resolved, that does not retroactively update the Cortex ticket; close it out in whichever system is doing the work.

Revoking access

Back in **Admin** → **Integrations**, click **Revoke** to invalidate the key immediately (NetscanXi's next push, of either kind, will get a 401) or uncheck **Enable** in NetscanXi to pause pushing without touching the key.

9. AI Remediation Module

This is new in v2. Claude reviews a ticket's title, description, priority, and linked asset, selects the single best-matching **Playbook** from the ones your tenant has enabled, and proposes it — it never writes or invents commands itself. A human then explicitly approves or rejects the proposal (unless a playbook is specifically configured to skip that click — see *Approval levels* below), and only after approval does a guardrailed executor run the playbook's commands against the ticket's linked credential. Every step — the AI's proposal and the final outcome — is written permanently to the ticket's **Actions Taken** log, on top of the dedicated **AI Remediation** panel on the ticket page.

Setting Up the Credential Vault

Before anything else in this section, the tenant needs a **vault** — the encryption key that protects every vaulted host credential and the Claude API key below. **Admin** → **Vault** (visible only to `global_admin`) is where this is created.

- Click **Create Vault**. This can only be done **once** — there is no button, endpoint, or setting anywhere in the app to recreate or rotate it, because doing so would make every credential and Claude API key already encrypted with it permanently undecryptable.
- The generated key is shown to you **exactly once**, immediately after creation, with a **Copy to Clipboard** button. Store it somewhere safe (a password manager or other secure record) before navigating away — it cannot be displayed again afterward, by anyone, through any part of the app or API.
- Losing that copy doesn't delete anything or break the running deployment (the app itself keeps working with the key it already stored internally), but it does mean that if you ever needed to migrate the deployment's `SECRET_KEY` you'd have no way to re-derive the vault key, since the vault key is wrapped using `SECRET_KEY` at rest (Section 10).
- Once created, the Vault page simply shows **Vault Created** with the creation date — there is deliberately no further action available there.

Credential vaulting (linking a host credential to a ticket, below) and saving a Claude API key both require this vault to exist first; each fails with a clear, specific error pointing back here if it doesn't.

Turning it on

Admin → **AI Settings** (visible only to `global_admin`) has the master kill switch: **AI Remediation Module — Enabled/Disabled**. Turning it off immediately hides the AI panel on every ticket and blocks the investigate endpoint outright, regardless of how many playbooks are configured. It's off by default for every new tenant. The Dashboard shows an **AI Enabled/AI Disabled** badge at the top so anyone can see the current state at a glance.

Turning the module on also requires a Claude API key. The same page has a **Claude API Key** section:

- Paste the key into the field and click **Save**. It's encrypted immediately with the tenant's vault key (so the vault must exist first — see above) and is never shown again anywhere in the UI or API afterward; the page only ever shows whether one is **Configured**.
- Click **Test this key** (before saving) or **Test saved key** (after) to send one minimal request to Claude and confirm it actually works, without waiting to find out during a real investigation. The result and timestamp of the most recent test against the *saved* key are shown under its status.
- **Remove** clears the saved key entirely. The investigate endpoint then falls back to an `ANTHROPIC_API_KEY` environment variable if the deployment has one configured (Section 10), or returns a clear "no Claude API key configured" error if neither is set — never a crash.

Playbooks and Playbook Creation

A **Playbook** is the unit of remediation the AI investigator chooses from: a small, deterministic flowchart of shell commands plus the policy (guardrails) that governs whether it's even allowed to run. Admins build the flowchart; nothing about *what commands actually execute* is ever left to the AI at run time — Claude only ever picks *which already-authored playbook* best fits a ticket (Section 9, *Running an investigation*).

Admin → **Playbooks** lists every playbook in the tenant with its approval level, allowed OS, and enabled/disabled status, alongside a **Template Catalogue** sidebar for starting a new one:

- **+ New Playbook (Blank)** — opens the builder with just the required Start node, for building a playbook from scratch.
- **Template Catalogue** (right-hand sidebar) — a permanent, scrollable list of **72 built-in starter playbooks**, grouped into **Ubuntu / Debian (36)**, **Windows (10)**, **Docker (11)**, **Proxmox (14)**, and **General (1)** — the last being the AI command-runner described in *Running an AI-suggested command* below, covering the most commonly needed remediation commands for each platform. The Ubuntu/Debian set spans service restarts and reloads, disk/log/memory/inode cleanup and largest-file triage, `/boot` and old-kernel cleanup, package updates, DNS resolution and firewall reloads, port freeing, read-only-filesystem remounts, clock-drift/NTP fixes, fail2ban unbans, TLS-certificate renewal, runaway-process kills, account unlocks, an SSH-authentication diagnostic, and read-only triage (server-slowness and OOM-killer history). Windows covers service/spooler/IIS restarts, DNS/network resets, temp/disk cleanup, and event-log checks.

Docker covers container restart/recovery, safe and deep image/space pruning, log truncation, health diagnostics, a "locate a container's Compose project" discovery step, and compose restart/update. Proxmox covers VM start/force-restart/unlock and status/agent checks, LXC container start/force-restart/unlock, storage and cluster/quorum health, on-demand VM backup, management-service restarts (for an unresponsive web UI), and version/repo/update maintenance. (The Proxmox node itself is Debian-based, so the Ubuntu/Debian playbooks also apply to the host.) The Docker container-level playbooks address containers by name via the daemon, so they run from any directory. The Compose playbooks (*Restart a Compose Stack*, *Update a Compose Service*) **auto-locate the project directory at run time** from the container's own Compose labels — so once the AI proposes the fix, it executes end to end with **no human pointing it at a folder**; the *Update* playbook likewise auto-derives the service name from the container. All you supply is the container name (the same edit-me token every other container playbook uses). The standalone *Locate a Container's Compose Project* playbook exposes those same labels as a read-only diagnostic when you want to see a container's project directory and compose-file path yourself. Clicking any entry opens the builder with its name, guardrails, and full node graph already loaded — nothing is saved until you click **Save** there, so a template is only ever a starting point, exactly as editable as anything hand-built. Three of the Ubuntu/Debian entries correspond directly to the worked examples further down this section (*Example playbook workflows*), so what you see on the canvas for those matches what's documented there; the rest follow the identical pattern (a short check → action → confirm chain, or a Condition-branch chain for the one that demonstrates branching).

Either way, the visual builder itself is available to both `global_admin` and `tenant_admin`.

Importing and exporting playbooks

A playbook is just data, so it can be moved between instances as a file — useful for sharing a playbook with a colleague, version-controlling it in git, backing up what you've built, or pre-loading a fresh deployment with a standard set. This happens entirely through the running app: **no container rebuild or restart is involved**, because the browser reads the file and uploads it to the live API exactly like the builder's Save does.

- **Export** — every row in the Playbooks list has an **Export** link that downloads that playbook as a `<name>.json` file. The file contains only the portable fields (name, enabled, graph, guardrails) — none of the instance-specific ones (internal id, tenant, timestamps) — so it re-imports cleanly anywhere.
- **Import from File** — the button at the top of the Playbooks page opens a file picker; choose a `.json` file and its playbook(s) are created immediately in the running tenant. A file may hold a single playbook, an array of them, or a `{ "playbooks": [ ... ] }` wrapper — all three are accepted.

Import validates each playbook independently and never aborts the whole batch:

- a **name clash** with an existing playbook is **skipped** (an existing playbook is never overwritten — rename it in the file, or delete the existing one first, to bring it in);
- an **invalid graph** (no Start node, an unreachable command, a cycle) or a disallowed **Auto-Approve** level is reported as an **error** for that item only;

- everything valid and new is created, and a summary (N created, N skipped, N error(s), with the reason for each non-created item) is shown after the import.

Each import is recorded as a single **Audit Trail** entry (`playbook_import`) with the created/skipped/error counts, mirroring how NetscanXi asset imports are logged.

This is also the recommended way to load a playbook that was authored outside the app: drop the `.json` file somewhere handy (the repo's own `playbooks/` folder is a natural home) and Import it. That folder already ships **every catalogue template as an individual importable .json file** (all 72 — 36 Ubuntu/Debian + 10 Windows + 11 Docker + 14 Proxmox + 1 General), plus a `README.md` documenting the exact file format, so you can pull the whole standard set into a fresh instance without rebuilding.

The builder toolbar

- **Playbook name** (text field) — see *Naming matters* below; this is not just a label.
- **Enabled** (checkbox) — disabled playbooks are invisible to the AI investigator entirely; it structurally cannot select one, even by mistake. Use this to take a playbook out of rotation without losing its graph.
- **+ Command** — adds a new Command node to the canvas.
- **+ Condition** — adds a new Condition node to the canvas.
- **Cancel** — discards changes and returns to the playbook list.
- **Save** — disabled until the playbook has a name. Validates the graph server-side (see *Validation errors* below) before writing anything.

Nodes are connected by dragging from the small circular **handle** at the bottom of one node to the handle at the top of another. A node with no outgoing connection is a dead end; a Command node that no path from Start ever reaches is rejected at Save time (see below) rather than silently ignored.

Node types reference

Node	Purpose	Fields	Connection rules
Start	The single entry point every playbook has. Created automatically — it cannot be deleted, and a playbook can never have more than one.	None.	Exactly one outgoing connection, to whatever should run first.

Command	One shell command to run.	Step label — a short free-text name shown in the proposed plan and execution log (e.g. "Restart nginx"), purely descriptive. Command — the literal shell command string that the executor runs verbatim against the ticket's linked credential. May include the <code>{{username}}</code> / <code>{{ip_address}}</code> / <code>{{password}}</code> placeholder tokens (inserted via the node's own buttons) — see <i>Auto-filling commands from the ticket</i> below.	One incoming connection. At most one outgoing connection (a Command node cannot branch — use a Condition node for that).
Condition	A documented branch point with True and False output handles.	Step label — descriptive name (e.g. "Disk usage over 90%?"). Expression — free-text description of the check (e.g. <code>df -h / \ usage > 90%</code>).	One incoming connection. Up to two outgoing connections, one from the True handle and one from the False handle.

Important — Condition nodes are documentation, not logic. Nothing evaluates a Condition's expression at run time. Execution always deterministically follows the node connected to the **True** handle; whatever hangs off **False** never executes. This exists so a playbook's diagram can accurately *document* a real decision point a human engineer would recognize (e.g. "only clear the cache if disk usage is actually high") without Cortex having to safely evaluate arbitrary shell logic itself — a real conditional-execution engine is a larger, separate guarantee this release doesn't make. Two consequences follow directly from this:

- Anything you want to **actually run**, put on the **True** branch.
- A **False** branch is optional, but if you add one, it still has to be a structurally valid, connected, reachable chain — an unconnected dangling node on either branch fails validation at Save (see below) even though the False side will never execute. Model it as "what a human would have done instead" for the audit trail's sake, or leave it empty.

Naming matters — what the AI actually sees

When Claude reviews a ticket, it is shown each **enabled** playbook's **name**, **allowed target OS**, and **required approval level** — nothing else. It never sees a playbook's commands, step labels, or Condition expressions when deciding which one to propose (Section 9, *Running an investigation*, and `app/ai/investigator.py`). That makes the **name** the single most important piece of context the model has for telling two similar playbooks apart, so name playbooks the way you'd label a runbook for a new hire, not a variable:

- Good: `Restart Nginx Service (Linux)`, `Clear Disk Space – Rotate & Purge Old Logs`, `Windows: Restart Print Spooler`
- Poor: `Playbook 1`, `Fix`, `Remediation A`

Validation errors

Save re-validates the graph server-side every time (the same check also re-runs immediately before execution, never trusting a prior save) and rejects it with a specific reason rather than a generic failure:

Problem	What you'll see
No Start node	"Playbook graph has no Start node." (shouldn't normally happen — Start is created automatically and can't be deleted)
More than one Start node	"Playbook graph has more than one Start node."
A Command node no path from Start ever reaches	"Playbook graph has unreachable Command node(s): <code><node id></code> ."
A loop back to an earlier node	"Playbook graph contains a cycle."
A Command or Start node with more than one outgoing connection	"Node <code><node id></code> has more than one outgoing edge but is not a Condition node." (only Condition nodes may branch)

Example playbook workflows

These walk through the exact node-by-node shape of a few common remediation patterns. Build each by adding the described nodes and dragging connections top to bottom; the Start node is always first and is already on the canvas.

1. Restart a hung Linux service

A straight-line playbook — no branching needed, since the point is always to restart the service regardless of its current state.

Start

```
→ Command "Check service status"      systemctl status nginx
→ Command "Restart the service"        systemctl restart nginx
→ Command "Confirm it's active"        systemctl is-active nginx
```

Guardrails: Allowed Target OS `linux`; Required Approval Level `Human-in-the-Loop` (the default — an agent should see the plan before restarting a production service).

2. Free up disk space (Linux)

Another straight-line example, useful when the ticket is a low-disk-space alert and the fix is routine housekeeping rather than a judgment call.

Start

```
→ Command "Show current disk usage"      df -h /
→ Command "Rotate and compress logs"      logrotate -f /etc/logrotate.conf
→ Command "Purge logs older than 14 days" find /var/log -name "*.log.*" -mtime +14 -delete
→ Command "Confirm space was reclaimed"    df -h /
```

Guardrails: Allowed Target OS `linux`; Forbidden Command Strings left empty (none of these steps need an extra tenant-specific block, since the baseline list already blocks anything destructive like `rm -rf`).

3. Windows: restart the Print Spooler

The same restart pattern as example 1, translated to a Windows PowerShell target — demonstrates that the OS guardrail, not the command syntax, is what keeps a playbook from being proposed against the wrong platform.

Start

```
→ Command "Stop the spooler"      Stop-Service -Name Spooler -Force
→ Command "Clear the print queue"  Remove-Item "C:\Windows\System32\spool\PRINTERS\*" -Force
→ Command "Start the spooler"     Start-Service -Name Spooler
```

Guardrails: Allowed Target OS `windows`. Setting this means the same ticket-review pass that would consider example 1 for a Linux asset will never propose this one for it, and vice versa.

4. Conditional cleanup based on disk usage

Demonstrates a Condition node. Remember only the **True** branch actually executes — this shape documents "we only purge temp files when usage is genuinely high," rather than doing it unconditionally on every run.

Start

```
→ Command "Check disk usage"      df -h / --output=pcent
→ Condition "Usage over 90%"      (expression is descriptive only)
    ■■ True → Command "Clear temp files"      rm -f /tmp/*.tmp
    ■■ False → Command "Log OK, no action"     echo "Disk usage nominal, no action taken"
```

Guardrails: Allowed Target OS `linux`. Note the **True** branch uses `rm -f`, not `rm -rf` — the baseline forbidden-pattern check blocks any `rm` command combining recursive (`-r`) and force (`-f`) flags in *any* order or spacing, regardless of what path it targets or what a tenant configures (see *Guardrails* below), so a playbook step that only needs to delete files (not directories) should use a non-recursive flag like `-f` on its own.

Guardrails

The right-hand panel on the builder configures three guardrails that are enforced **server-side**, independent of anything in the graph — the graph's content can never be used to bypass them:

- **Allowed Target OS** — a tag list (e.g. `linux`). If set, the playbook can only run against a ticket whose linked asset's discovered OS matches (case-insensitive substring) one of these tags. Leave it empty to allow any OS, but a ticket with no linked asset at all is always blocked from a playbook that has any OS restriction — there's nothing to check the restriction against.
- **Required Approval Level** — **Human-in-the-Loop** (the default, and the only option for `tenant_admin`) always requires an explicit **Approve & Execute** click before anything runs. **Auto-Approve** skips that click and executes immediately once the AI proposes the playbook — it still always posts both the proposal and the outcome to Actions Taken, so nothing about the audit trail changes, only the human click is removed. Auto-Approve is deliberately hard to turn on: it requires **both** the deployment-wide `ALLOW_AUTO_APPROVE=true` environment variable (Section 10) **and** that you are a `global_admin` — a `tenant_admin` can never set it, even with the environment variable enabled.
- **Forbidden Command Strings** — a tag list of substrings (e.g. `deploy_hook`) that block execution if any command in the playbook's plan contains one. This is checked **in addition to** a hardcoded baseline list that no tenant configuration can weaken or override. A playbook that trips this check (or the tenant list) is rejected the moment the AI tries to select it — it's marked **Blocked** and never reaches the approval step at all.

The baseline itself comes in **two tiers**:

- **Catastrophic — never runnable.** Irreversible data/host destruction: `rm -rf`, `mkfs`, `dd ... of=/dev/`, a redirect to a raw disk device, a classic fork-bomb, and a recursive `chmod 000 /`. These can never be run, acknowledged, or overridden by any configuration. A match is always a hard **Blocked**.
- **Review-eligible — blocked by default, but acknowledgeable.** Disruptive but legitimate and reversible actions — currently **host power control** (`reboot`, `shutdown`, `halt`, `poweroff`, `init 0`). By default these are blocked exactly like the catastrophic tier.

Dangerous Command Review (acknowledging a reboot/shutdown)

Some playbooks legitimately need a review-eligible command — e.g. an OS upgrade that must reboot after a kernel change. Rather than leaving those permanently blocked, a `global_admin` can **acknowledge** the category on a specific playbook, in the builder's **Dangerous Command Review** panel (a `tenant_admin` cannot — this is an intentional loosening of a safety default, so it's gated the same as a free-form command playbook).

Acknowledging a category does **not** make the command run freely. It changes the outcome from an automatic **Blocked** to **Pending Approval**: the run reaches a human, who sees a prominent "disruptive action — review carefully" warning on the AI Remediation panel and must click **Approve & Execute** for it to proceed. An acknowledged run is **never** executed without that explicit human approval, and this is enforced in two independent layers:

1. **Authoring** — a playbook that acknowledges any category **cannot be set to Auto-Approve** at all (the save/import is rejected), so an acknowledged playbook is always Human-in-the-Loop by construction.
2. **Run time** — the executor is only ever reached through a strict allow-list: a run auto-executes *only* if the playbook is Auto-Approve **and** carries no acknowledged (review-required) command **and** is not a free-form runner. Any acknowledged host-power command falls through to Pending Approval regardless of approval level or `ALLOW_AUTO_APPROVE`.

Whatever the human approves is re-checked against the full guardrail engine (including the catastrophic baseline) before it runs. Only the **review-eligible** categories can be acknowledged; the catastrophic patterns are never offered.

The built-in **Full Dist-Upgrade (Ubuntu/Debian)** template ships with `host_power` pre-acknowledged and a final "reboot only if the upgrade requires it" step, so it's a complete worked example: it runs the upgrade, then stops for your explicit approval before rebooting the host.

Linking a credential to a ticket

Before a playbook can actually execute against a host, the ticket needs a linked credential. The **AI Remediation** panel on a ticket's detail page always shows three fields up front — **IP Address** (from the ticket's linked asset), **Username**, and **Password** — so it's clear at a glance what's available. The IP address comes straight from the asset (originally ingested from NetscanXi, or entered manually on the asset's own page); Username shows the linked credential's username; **Password is never shown anywhere** — the field only ever reads "Configured" or "Not set."

Below those fields, a **Linked Credential** dropdown lists every credential vaulted in the tenant by its label and a masked hint (`username@host`). Any authenticated user (not just admins) can vault a new credential via **+ Vault new credential...**

Vaulting once per asset (auto-matching). When a ticket is linked to an asset, the new-credential form collapses to just **Username** and **Secret** — the host is taken directly from the asset's own recorded IP address, not typed in, so the two can never drift apart. That credential is then cross-referenced to the asset itself: the very next ticket opened against the *same* asset finds it automatically and links it with no action needed, exactly as if the agent had picked it from the dropdown. There is at most one such credential per asset — vaulting a second one for the same asset is rejected with a clear conflict error pointing you at the existing entry to update instead (via the dropdown's existing entry, `PATCH`-style, rather than duplicating it). A ticket with no linked asset (or an asset with no recorded IP yet) falls back to the original manual form — label, username, host, port, secret — exactly as before.

Credentials are encrypted at rest and decrypted only in-memory, at the exact moment a command is about to be dispatched — never returned by any API response.

Auto-filling commands from the ticket

A Command node's command field can contain three placeholder tokens instead of a hardcoded host: `{{username}}`, `{{ip_address}}`, and `{{password}}`. The builder's Command node has **+ Username / + IP Address / + Password** buttons that insert the corresponding token at the cursor — so, for example, an SSH login step becomes:

```
{{username}}@{{ip_address}}
```

which resolves per-ticket at investigate/execute time — e.g. `alice@10.0.0.5` — using whatever asset and credential are linked to *that* ticket, rather than a playbook needing a separate copy per host.

- `{{username}}` and `{{ip_address}}` resolve to their real values everywhere a command is shown or stored — the AI's proposed plan, the Actions Taken log, and the execution log all show the actual resolved command, so an approver can read exactly what will run.
- `{{password}}` never resolves to the real secret in anything displayed, logged, or returned by any API response — it always shows as a fixed mask there. The real secret is substituted only in-memory, in the single moment a command is handed to the executor, and is scrubbed from anything the executor echoes back before it's stored — mirroring the same decrypt-only-immediately-before-use rule the credential vault already follows everywhere else.
- If a playbook uses a placeholder the ticket can't currently supply — say `{{ip_address}}` with no linked asset, or `{{password}}` with no linked credential — the run is **Blocked** with a specific reason naming what's missing, the same way an OS or forbidden-command mismatch is blocked (Section 9, *Guardrails*), rather than sending the literal placeholder text as a command.

Running an AI-suggested command (the command-runner playbook)

Every other playbook has fixed commands; the AI only ever *chooses* one. The **command-runner** playbook is the single deliberate exception: its command is the `{{command}}` token, and the actual command is supplied at run time. The catalogue ships one — **Run an AI-Suggested Command (review required)** (the *General* group) — and you can build your own by putting `{{command}}` in a Command node (the builder's **+ AI Command** button inserts it).

How it works, and why it's still safe:

- When the AI investigator selects a command-runner playbook for a ticket, it also **proposes the exact command** to run (based on the ticket's title/description). It never runs it.
- The proposed command appears on the ticket's AI Remediation panel in an **editable field** — the approver can run it as-is, tweak it, or replace it entirely — and it executes **only** after an explicit **Approve & Execute** click. A command-runner run is **always** Human-in-the-Loop, even on a deployment where Auto-Approve is enabled; there is no way to make it run unattended.
- Whatever is approved — the AI's suggestion or the human's edit — is put through the **full guardrail engine before it runs**, so the **non-overridable baseline forbidden-command filter still blocks it** (rm

-rf, mkfs, dd of=/dev/..., shutdown/reboot, fork-bombs, and the rest). A dangerous command can't get through whether the AI proposed it or a human typed it — this is re-checked at approval time, on the final edited command, not just at suggestion time.

- The exact command, and its outcome, are written to the ticket's Actions Taken log like any other run. A `{{password}}` inside a command-runner command is still masked everywhere it's shown or stored and only resolved in-memory for the executor (as above).

Because it runs arbitrary input, the command-runner is locked down at the authoring boundary: **only a global admin can create, edit, enable, or import** a playbook containing `{{command}}`, and it can never be set to Auto-Approve. Importing/enabling it is a deliberate opt-in — it isn't active until an admin brings it in and enables it.

Client Context (per-asset guidance the AI reads)

Each asset has a **Client Context** section on its detail page (**Assets** → **open an asset**), marked with a **"Read by AI"** badge. It's a free-text note describing the client and the asset's operational role, plus any handling constraints — for example:

Production database for Acme Corp (24/7 SLA). Never reboot or stop services during business hours (08:00–18:00). Snapshots run nightly at 02:00.

Two things use it:

- **Operators** — it travels with the asset as plain reference, so whoever picks up a ticket sees the client's ground rules without hunting through a CMDB.
- **The AI investigator** — when a ticket linked to that asset is investigated, its Client Context is included in what Claude reads **before** it proposes a fix, with explicit instruction to treat it as authoritative. The AI is told to let it govern the choice: to prefer a safer or read-only playbook, or to select **NONE** (deferring to a human), when the context indicates that the available fixes could be disruptive or unsafe for that particular box. Its rationale will reflect how the context shaped the decision.

It is **advisory input, not a hard control** — the client context steers the AI's *selection*, while the non-overridable guardrail engine (OS match, forbidden-command baseline, and the mandatory approval click for Human-in-the-Loop playbooks) remains what actually gates execution. Keep the notes short and specific (maintenance windows, do-not-reboot rules, SLA sensitivity, "standalone box, safe to take offline"); an asset with no Client Context simply contributes nothing extra to the prompt. It's edited and saved independently of the asset's other sections, and left blank by default on existing and newly-discovered assets.

Running an investigation

On a ticket's detail page (with the AI module enabled and a credential linked), click **Investigate with AI**. The panel shows a spinner while Claude reviews the ticket and asset context (including the asset's **Client Context**, if set — see above) against every currently-*enabled* playbook (a disabled playbook is never even shown to the model — it structurally cannot be selected) and picks the best match, or determines that none apply.

What happens next depends on the outcome:

- **No suitable playbook** — the run is marked **Failed** with the AI's stated reasoning; nothing was proposed and nothing executed.
- **Blocked by guardrails** — the selected playbook failed its OS or forbidden-command check; the run is marked **Blocked** with the specific reason, and it never reaches an approval step.
- **Pending Approval** (Human-in-the-Loop playbooks, the default) — the panel shows the AI's rationale, the selected playbook's name, its guardrail summary, and the exact ordered list of commands it's proposing to run. From here you have three actions: **Simulate**, **Execute Live on Client** (unlocked only after a simulation), or **Reject** (see *Simulate first, then execute live on the client* below). Every action is attributed to you and logged.
- **Succeeded / Failed** — once a run executes (whether you approved it or it was Auto-Approve), the panel shows an **Execution Results** section: each step lists its command, exit code, duration, **and the actual command output** in a scrollable block, so the operator can read exactly what each step returned (e.g. the text of a `systemctl status` or `df -h`). A concise outcome summary sits below it.
- **Pending Approval, editable command** (command-runner playbook) — instead of a fixed plan, the panel shows the AI's proposed command in an editable field for you to review, edit, and approve (see *Running an AI-suggested command* above).

Viewing and storing the results. The AI panel's Execution Results show the *latest* run's output. Cortex also **stores the full results permanently** in two places: the run itself keeps its complete execution log (every command, output, exit code, and duration), and — so the results stay viewable in the ticket's own history regardless of how many later investigations happen — the per-step command output is folded into a **Results** entry in the ticket's **Actions Taken** log. Every run leaves its own permanent Results entry there, so an operator can scroll a ticket's history and read exactly what each past run returned. (Any credential secret in a command's output is masked in both places, exactly as it is in the command text.)

A ticket can be investigated again at any time, starting a fresh run without disturbing the stored results of previous ones.

Simulate first, then execute live on the client

Live execution is available on every deployment and for every playbook — but it is deliberately a two-step action, so nothing runs on a client until an operator has seen a simulation of exactly what will run:

- **Simulate** — runs every step through the **simulated** executor, no matter how the deployment is configured. Nothing touches the client; you get the exact command list and simulated output in the **Execution Results** block, and the run **stays Pending Approval**. Simulate as many times as you like (e.g. after editing a command-runner's command).
- **Execute Live on Client** — the operator's explicit go-ahead. It is **disabled until you have simulated this run at least once** (the server enforces the same rule — a live request without a prior simulation is refused). Clicking it opens a final **"Are you sure?"** confirmation popup that spells out that the commands will run on the client for real and cannot be undone from Cortex; only on confirming does it **actually run the**

commands on the client over SSH, using the credential linked to the ticket (username/host/port + the vaulted secret), and record the real stdout/stderr and exit code.

- **Reject** — stop here; nothing runs.

Every action is recorded in the Audit Trail (Section 7), attributed to the operator: `remediation_investigate`, `remediation_dry_run` (each simulation), `remediation_execute_live` (the confirmed live run, with the run id, playbook, step count and whether it was real or demo), and `remediation_reject`. The per-step results are also written to the ticket's **Actions Taken** log. Together these give a complete, permanent record of who ran what, when, and with what outcome.

Execute Live on Client always opens a real SSH connection — there is no "demo" execution; the simulated path is the separate **Simulate** button. The full guardrail engine is re-checked at execution time (a command-runner's edited command included), so nothing that would be blocked can run. Credentials are decrypted in-memory only, immediately before dispatch, and any secret is masked out of everything stored or displayed. If a deployment has no reachable host (or `paramiko` isn't installed), the live run is recorded as a clean **Failed** with an SSH-error message — it never crashes the request.

Root vs non-root is auto-detected (sudo when needed). A fleet often mixes hosts where the SSH account is root with hosts where it's a non-root service account. Cortex handles both automatically: on first contact with a host it checks `id -u`, then runs commands **directly** on a root account or **through sudo** on a non-root account — the whole command runs in a shell under `sudo -S`, with the sudo password fed on **stdin** (never in the command text or the logs). The sudo password defaults to the ticket's vaulted credential secret (the account's own login password), so the common case needs no extra config. **Write playbook commands *without* a leading sudo** — Cortex adds it only where required. On a non-root target, the account's sudoers entry must permit the command (its own password, or `NOPASSWD`), and `requiretty` must be off (the modern default). Controls: `SSH_USE_SUDO` (`true` default = auto-detect; set `false` to never use sudo), `SSH_SUDO_PASSWORD` (override, rarely needed), `SSH_SUDO_SHELL` (default `bash`).

Host keys. By default Cortex **trusts a host's key on first contact** (`SSH_AUTO_ADD_HOST_KEYS=true`) so a newly-added host connects without manual setup. For stricter security, set it to `false` to reject any host not already in the container's `known_hosts` (pre-seed with `ssh-keyscan`).

Auto-Approve playbooks get a head start under this model: instead of running live unattended, an auto-approve run **auto-runs the simulation** during investigation, so the operator opens the ticket to a completed rehearsal and can go straight to **Execute Live**. Live execution therefore *always* takes an explicit human click plus a prior simulation — there is no configuration that runs a playbook on a client with no human and no rehearsal.

Order of operations to actively fix a client: link a credential to the ticket → Investigate → review the proposed plan → **Simulate** to rehearse and read the simulated output → **Execute Live on Client** to apply the fix for real. The reboot/host-power review (Section 9) and the forbidden-command baseline still apply on top of this.

10. Deployment Notes

- Schema changes ship as Alembic migrations, applied automatically on container startup (`alembic upgrade head`). There is no manual migration step for a standard Docker Compose deployment.
- The app requires Postgres in production; the `DATABASE_URL` in `docker-compose.yml` points at the bundled `db` service by default.
- The frontend is a Vite-built React SPA, compiled in a separate Docker build stage and served by the same FastAPI container as static files — there is no separate frontend server or port to run in production.
- The credential vault is **not** an environment variable in this release — it's created once from **Admin** → **Vault** by a `global_admin` (Section 9), and the resulting key is wrapped at rest using `SECRET_KEY`. There is no `VAULT_KEY` variable to set, and no supported way to migrate a vault to a new `SECRET_KEY` after the fact — treat `SECRET_KEY` as permanent for the lifetime of the deployment once a vault exists.
- AI Remediation Module environment variables (see `.env.example` for the full reference):
 - `ANTHROPIC_API_KEY` — a **fallback** Claude API key, only used when no key has been saved from Admin → AI Settings (Section 9). Leave it unset in environments where the AI module will stay disabled, or where the key is managed entirely through the UI; the app starts fine either way, and the investigate endpoint fails with a clear, specific error rather than crashing if neither source has a key configured.

Live execution is always on — **Execute Live on Client** is available on every deployment and always opens a real SSH connection to the client using the ticket's linked credential. There is no on/off or demo switch; it is gated instead by an explicit human confirmation **and** a prior simulation of the same run (Section 9), and the **Simulate** button provides the safe rehearsal. `paramiko` is a required dependency (installed in the image); if it's ever missing or the host is unreachable, a live run is recorded as a clean **Failed**, never a crash.

- `SSH_AUTO_ADD_HOST_KEYS` — `true` (default) trusts a host's key on first contact so a new host just connects; set `false` to verify against `known_hosts` and **reject unknown hosts** (pre-seed with `ssh-keyscan`). `SSH_CONNECT_TIMEOUT` / `SSH_COMMAND_TIMEOUT` (seconds) bound the connection and each command. Authentication uses the vaulted secret as a password; the local SSH agent and on-disk keys are never consulted.
- **Privilege elevation is auto-detected per host** — `SSH_USE_SUDO=true` (default): Cortex probes `id -u` on first contact and runs directly on a root account or via `sudo` on a non-root one. Set `false` to never use `sudo`. `SSH_SUDO_PASSWORD` overrides the `sudo` password (defaults to the credential secret); `SSH_SUDO_SHELL` sets the shell (default `bash`).
- `ALLOW_AUTO_APPROVE` — `false` by default. Set to `true` only if you want to permit `global_admin`-created playbooks to skip the human approval **for the simulation step**; even then, an auto-approve run only auto-**simulates** and still requires an explicit human **Execute Live** click to touch a client. `tenant_admin` can never enable this regardless of the environment setting.