AI OPS · AI ENGINEERING · AUTOMATION ENGINEERING

# An AI-supervised deployment pipeline that ships a secured full-stack server for the price of a coffee.

How an n8n workflow, an Ansible control plane, and a bounded self-healing AI agent turn a multi-day, senior-level server build into a single form submission — and what it took to make the machine actually block an attack.

PROJECT

Project NSD — New Server Deployment

DOMAIN

Applied AI for Infrastructure

STACK

n8n · Ansible · ModSecurity · Qwen

**Tesfa Michael** — AI Engineer building production systems across automation, ML, and applied AI. Built with a business operator's mind.

# Contents



A NOTE ON SCOPE

This is the portfolio-depth account of a working system, not a tutorial. It assumes you know what a reverse proxy, a WAF, and an SSH key are. Every number in the outcomes and value sections is either measured on the live system or cited to a named source in the references.

SECTION 01

# Executive Summary

Standing up a hardened, full-stack web server — firewall, reverse proxy, web application firewall, database, backend, frontend, TLS, and a real verification pass — is a day or more of careful, senior work. Project NSD compresses that work into a single form submission, supervises every step with a bounded AI agent, and returns a live, attack-tested link. The compute cost of a run is about six US dollars.

Here's what's actually happening under the hood. An **n8n** workflow accepts a deployment request, normalizes it, and hands each stage of the build to an **Ansible** control plane over SSH. Ansible provisions the target server role by role: system hardening, Nginx, ModSecurity with the OWASP Core Rule Set, PostgreSQL, a Laravel backend, a React frontend, routing, and Let's Encrypt TLS. Between every role, the workflow reads the exit code and decides what happens next. When a role fails, a Qwen-powered agent is handed the sanitized error and a single, tightly bounded tool that can only edit that client's configuration files. It proposes one fix; the workflow re-runs the role and counts the attempt; after five it stops and escalates to a human over Telegram rather than build anything on a broken base.

The part most people skip is the ending. A deployment pipeline that reports success is not the same as a server that is actually safe. The final gate runs a thirteen-attack suite against the live site, and the workflow only calls a deployment done when the machine genuinely blocks those attacks and an independent HTTP request — made by n8n itself, not by the target — confirms the site responds. Getting that gate to pass honestly, rather than pass falsely, was the hardest part of the build. It is covered in full in *What Failed*.

| ~6 USD | ~300× | 12/13 | 5 |
|---|---|---|---|
| approximate compute cost of one full deployment | local cost reduction vs. a contractor doing the same work | WAF attacks blocked on the live site after tuning — a passing gate | bounded self-heal attempts per role before human escalation |

Everything that follows is the reasoning, the architecture, the parts that broke, and a researched account of the value the system produces for anyone who runs it.

SECTION 02

# The Operational Friction

In Ethiopia, a professional contractor handling this kind of end-to-end server deployment and configuration can charge up to **300,000 Ethiopian Birr** for the engagement. That is the number this project starts from, because it names a cost that a small business or a solo founder absorbs quietly, one deployment at a time.

Convert it and the number keeps its weight. In May 2026 the birr traded at roughly **161 to the US dollar**, after depreciating about 18.7% over the preceding twelve months in the wake of the country's 2024 currency float.[1] At that rate, 300,000 birr is on the order of **1,860 US dollars** — which, notably, is not an outlier. It sits squarely inside the global market rate for the same work: freelance infrastructure-automation projects are quoted at **2,000 to 5,000 dollars** on Upwork, and DevOps engineers bill a median of **60 dollars an hour**, rising past 150 for senior work.[2] The Ethiopian figure isn't a local quirk. It's the worldwide price of an afternoon of scarce, senior expertise.

Now look at what that expertise is actually spending its hours on. A correct first deployment is not one task; it is a chain of interdependent ones, and each has a way of going wrong that only shows up at the end.

| STAGE OF A MANUAL BUILD | WHERE THE TIME GOES | THE FAILURE MODE NOBODY WARNS YOU ABOUT |
|---|---|---|
| System hardening | Firewall rules, fail2ban, unattended upgrades | Locking yourself out with a bad UFW rule |
| Nginx + ModSecurity WAF | Compiling ModSecurity v3 from source (15–25 min), wiring the OWASP CRS | The WAF loads 800+ rules and still blocks nothing |
| PostgreSQL | Per-app database, user, and grants | Postgres 15 schema-grant changes that silently break migrations |
| Backend + frontend | Composer, PHP 8.3, Node builds, heap sizing | A React build that runs the box out of memory and dies |
| Routing + TLS | Site templates, Certbot, renewal | Certbot's edits getting overwritten on the next deploy |
| Verification | Is it up? Is it safe? Is it actually serving? | Calling it done because there were no visible errors |

*The manual workflow this system replaces. Every row is a place a senior engineer earns their rate — and a place a junior loses a day.*

The friction isn't only cost and time. It's risk that lands after the invoice is paid. Human error is estimated to account for roughly **80% of IT outages**,[3] and a misconfigured or absent web application firewall is a direct path to the kind of incident whose average cost reached **4.44 million dollars** globally in 2025 — and **10.22 million** in the United States.[4] The gap between "the deploy finished" and "the server is defensible" is exactly the gap where those numbers live.

This is the market the system is built for, and it is not a small one. The broader category — using AI to run and repair IT operations, what the industry calls AIOps — was valued at **12.4 billion dollars in 2024** and is projected to reach **123.1 billion by 2034**, a compound annual growth rate of 25.8%.[5] The thesis underneath Project NSD is that the capabilities driving that growth do not have to stay locked inside enterprises with enterprise budgets.

The real cost of a manual deployment isn't the day it takes. It's the week you don't notice it went wrong.

SECTION 03

# What the System Is

Project NSD is a form. An operator fills in a client name, a target server, a domain, a repository or two, and a few options, and submits. What happens next is a supervised, end-to-end build of a production web server that ends with a verified, live link — or a precise, human-readable escalation explaining exactly where and why it stopped.

The workflow is deliberately parametric. Nothing about a specific client is hard-coded into the shared automation. The form submission is normalized into a single set of per-client variables — the client name, derived database credentials, a generated password, the deployment profile, file paths, domain handling, and base64-encoded environment blobs — and those variables drive everything downstream. Two different clients are two different sets of files in a `clients/<name>/` directory and nothing else. The shared roles are never edited to accommodate a deployment. That single design rule is what makes the system reusable rather than a one-off.

## The path a request takes

In practice, a run moves through four movements. First, **intake and preparation**: the form is cleaned, the per-client files are written on the control plane, SSH trust is established to the target, and a pre-deployment snapshot is taken so there is something to roll back to. Second, **the build**: nine Ansible roles run in dependency order, each one wrapped in a self-healing supervisor. Third, **the security gate**: a thirteen-attack suite runs against the live domain, and if the machine isn't blocking properly, a bounded agent tunes the firewall and the suite runs again. Fourth, **verification and handoff**: health checks confirm the services are active, n8n independently requests the live URL to prove it responds, and a Telegram message reports the result with the link.

The order matters and it is not arbitrary. You cannot route traffic to an application that doesn't exist yet, you cannot obtain a certificate before the domain resolves to a running server, and you cannot honestly verify security before the thing you are securing is up. The pipeline follows the real dependency chain of the work, which is why a failure early in the chain halts the rest instead of stacking more work onto a broken foundation.

THE ONE BEHAVIOR THAT DEFINES THE SYSTEM

Every role-running command ends by printing its exit code, and — unlike the original pipeline — the workflow now *reads* it. Success moves to the next role. Failure enters a bounded loop: the AI proposes one fix to the client's config, the workflow re-runs the role and increments a counter it owns, and after five attempts it escalates to a human. The AI never decides when to stop. The workflow does.

The result is a system that behaves less like a script and more like a careful engineer who checks their work between steps, knows when to try again, and knows when to stop and ask for help.

SECTION 04

# The Stack, and Why Each Tool Is There

Every tool in this system is doing a job that would be harder, slower, or more fragile without it. None of them is the identity of the project. They are the instruments; the system is the composition.

## n8n — the orchestration layer

n8n is a fair-code workflow automation platform, founded in Berlin in 2019 and built on Node.js and TypeScript, with native support for building AI agents alongside more than 400 integrations.[6] It reached a 2.5-billion-dollar valuation in its October 2025 Series C, backed by Accel, Sequoia, and Nvidia's venture arm[6] — worth noting only because it signals that visual, self-hostable orchestration is now a serious category, not a hobbyist's toy. In this system, n8n is the conductor. It owns the form, the control flow, the retry counters, the human-approval and escalation gates, and the connections to every other service. Critically, it is self-hosted: the whole pipeline runs inside a Docker container on a server the operator controls, which is what makes a six-dollar deployment possible in the first place.

## Ansible — the execution layer

Ansible is Red Hat's open-source automation engine: agentless, driven over SSH, and idempotent by design, meaning a task only changes what needs changing and is safe to run again.[7] Its role in this stack is the actual work of building the server. n8n decides *what* should happen and *when*; Ansible carries out the *how* — installing packages, compiling ModSecurity, templating Nginx sites, creating databases, deploying code. The division is deliberate. Ansible's idempotency is what lets the self-heal loop re-run a failed role safely; a role that half-completed and then failed can be run again without corrupting the state it already reached. That property is not a nice-to-have here. It is the foundation the entire retry mechanism stands on.

> THE BRIDGE MOST PEOPLE GET WRONG
>
> n8n runs inside Docker; Ansible runs on the host. They are different environments. The only connection between them is SSH — the n8n container reaches the host over the Docker bridge gateway, and only from there does Ansible reach the target. Nothing in the pipeline pretends the two are the same machine. That boundary is a security feature, not an inconvenience.

## ModSecurity + the OWASP Core Rule Set — the defense layer

ModSecurity v3 is the web application firewall engine; the OWASP Core Rule Set (CRS) v4 is the ruleset it enforces. The CRS uses a paranoia-level model with an anomaly-scoring engine: the paranoia level decides *which* rules run, while a separate anomaly-score threshold — 5 by default — decides *how many* triggered rules it takes to block a request.[8] Understanding that these two dials are independent turned out to be the difference between a WAF that looks installed and one that actually defends. More on that in *What Failed*.

### The application tier

- **PostgreSQL** — one isolated database and user per client, with the schema grants that modern Postgres requires.
- **PHP 8.3 + Laravel** — the backend, deployed with Composer and migrated on the box.

### The intelligence & control tier

- **Qwen 3.7 (via Alibaba Cloud)** — the model behind the self-heal agent; priced at roughly `$0.78` per million input tokens and `$3.90` per million output.[9]
- **A bounded "Client File Ops" tool** — the agent's only hands, restricted by an allowlist to the client's own config files.

- **Node 20 + React (Vite)** — the frontend, built on the server with a swap file and a raised heap ceiling so large builds don't die.
- **Nginx** — reverse proxy and static host, with single-domain and split-domain routing templates.
- **Certbot / Let's Encrypt** — TLS provisioning and renewal.
- **Telegram** — the human-in-the-loop channel for success and escalation.
- **SSH key pairs** — two distinct trust hops: container→host, host→target.

The choice of Qwen for the agent is worth a sentence of honesty: it was selected because it is capable, inexpensive, and already integrated, not because it is the only model that could do the job. The architecture treats the model as a replaceable component. The guardrails around it — the allowlist, the deterministic counter, the human escalation — are the parts that are not replaceable, and they would matter regardless of which model sat inside them.

SECTION 05

# The System Map

The diagram below traces one deployment from form submission to verified link. Read it left to right in three bands: the operator and orchestration plane, the command center where Ansible runs, and the target server being built. The single most important element is the loop in the middle band — the part that reads an exit code and decides whether to continue, repair, or escalate.

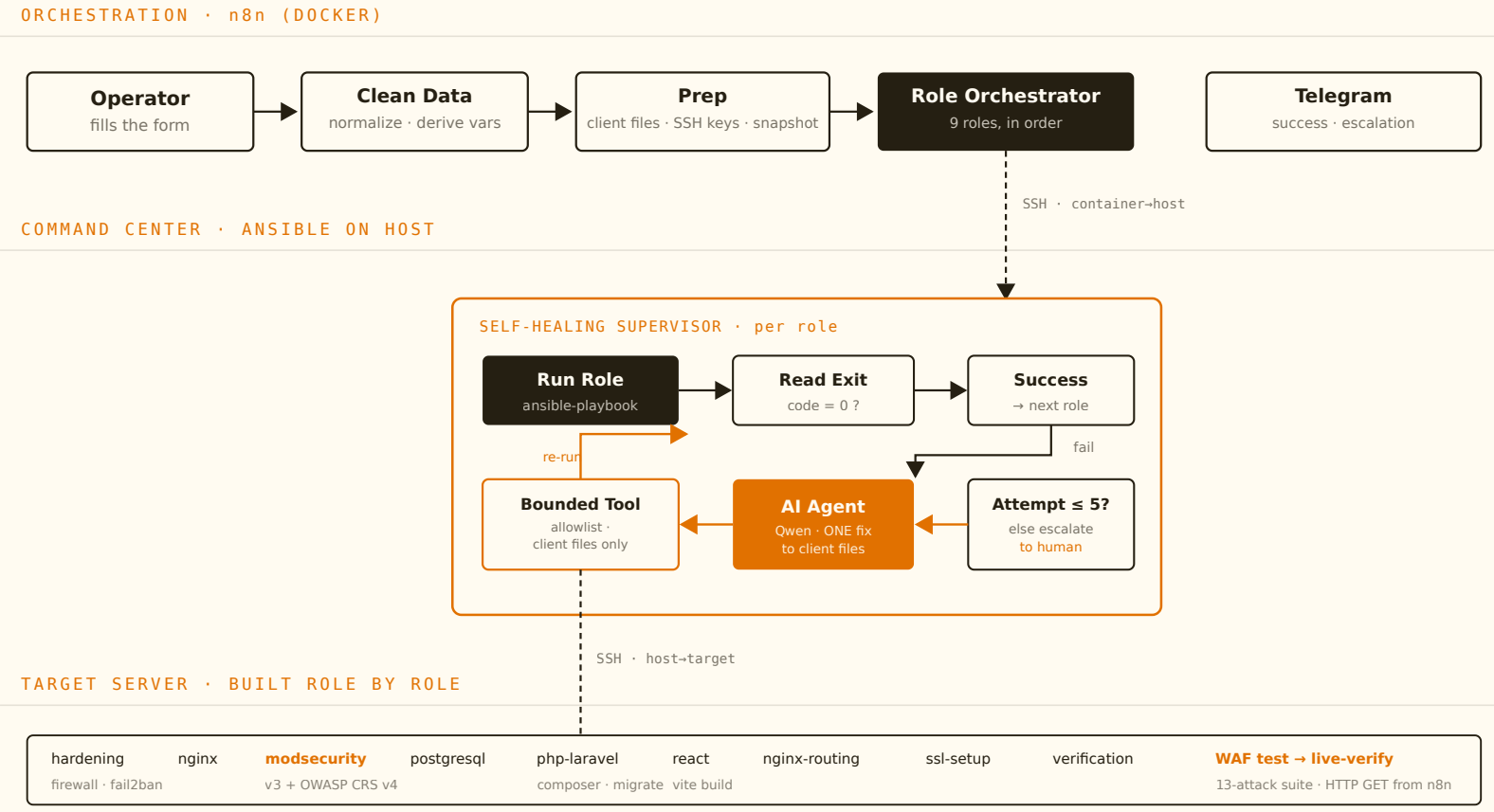


Figure 1 — Project NSD data and decision flow. Amber marks the elements that make it self-healing: the bounded tool, the AI's single fix, and the security gate. The two dashed SSH hops are the only paths between environments.

What the diagram cannot show is the discipline in the loop. The AI agent produces exactly one change per pass and then stops; it does not re-run the role, and it does not decide how many attempts remain. The workflow owns the counter. This is the structural reason the system is safe to let run unattended: the intelligence is bounded on every axis that matters — what it can touch, how many times it can try, and who it answers to when it can't.

SECTION 06

# Techniques Implemented

A working system is a stack of decisions that each closed a specific failure mode. These are the ones that carry the most weight.

## Reading the exit code, not the absence of errors

The original pipeline ended every deployment command with `echo "EXIT_CODE:$?"` and then never read it. Because the `echo` always succeeded, the SSH node reported success even when the playbook underneath it had failed, and a broken step flowed quietly into the next one. The core technique of the rebuild is embarrassingly simple to describe and consequential in effect: parse that exit code, branch on it, and treat "no visible error" as the beginning of verification rather than the end of it.

## Bounded agency: AI proposes, deterministic code disposes

The self-heal agent can read the sanitized error and the failing client files, and it can make exactly one change — but only to files inside the client's own directory, enforced by an allowlist that refuses the shared Ansible, the SSH keys, the vault, and the inventory's target line. It cannot run arbitrary shell. It cannot touch another client. When the fix needs to reach the live server, a *deterministic* node does the deploying and the Nginx restart, not the model. The pattern is worth naming because it generalizes: let the AI analyze and propose inside a narrow, typed interface; let ordinary code execute, count, and gate.

## Anomaly-score tuning instead of blunt blocking

Making the WAF actually block attacks came down to the CRS's own model: the ruleset ships at paranoia level 1, which catches obvious threats but lets cleverer evasions through. Raising the blocking and detection paranoia to level 2 — while keeping the anomaly threshold at the recommended 5 — closed the evasions without flooding the site with false positives on legitimate traffic.[8] The tuning lives in a per-client override file that the system seeds by default, so every future deployment inherits a WAF that blocks rather than merely observes.

## Independent verification

The final proof is an HTTP request made by n8n itself against the public URL — not a `curl` run on the target checking its own health. A server asserting that it is up is not evidence. An external client reaching it is. That distinction is the whole reason the success message can be trusted.

### Also in the build

- **Per-client isolation** — each deployment gets its own database, user, and grants; app files owned and scoped per client.
- **Idempotent recovery** — roles designed to survive a half-completed run and re-converge on the next pass.
- **Deterministic retry with a hard cap** — five attempts, owned by the workflow, never by the model.

### Guardrails

- **Secret hygiene** — errors are sanitized before they reach the model, Telegram, or logs; secret-bearing tasks are marked no-log.
- **Fail-closed ordering** — an escalation halts later roles rather than building on a broken base.
- **Rollback position** — a pre-deploy snapshot exists before the first change lands.

SECTION 07

# Decisions & Trade-offs

Good engineering is mostly the record of what you chose *not* to do, and why. These are the choices that shaped the system, stated with their costs.

### n8n over a bespoke orchestrator

A custom Python or Go service could have run this pipeline. n8n won because the workflow is a control-flow problem with human gates, retries, and a dozen integrations, and n8n makes that visible and modifiable without a redeploy. The trade-off is real: a visual graph is harder to unit-test than a codebase, and expression bugs hide in places a type checker would catch. The *What Failed* section opens with exactly such a bug. The bet was that legibility and speed of change outweighed the loss of static guarantees for a system that a small team needs to keep adapting. For this use case, that bet held.

### Ansible over shell scripts

The temptation in a deployment pipeline is to reach for SSH and a pile of shell. Ansible's idempotency is the reason not to. A shell script that fails halfway leaves the box in an unknown state; an Ansible role that fails halfway can be re-run to converge. Since the entire self-heal mechanism depends on safely re-running a failed role, shell was never actually an option. The cost is a steeper concept — roles, plays, inventory, variable precedence — but it buys the one property the system cannot live without.

### Let the AI write config, but never let it execute

The sharpest line in the design is the one between analysis and action. The agent writes a proposed configuration change into a file; a deterministic workflow node deploys it and restarts the service. This is slower to build than handing the model an open SSH tool, and it is far safer. An agent with an unrestricted shell is a liability whose blast radius is the whole server. An agent that can only write to one allowlisted directory, whose output is then applied by code that validates and can roll back, is a bounded, auditable component. Given a choice between a capable agent and a governable one, the system chooses governable every time.

### Seed the WAF hardening per client, not in the shared role

The durable fix for the firewall could have gone into the shared ModSecurity role. It went into the per-client configuration instead, seeded automatically at file-creation time. The reason is the project's central rule: the shared automation is never edited for a specific deployment. Keeping the hardening in the client layer preserves that rule, keeps the shared roles clean for an eventual open-source release, and still gives every deployment a WAF that blocks by default. The trade-off is that the same two lines live in every client folder rather than in one place. For a reusable system, that redundancy is the correct price.

SECTION 08

# The Threat Model

This system takes input from a form, hands work to an AI, and holds real credentials for a live server. Each of those is a way in. A deployment tool that ignores that isn't a tool — it's an incident waiting for a start date. Here is the model I built the system against, before the first live run.

The design rule underneath everything in this section is simple to state and unusual to actually hold: **nothing that arrives from outside is trusted, and nothing sensitive is ever allowed to leave.** Form values are attacker-controlled until proven otherwise. The AI's output is a suggestion, never a command. Secrets exist to be used by the machine and never to be seen — not in a log, not in a prompt, not in an error message, not in the execution history n8n keeps by default. Most of the engineering below is the cost of taking those two sentences literally.

## The surfaces that matter

I mapped the system into five places an attack or a leak can originate, and treated each one as a boundary to defend rather than a convenience to assume. The form is a semi-public entry point. The AI agent is a component that can be steered by the very text it's asked to analyze. The credential store holds the keys to a live host. The SSH path crosses three environments. And n8n's own execution log is a quiet exfiltration risk that most people never think to check.

| SURFACE / THREAT | WHY IT'S DANGEROUS | CONTROL IN THE SYSTEM | RESIDUAL RISK |
|---|---|---|---|
| **Form input → command injection** | A hostname, path, or repo URL concatenated into a shell string is a remote-code-execution hole with a friendly UI. | Every value is validated against an expected shape and written into structured per-client files; nothing from the form is interpolated into a shell command. Playbooks and profiles are allowlisted, not free-text. | Low — bounded by the allowlist and the file-based hand-off. |
| **Prompt injection → the AI agent** | The agent reads failure logs. A crafted error or a poisoned file could try to talk the model into touching something it shouldn't. | The agent has exactly one tool, and that tool can only write inside the current client's config directory. It cannot run shell, cannot reach other clients, cannot change its own retry budget. Injection can waste an attempt; it cannot escalate. | Low–moderate — contained by the tool allowlist and the hard cap. |
| **Secret leakage → logs & execution data** | The most common real-world leak isn't theft; it's a password printed into a debug line, a diff, or n8n's saved execution history. | `no_log` on the sensitive Ansible tasks only; diff disabled for secret-bearing templates; sanitized error text is what reaches the AI and the operator. Credentials live in n8n's store and Ansible Vault, referenced by name, never inlined. | Low — but the highest-vigilance item, re-checked on every change. |
| **SSH trust → three environments** | A private key readable by the wrong process, or a disabled host-key check, turns one compromise into three. | Two scoped keypairs, not one shared key: n8n-container→host and host→target. Host-key verification stays on. Least privilege per hop; no security downgrade for convenience. | Low — key scope and host-key checks intact. |
| **Result spoofing → false "success"** | An AI that says "it worked," or a bare exit code, is not proof a server is live and safe. | Verification checks real state: the security suite attacks the live domain, and n8n makes an independent HTTP request to confirm the response. Success is a measured fact, not a claim. | Very low — evidence-based, not assertion-based. |

*The abbreviated threat model the system was reviewed against before its first production deployment. Likelihood and impact were assessed per row; only the controls and residual risk are shown here.*

## Why the boundaries are drawn where they are

The most important architectural decision in the whole system is a security decision disguised as a plumbing detail: n8n and Ansible run in different environments, and the only thing connecting them is a narrow SSH hop. It would have been easier to install Ansible inside the n8n container and let the workflow shell out directly. It would also have collapsed two trust domains into one and handed the orchestration layer — the part exposed to form input and AI output — direct execution rights on the host. Keeping them apart means a compromise of the workflow does not automatically become a compromise of the machine. The boundary is inconvenient on purpose.

The same instinct governs the AI. The agent is the least-trusted component in the system, not the most — it's the one piece that can be influenced by adversarial text, so it gets the smallest possible blast radius: one tool, one directory, no shell, a counter it can't touch, and a human at the end of the rope. The stat that frames why this care is proportionate rather than paranoid: roughly 80% of IT outages trace back to human error,[3] and the average data breach now costs $4.44M globally and more than $10M in the United States, on a lifecycle that stretches past 240 days before it's contained.[4] A deployment tool that provisions public-facing servers is not a place to be casual about the parts that can go wrong quietly.

THE RULE THAT MADE THE RELEASE SAFE TO PUBLISH

Because secrets are referenced by name and never inlined, the entire workflow and Ansible project can be sanitized for open source by swapping credential references for placeholders — no values to scrub, because none were ever written down in the first place. Security-by-architecture is what made the open-source cut a mechanical step instead of an audit.

SECTION 09

# What Failed

Every honest case study has this section. Here is what broke, what surprised me, and what the actual constraint turned out to be — because the fixes are where the real engineering lives.

### FAILURE 01 — THE WAF THAT INSPECTED EVERYTHING AND BLOCKED NOTHING

The security gate's whole purpose is to confirm the firewall blocks attacks. For a long stretch, it didn't. ModSecurity was installed, more than 800 OWASP rules were loaded, and a thirteen-attack suite sailed straight through — every attack returned `200 OK`. The instinct was to assume the engine was off. It wasn't. `SecRuleEngine` was `On`, the rules were loaded, and the site still failed. The part most people skip is the layer below the on/off switch: the CRS runs an *anomaly-scoring* model, and its behavior is governed by a paranoia level that decides which rules even execute. The ruleset was sitting at level 1, catching the blunt attacks and missing the evasions. The real challenge wasn't turning the WAF on. It was understanding that "on" and "blocking effectively" are two different dials.[8]

### FAILURE 02 — A PASSING TEST THAT WAS LYING

Two of the pipeline's branch conditions read a value from an upstream node using n8n's paired-item reference. After the pipeline was rewired to route each role through a sub-workflow, that reference quietly stopped resolving, because a sub-workflow's output does not carry the item lineage the reference depends on. The conditions didn't error — they were rescued by a second, redundant condition that happened to be true — so the branches "passed" while silently ignoring the field they were supposed to evaluate. This is the exact hazard of visual orchestration: a type checker would have caught it; the canvas hid it behind a green checkmark. The fix was to switch the reference to one that doesn't depend on item lineage and remove the condition that was masking the bug.

### FAILURE 03 — CORRUPTION I CAUGHT BECAUSE I WAS CHECKING

While preparing the sanitized open-source release, a base64-encoded configuration blob was transcribed with a single stray non-ASCII character mid-string. Decoded, it would have produced a broken firewall config that failed `nginx -t` and stopped a fresh deployment cold. It was caught because the process verifies checksums rather than trusting a copy, and because a later scan re-read the file it had written. The lesson isn't "be careful." It's that a pipeline handling config at scale has to assume transcription and transfer will occasionally corrupt data, and build the verification in so the corruption surfaces immediately instead of three deployments later.

### FAILURE 04 — THE SSH KEY THAT WAS TOO PRIVATE TO WORK

The bridge from the n8n container to the host refused to authenticate, and every instinct said the private key's permissions were the problem — so the first move was to tighten them to `600`, owner-only, the setting SSH nags you to use. It kept failing. The counterintuitive fix was to loosen them to `644`. The key isn't owned by the `node` user that n8n's process runs as; at `600` that process simply can't read it, and SSH reports the failure as an auth rejection rather than a permission error, which sends you debugging the wrong layer. The lesson is a specific one about containerized automation: "more locked down" and "working" are not the same axis, and the right permission is the one that matches the process identity that actually has to read the file — verified, not assumed from habit.

## What surprised me

The slowest single step in the build is compiling ModSecurity v3 from source — fifteen to twenty-five minutes on a fresh target. It dominates the wall-clock time of a first deployment, and it is the one part a prebuilt module would meaningfully accelerate. I left it as-is for now because it is idempotent and correct, and correctness came first. It is the obvious next optimization, and naming it here is more useful than pretending the run is instant.

SECTION 10

# The Skill Set Demonstrated

This system sits at the intersection of three disciplines that are usually held by three different people. Building it end to end is the point.

| DISCIPLINE | WHAT IT MEANT ON THIS PROJECT |
|---|---|
| **AI Ops** | Applying AI to run and repair operational work: an agent that diagnoses a failed deployment step from a sanitized log and proposes a corrective change, inside a bounded retry-and-escalate loop. This is the AIOps pattern — automated remediation with a human backstop — applied to first-time provisioning rather than only incident response. |
| **AI Engineering** | Designing the agent as a governed component: a defined input contract, a single allowlisted tool, an output schema, a hard iteration cap, run-scoped memory, and secret-sanitized context. The engineering is not "add an LLM." It is building the cage the LLM operates safely inside. |
| **Automation Engineering** | The deterministic backbone: an idempotent Ansible project, a parametric per-client model, an SSH trust architecture across three environments, exit-code-driven control flow, and a security gate with independent verification. The automation is what makes the AI's contribution small, bounded, and optional rather than central and dangerous. |
| **Security Engineering** | Threat-model-aware choices throughout: input treated as untrusted, secrets kept out of logs and prompts, host-key handling, least-privilege tooling, and a WAF tuned to actually block rather than merely to be present. |

The through-line is a business operator's instinct applied to infrastructure: not "what is the most sophisticated system I can build," but "what is the smallest, safest system that removes the most expensive human bottleneck." The AI is there because it closes the last gap — the judgment call on a failed step — not because a system needs an AI to be interesting.

> **The engineering here isn't adding an LLM to a pipeline. It's building the cage the LLM operates safely inside.**

SECTION 11

# Measurable Outcomes

Vague outcomes are not outcomes. Here is the before and after, measured on the live system, with no rounding in the flattering direction.

| DIMENSION | BEFORE | AFTER |
|---|---|---|
| Human time per deployment | ~1–3 person-days of senior work | **~0 — one form submission, unattended** |
| Wall-clock per deployment | Hours to days, with debugging | ~20–40 min, dominated by the ModSecurity compile |
| Behavior on a failed step | Whole pipeline stops, or worse, continues on a broken base | **Bounded self-heal, cap 5, then human escalation** |
| WAF attack-block score (live) | 10 / 13 — not production-ready | **12 / 13 — "FULLY PRODUCTION READY", both domains** |
| False positives on legitimate traffic | — | None observed (home, API, assets, multi-param queries all served) |
| Proof of "live" | Absence of visible errors | **Independent HTTP request from n8n confirms response** |
| Compute cost per deployment | — | **~$6 (≈ 1,000 ETB)** |

*Measured on the target racknerd server against a live split-domain deployment. The WAF score is the exact output of the thirteen-attack suite the pipeline runs as its gate.*

Two of these deserve emphasis. The move from 10/13 to 12/13 on the WAF is not a cosmetic bump; 12/13 is the exact threshold at which the test reports a production-ready firewall, and it was reached by tuning the anomaly-scoring model rather than by weakening any check. And the shift in failure behavior — from "stops or corrupts" to "repairs within a bounded budget, then asks a human" — is the difference between a script you have to babysit and a system you can trust to run while you sleep.

SECTION 12

# The Value Analysis

The headline is a roughly 300× local cost reduction. Here is the arithmetic behind it, the global picture around it, and an honest account of where the value is real and where it has limits.

## The cost saved

A contractor handling this end-to-end deployment in Ethiopia can charge up to 300,000 birr. A run of this workflow costs on the order of 6 US dollars — about 1,000 birr — in compute. That is a reduction of roughly **300×** on the same outcome, computed at the local currency the buyer actually pays in.

| COMPARISON | MANUAL / CONTRACTOR | PROJECT NSD | REDUCTION |
|---|---|---|---|
| Local (Ethiopia) | 300,000 ETB | ~1,000 ETB (~$6) | **~300×** |
| In USD terms[1] | ~$1,860 | ~$6 | **~310×** |
| Global market band[2] | $2,000–$5,000 | ~$6 | **~330–830×** |

*USD conversion at the May 2026 rate of ~161 ETB/USD. The global band is Upwork's quoted range for freelance infrastructure-automation projects.*

The number that keeps the claim honest is the compute breakdown. The ~6 dollars is dominated by virtual-server runtime — the control plane and the target machine, running for the twenty-to-forty minutes a build takes, with the ModSecurity compile eating most of it. The AI portion is close to a rounding error. At Qwen 3.7's pricing of `$0.78` per million input tokens and `$3.90` per million output,[9] and with the agent firing *only* on a failed role and capped at five attempts, a clean deployment spends essentially zero on the model, and even a run that triggers several self-heal cycles spends cents. The AI is the cheapest part of the system. Its value is not in tokens consumed; it's in the senior judgment call it removes from the critical path.

## The time saved

Time is the larger story, and the harder one to fake. A manual build of this stack is one to three days of focused, senior work — not because any single step is long, but because the steps are interdependent and the failures surface late. The workflow reduces the *human* time to the minute it takes to fill a form, and the wall-clock to well under an hour of unattended compute. For a team that stands up servers regularly, that is the reclaiming of the scarcest resource they have: the attention of the one person who knows how to do this correctly.

This is not a claim invented for a case study; it is the shape of value the wider field is measuring. Organizations applying AI to operations report mean-time-to-resolution improvements of **40–60%**, and teams that adopt self-healing infrastructure report incident-resolution-time reductions around **65%**, with mature deployments resolving a large share of issues autonomously.[3] The 2024 DORA research adds the necessary caution: AI raises individual throughput, but larger, faster changes can hurt delivery stability if they aren't governed.[10] That caution is precisely why this system caps the AI at five attempts and puts a human at the end. Speed without a backstop is a liability. Speed with one is leverage of the only kind worth having.

## Where the value has limits

The honest boundary: this system does initial deployment, and it does it well. It is not — yet — drift detection, ongoing patching, or disaster recovery. The ModSecurity compile is slow. The reusable model is validated on one platform, Ubuntu, and extends to others only through an explicit compatibility matrix, not a universal promise. The value is real and it is bounded, and saying so is the difference between a case study and a sales page.

SECTION 13

# Reflections

The capability on display here — supervised, self-healing, security-gated deployment — is the sort of thing that has lived inside platform teams at companies with platform-team budgets. There is no longer a good reason for that.

Everything in this system is either open-source, self-hostable, or priced in cents: n8n on a server you own, Ansible from Red Hat, ModSecurity and the OWASP CRS from the community, an inexpensive model doing a small, bounded job. The enterprise version of this is a six-figure platform investment and a team to run it. This version is a workflow, a control plane, and a well-drawn cage around an AI, and it fits on two modest virtual machines. That gap — between what enterprise-grade operations cost and what they need to cost — is the whole point. Affordable enterprise AI is not a slogan here; it is the measured outcome of the build.

The real challenge with this workflow isn't teaching an AI to fix a broken deployment. Models are good at that, and getting better weekly. The real challenge is building the structure around the AI that makes its help trustworthy — the allowlist that bounds what it can touch, the counter it doesn't control, the human it has to answer to, the independent check that proves the result. The intelligence was the easy part. The governance was the engineering.

If you're running deployments by hand, or paying senior rates for work a governed system could do while you sleep, that's the conversation worth having. And if you'd rather read the code than take my word for it, the sanitized workflows and playbooks that produced everything described here are published alongside this document.

APPENDIX

# References

Sources cited inline throughout. Figures were current as of retrieval in July 2026; exchange rates and market projections change.


1. Ethiopian Birr exchange rate and depreciation trend (USD/ETB ≈ 161.26, May 2026; −18.68% over 12 months). tradingeconomics.com/ethiopia/currency
2. Freelance DevOps engineer rates and project bands (median $60/hr; infrastructure automation $2,000–$5,000). upwork.com/hire/devops-engineers/cost
3. AIOps and self-healing operations impact (MTTR −40–60%; self-healing resolution-time −65%; ~80% of IT outages attributable to human error). teamcomputers.com/blog/aiops-roi-automation-report-2026
4. IBM Cost of a Data Breach 2025 (global average $4.44M; U.S. $10.22M; lifecycle 241 days). ibm.com/reports/data-breach · via bluefin.com
5. AIOps market size and growth ($12.4B in 2024 → $123.1B by 2034, CAGR 25.8%). market.us/report/ai-operations-aiops-market
6. n8n platform, licensing, founding, integrations, and 2025 Series C. en.wikipedia.org/wiki/N8n · n8n.io
7. Ansible architecture: agentless, idempotent, playbooks/roles/modules/inventory (Red Hat). spacelift.io/blog/ansible-configuration-management
8. OWASP Core Rule Set paranoia levels and anomaly-scoring model. coreruleset.org/.../working-with-paranoia-levels
9. Qwen 3 / 3.7 Max API pricing ($0.78 per 1M input, $3.90 per 1M output tokens). pricepertoken.com/.../qwen-qwen3-max
10. 2024 DORA State of DevOps: AI's effect on throughput vs. delivery stability. getdx.com/blog/2024-dora-report