TECHNICAL WRITEUP · AVYXEN LLC

Building a system that fails loudly instead of quietly

Security patterns, real bugs, and the verification discipline behind an n8n lifecycle automation build: what held up, what broke, and how each break was actually caught.

System: n8n · Airtable · WordPress/WooCommerce · Stripe · DocuSign · Gmail · Resend · Slack

Author: Tony Ngo, AI Automation Specialist

Role: design, build, and audit of the full pipeline

> This writeup documents a portfolio demonstration system (PST Academy) built to a standard suitable for review in an interview, not a live client deployment. Names attached to test records throughout (Bruce Banner, Peter Parker, Steve Rogers) are fictional test personas, not real customers. Workflow internals (raw JSON, record IDs, credentials) are intentionally omitted; what follows is the reasoning, not the exportable source.

01

What this system does

One board, one row per person, moved forward by real events, not a demo of buttons: a system that has to survive real webhooks, real race conditions, and real human typos.

PST Academy sells a certification course in three tiers. Behind the storefront sits a lifecycle board, one row per prospective student, one column per stage: *Inquiry* → *Contacted* → *Registered* → *Agreement Signed* → *Access Granted* → *Attended/CE Issued.* Nine n8n workflows run the system: six move a row forward as real events happen (a form submission, a completed order, a signed agreement, a logged phone call, a certificate of attendance), two are supporting paths that deliberately advance no one (a signing link redirect that reissues expired links, and a staff triggered certificate resend), and one watches the other eight and reports any failure. Nothing moves a card by hand except the parts that genuinely require human judgment: following up, correcting a mistake, deciding a refund.

Built on WordPress and WooCommerce for this demo, but that's the swappable part, not the core. The trigger layer, whatever fires "a form was submitted" or "an order was paid", is deliberately isolated from everything else: the upsert that dedupes by email, the no regression guard, the atomic claim pattern, the stage model itself. The design intent is that pointing the same board at Shopify and Stripe instead of WooCommerce, or Jotform instead of WPForms, would only mean swapping the trigger nodes; the security and reliability patterns underneath wouldn't need to change. That's a design property, not a claim proven here: this build has only ever been tested against the platforms it was actually built on.

On this specific platform, that trigger layer is a small custom bridge: a automatically loaded WordPress plugin that listens for the real WordPress or WooCommerce event and forwards it to n8n as a webhook call. It's the only custom code sitting between the storefront and the automation itself; everything past that boundary is the reusable core described above.

Picture one applicant, Maria, moving through it: she fills out an inquiry form and her row appears at *Inquiry* within seconds, a Slack ping and an email already in the team's queue. A staff member reaches out (a call, an email, or a quick reply in the site's Tidio chat) and logs the attempt; her row slides itself to *Contacted* without anyone dragging a card. She buys the Online Training tier; the same row (not a duplicate) advances to *Registered*, pulling the product she actually bought, not the one she originally asked about. She signs electronically the participation agreement over a link delivered by email; the row advances again, automatically, once DocuSign reports the signature back. She attends, staff mark her attended, and a certificate generates itself with the correct CE hours for her tier.

Inquiry isn't a required gate Maria has to pass through first. It's simply the earliest event the system knows how to react to. Plenty of real applicants never submit that form at all: they call the number on the website, message the team on social media, meet someone from the team at an event, or just decide to buy outright. When an order arrives from someone the board has never seen, the system doesn't wait for a missing inquiry that will never

come. It creates that person's row fresh, straight at *Registered*. The stages describe where someone stands, not a checklist everyone must complete in order.

Every one of those automatic hops is a place where something can go wrong: an attacker can try to forge the event, two genuine events can race each other and leave the board in a state nobody intended, or a staff member can enter something that's present but wrong, an attendance date typed a day off, a checkbox clicked before the agreement was actually signed. The rest of this document is about all three: the patterns built in from the start to catch each one, and, just as importantly, the real failures that got through anyway and how they were actually caught, not just how they were fixed.

That second half is the part worth reading closely. Anyone can claim a system is secure and reliable. What follows is the harder, more honest version: specific bugs, specific root causes, and specific evidence that the fix actually worked, including two bugs that survived multiple earlier passes before a human reading the actual output caught them.

02

Security & reliability patterns

The patterns that make "two events arrive at once" and "someone forges a request" boring, survivable problems instead of live incidents.

Proving a request is genuine: HMAC signature verification

The order completed webhook is a public URL. Without a check, anyone who discovered it could POST a forged "order paid" event and walk straight to a real certificate with no actual payment. WooCommerce and n8n share a secret known only to those two systems, closer to a wax seal than a password. WooCommerce signs every webhook payload with `HMAC-SHA256(payload, secret)` and attaches it as a header; n8n recomputes the same signature independently and compares before any business logic runs. A mismatch is rejected outright, no logic executed at all.

Answering "have I already done this?": the atomic claim pattern

Two webhook deliveries can arrive within milliseconds of each other: WooCommerce firing "order placed" and "order completed" back to back is a real, observed case. A naive "look up, then create or update" sequence leaves a window where both executions see "not found" and both create a duplicate row. The fix replaces that two step sequence with a single upsert call against a shared table keyed on a unique event string, and lets the destination

database, not n8n, arbitrate the race: whichever write lands first creates the row; every later write for the same key just matches and updates it.

```
CONFIRMED UNDER GENUINE SIMULTANEOUS FIRING, NOT SEQUENTIAL
TESTING

PATCH  Processed Events table
{
  "performUpsert": { "fieldsToMergeOn": ["Event Key"] },
  "typecast": true,
  "records": [{ "fields": { "Event Key": "agreement-claim-{{ bi
}

If created_records.length > 0  ->  this execution won the claim
If created_records.length == 0 ->  someone already won it  ->
```

One shared table covers every stage's idempotency check: a prefix on the key (`inquiry-...`, `agreement-claim-...`, `ce-issued-claim-...`) is what keeps different claim types from colliding. The same mechanism does double duty later as a **notify once** guard, not just a create once guard (see the date anomaly guard below).

Never moving a record backward: no regression guards

A duplicate or late arriving event must never undo real progress already made. Before any stage advancing action runs, the workflow checks where the record currently sits and only proceeds if the move is genuinely forward. The shape of that check depends on whether the guarded action itself moves the stage field: sending an agreement, for instance, doesn't change stage at all under this system's design, so stage alone can't tell "already sent" from "not yet sent." That gap is exactly what caused a real early bug: toggling an order's status resent a brand new signing envelope every time. The fix pairs the stage check with a dedicated timestamp field that only gets set once the action actually happens.

Catching bad data, not just missing data: the date anomaly guard

A checkbox and a date field are easy for staff to get right in spirit and wrong in the details: attendance marked before the agreement was actually signed, or a future date typed by mistake. The certificate issuance guard checks logical consistency, not just presence: attendance date can't precede the signed agreement, and can't be in the future. A failure here doesn't just block the certificate. It fires a notify once alert (via the same atomic claim

mechanism above) naming the specific problem, so staff know exactly what to go fix rather than just that something's wrong.

Neutralizing injected input: Slack and Airtable

A public form field is attacker controlled text. Slack's `<!channel>` mention syntax is real, server interpreted markup. A customer's name field containing that literal string, pasted unescaped into an internal notification, would ping the whole team at once the whole team. Every customer originated field going into a Slack message is escaped (`&` / `<` / `>`) before it's interpolated, verified with a real injected payload sent through a live order: confirmed to render as inert literal text, never as an executed mention.

The same class of problem exists one layer down: Airtable's search filters are themselves a small formula language, and an unescaped quote character in a search value can break out of the intended string and get interpreted as formula syntax, the same underlying failure shape as SQL injection, against a different query language. The lookup that matches a signed agreement back to its contact record escapes that quote character. Proving it mattered meant tracing the real path a malicious value would have to travel to reach it: a customer would need to check out with a quote character in their billing email, since that's the only place this value originates. A real checkout attempt with exactly that character was rejected outright by WooCommerce's own email validation before an order could even be created, and separately, no mainstream email provider delivers to an address shaped that way in the first place, so even a different intake path couldn't get a real signed agreement in front of a real inbox to complete the loop. The escaping stays in place as cheap defense that is correct in principle, but the real finding is stronger than "the code handles it": the legitimate path to exploit it is structurally closed, verified against the real checkout, not assumed.

Signed, expiring links with constant time comparison

The signing link a customer receives by email carries a record reference and an expiry, signed together with an HMAC over both values, not just a fixed secret alone, so tampering with either value invalidates the signature. Verification uses `crypto.timingSafeEqual` rather than plain string equality, closing a theoretical timing side channel where an attacker with very precise measurement could in principle infer a correct signature one character at a time instead of guessing it whole. All three failure modes (missing parameters, bad signature, expired link) show the same generic public message, with the real reason routed only to an internal alert for troubleshooting.

Retry, and the gap it didn't originally cover

The writes on the most expensive paths (the certificate generation chain, the signing envelope, and the stage stamps in the agreement, contacted, CE issued and both order

flows) retry automatically on failure: three attempts, a flat two second interval between them. Coverage is not universal, and it is worth stating precisely instead of rounding up to "everything retries": of the 30 data write operations across the system, **6 still carry no retry at all**: three record writes in the inquiry flow, the two flag clears in the certificate resend, and the signing session call in the redirect flow. Five of those six sit behind an idempotency or atomic claim gate, so a retry that did fire would be safe; they were simply never configured for it. That is a gap rather than a decision, and worth naming as one: the retry that exists was deliberately placed on the expensive paths, but its absence elsewhere was never reasoned about either way. The sixth is different in kind: the signing session call happens on the live redirect path with a customer waiting on the response, where a failure surfaces immediately to that person rather than disappearing into a log.

Four of those six gaps were closed after the fact, and why they mattered is more useful than the count. When an order comes in, the system writes the customer to the board in **two separate steps**: first it creates their record, then it sets their stage to *Registered*. Only the first step retried on failure. That is the wrong way round. The record is created with no stage on it at all, and the step that fills the stage in was the unprotected one.

The board groups cards by stage, so a record with no stage drops into a column called **Uncategorized**. Nothing retried that write, and nothing came back later to finish the job. So a single brief Airtable hiccup left a real customer parked there for good, their card showing an order number and a total but no stage, looking to staff like a glitch rather than a fault. All four of those stage writes now retry three times, two seconds apart.

That was not left as an argument on paper. The Airtable login was deliberately replaced with an invalid one so every write to Airtable would fail, and then a real order was placed through the real checkout, so WooCommerce sent its own genuine webhook. The step that creates the record signs in a different way, so it kept working. Only the stage write failed, which is exactly the situation being tested.

REAL RESULT: FORCED FAILURE, REAL ORDER THROUGH REAL CHECKOUT

```
Stage write (Advance Stage=Registered):  5,547 ms
Upsert on the same run, succeeded:         714 ms
  3 attempts with two 2-second waits is ~4,000 ms of waiting pl
  ~500 ms calls, which is what 5,547 ms is made of

Node's own error snapshot, captured during the failure:
  "retryOnFail": true, "maxTries": 3, "waitBetweenTries": 2000
  401 - {"error":{"type":"UNAUTHORIZED","message":"Invalid auth

Resulting board state: contact created, order number and totals
  Stage blank -> the card sat in Uncategorized, exactly as desc

Both order workflows fired on the same second; the atomic upser
  rather than duplicated, so the race produced one contact, not

Credential restored -> next real run: 2.16 s, back to normal
```

Retry narrows that window; it does not close it. A failure lasting longer than three attempts still leaves the card stuck, and the real answer is a routine sweep that looks for records with no stage and finishes the job. This build does not have one, though as of the alerting described below the failure at least announces itself instead of passing unnoticed.

And to be clear about what the test above does and doesn't cover: it proves retry fires on the two order flows. The six writes that still have no retry were not tested this way, and the earlier forced failure test in Section 04 was run on different nodes.

That said, the retry that does exist is a deliberate, honest tradeoff, not an oversight: this is fixed interval retry, not true exponential backoff, because n8n's native retry setting doesn't offer backoff natively, and at this system's real concurrency (never more than one or two overlapping events in the same few seconds) the "thundering herd" problem that backoff exists to solve simply doesn't apply here. What retry doesn't do is finish the job once all three attempts are exhausted, and for most of this build nothing announced that it had given up. That gap is closed by the alerting described next. Proving the retry actually fires, rather than just being switched on, took two separate forced failure tests: Section 04 covers the search nodes, and the test above covers the order flows.

Knowing when something failed

Retry buys three attempts. It does not tell anyone when all three are used up. A node that failed three times in a row simply stopped, and the only way to find out was to go looking. Every workflow now names a single shared alerting workflow as its error handler, so any

execution that fails anywhere sends the workflow name, the node that stopped, the error text, and a link straight to that execution, to a dedicated Slack channel and an inbox.

Two details decide how much that actually covers. The error handler is a per workflow setting, not a global one, so a workflow that does not name it fails in complete silence. That is why all eight were changed here when only two of them needed the retry fix. And because retry sits in front of the alert, a blip that recovers on the second attempt never sends anything at all; only failures that used up all three attempts reach a channel, which is what keeps the alerting attached to problems that actually persisted.

The interesting part was not the alert. It was making the alert survivable. Two of the polling workflows run every 60 seconds and three run every five minutes, so a single broken Airtable credential fails 156 times an hour. Those intervals are settings and can be changed to whatever a client wants. Alerting on each one produces 156 Slack messages and 156 emails, and the predictable result is a muted channel, which is worse than no alerting at all because it still looks like coverage. A suppression step sits between the failure and the notification, keyed on the workflow name plus the current hour: the first failure of a given workflow speaks, the rest of that hour stays quiet, and if the problem is still there an hour later it speaks again. The alerts also go to their own channel rather than the staff facing ones, since "a node threw a 401" is not something ordinary staff can act on the way "a new inquiry came in" is.

That guard deliberately keeps its state inside the automation engine rather than in Airtable, even though Airtable is where this system stores everything else. An alerting path must not depend on the thing most likely to be broken, and Airtable is exactly that: 36 nodes reach it across eight of the nine workflows, split between two credentials, so routing the suppression check through Airtable would mean an Airtable outage silences the alert about the Airtable outage. The ninth workflow is the alerting one, and it deliberately touches Airtable nowhere.

REAL RESULT: ALERTING UNDER A GENUINE CREDENTIAL OUTAGE

```
Airtable credential replaced with an invalid token, real schedu

26 real 401 failures over ~5.5 minutes, counted from the engine
  16:10  13    16:11  3    16:12  3    16:13  3    16:14  3

Alerts delivered: 6 Slack, 6 email. Counts matched exactly, not

Contact logged -> Contacted failed 6 times and alerted twice
  the second alert arrived only when the suppression window rol
  so suppression and expiry were both observed, not merely conf

Every field populated: workflow name, failed node, error text,
  execution link resolving to the real failed run
```

Three limits are worth stating plainly, because an alerting system people trust is more dangerous when it is quietly incomplete than when it is absent. First, the suppression window is a clock hour rather than a rolling one, so a failure at 14:59 and another at 15:01 will both speak. The length of that window is a setting rather than a fixed property of the design, so a client who wants to hear about repeats sooner can have it shortened. Second, Slack and the inbox are deliberately no longer a complete record: they carry the first failure of each hour by design, and the engine's own execution list is the only place holding every one. A quiet channel does not mean the problem stopped, and both messages say so in their own text rather than leaving that fact in a document nobody rereads.

Third, and least obvious, this fires on executions that *fail*. A workflow that finishes successfully while quietly doing nothing produces no alert, and the outage test demonstrated that in a way reasoning alone would not have. One workflow stayed completely silent throughout: its Airtable call sits fourth in the chain, behind a DocuSign query that returned nothing, so the run ended successfully before it ever reached the broken credential. That is correct behaviour, and also a real demonstration that coverage depends on a workflow actually reaching the thing that is broken. During an outage a workflow can stay silent while being just as unable to do its job. The fix is a scheduled health check: one workflow that calls each dependency just to see if it answers, on whatever interval a client chooses, so coverage stops depending on whether real activity happened to reach the broken one. It was left out of this build on purpose, and it is a small addition for a client who wants one.

Restricting who can even attempt a connection

The automation engine's own admin login was found, by direct test, reachable from any device on the same local network, not just the machine running it, because the default port binding wasn't loopback restricted. Fixed by binding explicitly to `127.0.0.1`. This is defense in depth in its plainest form: don't rely on login credentials as the only barrier when restricting who can even reach the login page is a separate, equally real layer. Confirmed before and after with a direct connection attempt from the machine's real network address, not assumed from the config change alone.

Worth being precise about, since it's not a universal rule: loopback only is the right call specifically because this is a single operator local build with no public webhook consumers. That same setting would be the wrong call on a real client's cloud hosted instance that needs to receive genuine public webhooks; the right control there is scoped reachability, a reverse proxy, TLS, and firewall rules, not a loopback restriction that would block the legitimate traffic the system exists to receive. The setting isn't the lesson. Knowing which deployment each setting actually belongs to is.

A platform quietly protecting itself from itself: the loopback webhook block

The commerce platform's own order webhook goes out through WordPress's built in HTTP layer, and that layer deliberately refuses to let a site call back into itself: a real SSRF

protection, blocking requests to localhost and loopback addresses by default, plus a separate restriction to only a small set of common ports. On a real client's site, pointed at a public automation URL, this protection is invisible and correct, doing exactly its job undetected. On this local, single machine build, it silently blocked every real order's webhook delivery, no error surfaced anywhere a person would normally look, with nothing to distinguish intentional platform security from a plain bug. Found in the commerce platform's own webhook delivery log, not guessed at. The Inquiry form's own bridge plugin, built earlier, never hit this at all: it happens to call a different WordPress HTTP function with no such restriction, a coincidence of implementation choice that could just as easily have gone the other way and hidden this same block behind Inquiry instead.

Fixed with two narrowly scoped filters explicitly allowing only localhost and the automation's specific port, and documented as a workaround for the local demo only, from the moment it was written, the same posture as the loopback binding pattern above: a real client's automation engine sits behind a real public or tunneled URL, so this exact block, and this exact fix, would simply never come up there.

Bounding what a polling system leaves behind: execution log pruning

Every stage that runs on a schedule rather than a webhook, checking Airtable every one to five minutes across five separate pollers, produces a permanent execution record for every single check, not just the ones that find something to do. Left unmanaged, that record grows forever. An earlier assumption that the automation engine's own default retention already covered this turned out to be exactly that: an assumption, never actually checked against the live configuration. Confirmed directly against the running system instead of trusted: retention had never been configured, and the real execution store had already grown to **7,564 executions and 154MB in just 8 days**, with no cap in sight. Fixed by explicitly configuring a 14 day pruning window rather than continuing to rely on an unverified default, then verified live against the running container's actual environment variables, not just the config file.

The lesson generalizes past this one setting: "the platform probably handles that by default" is exactly the kind of claim this whole system is built to distrust until it's actually confirmed, and this is a case where that claim was made once, sounded reasonable, and was still wrong.

Resending without regenerating

Every certificate this system issues is captured once, at the moment it's created, and kept permanently rather than treated as a one time email attachment. Staff can resend that exact file at any time by checking a single field on the student's record; a separate guard, built on the same atomic claim pattern used throughout this system, confirms the student was actually certified before anything gets sent, so the resend path can't be triggered by mistake on someone who never received a certificate in the first place.

03

Real bugs, found and fixed

This is the material worth reading most closely. Every case below is a real defect that made it past at least one earlier check. The value is in how each one actually surfaced, not just the one line fix.

A workflow reporting "success" and a workflow doing the *right* thing are different claims. n8n's own execution status only proves the first. Every case in this section is a moment where that gap mattered: a node ran clean, logged no error, and still produced the wrong real world outcome.

Rather than smoothing over a miss, every case below gets a full trace of exactly which checks ran, why each one missed it, and what changed afterward, held to the same standard throughout.

FOUNDATIONAL, REAL DUPLICATE CREATED

Two genuine webhooks, milliseconds apart, created a real duplicate customer record

FOUND LIVE, REAL DUPLICATE DATA

→ROOT CAUSED

→FIXED & PROVEN UNDER REAL CONCURRENCY

WHAT HAPPENED

A real test order triggered WooCommerce's order placed and order completed events close enough together that two separate workflows each ran their own "look up this person, then create or update" sequence at nearly the same instant. Both saw "not found." Both created a row. The result wasn't a theoretical race, it was two actual rows in Airtable for one actual customer.

ROOT CAUSE

A two step sequence, look up first, act second, is only safe if nothing else can act in the gap between those two steps. Two workflows reacting to the same order from different webhook events made that gap real, not hypothetical.

FIX AND VERIFICATION

Replaced the two step sequence with the atomic claim pattern described in Section 02: a single upsert against a shared table, arbitrated by the destination database itself rather than by n8n's own logic. Verified by deliberately refiring both webhooks under genuine simultaneous conditions, not sequential testing designed to avoid the race, and confirming both executions resolved to the exact same record. This incident is the reason the atomic claim pattern exists at all; every other idempotency check in this system uses the identical mechanism because this specific failure was seen firsthand once.

FOUNDATIONAL

A repeated status event silently resent a signing envelope every time

FOUND LIVE →ROOT CAUSED →FIXED

WHAT HAPPENED

Toggling an order's status during testing caused a brand new DocuSign signing envelope to go out on every single toggle, not just the first. A real customer hitting the same underlying event twice, for any reason, would have received a fresh signing link each time.

ROOT CAUSE

The no regression guard protecting this step checked only the record's stage, and sending an agreement doesn't move the stage in this system's design; the card sits at Registered both before and after the agreement is sent. Stage alone genuinely could not distinguish "already sent" from "not yet sent," because from stage's point of view those two states look identical.

FIX AND VERIFICATION

Added a dedicated timestamp field, set only once the envelope actually goes out, and changed the guard to check that field instead of relying on stage alone. Confirmed the honest way to state this one: the verification here doesn't carry the same forced concurrency proof as the case above, it was confirmed by retriggering the same repeat event scenario and watching exactly one envelope go out instead of several. The broader lesson generalizes past this one workflow: any guard built on "has the record reached the next stage" quietly breaks the moment the guarded action doesn't itself move the stage, worth checking for on every guard, not just this one.

HIGH SEVERITY, SELF CAUSED

The instinct that fixed one bug quietly created its mirror image elsewhere

FOUND DURING AN AUDIT

→ROOT CAUSED

→FIXED & VERIFIED AGAINST LIVE CONFIG

WHAT HAPPENED

An early real test order surfaced a genuine problem: a contact who had inquired about one tier ended up purchasing a different one, and their board card kept showing the stale tier they originally inquired about, even after advancing to Registered. The fix at the time was sound: make the order stage always pull Product/Tier from the real order's line item, since an order is the authoritative record of what someone actually bought.

ROOT CAUSE

A later audit found that the same reasoning, always reflect the freshest data, had been applied unconditionally to the Inquiry workflow's own record upsert step as well, without checking whether an inquiry form is actually authoritative for that field. It isn't: an inquiry reflects interest, not a purchase, and interest can legitimately differ from what someone eventually buys. But the

Inquiry workflow was overwriting Product/Tier on every single inquiry submission, including from contacts already well past Inquiry, already Registered, already signed, even already certified.

WHY THIS ONE MATTERED MORE THAN A DISPLAY BUG

Product/Tier feeds directly into the CE certificate's hours calculation, 7 hours for Online Training versus 14 for the in person tier. A stray follow up question from a customer who had already paid and already been certified could silently overwrite their purchased tier on the board. That's a real path to generating a certificate with the wrong CE hours for a paying customer, not a cosmetic glitch.

FIX AND VERIFICATION

Scoped the Inquiry workflow's upsert to only ever write Customer Name, Email, Phone, and Message on every inquiry. Product/Tier was moved to the one branch that runs exclusively for genuinely brand new contacts. Confirmed directly against the live workflow configuration, rather than the original design note, since the relevant logic had physically relocated during an earlier restructure and the note no longer matched where the code actually lived.

Worth stating plainly, since it's the clearest example of this in the whole system: this bug wasn't carelessness, it was a correct fix generalized one step too far. "Always trust the newest data" has to be scoped to which source is actually authoritative for a given field, an order is authoritative for what was purchased, an inquiry form is not, and the same instinct that was right in one workflow was a real bug in another.

COVERAGE GAP

A transient DNS blip exposed a real, unprotected coverage gap

FOUND LIVE | →ROOT CAUSED | →AUDITED SYSTEM WIDE

→FIXED & PROVEN

WHAT HAPPENED

A routine test run failed with a DNS resolution error on a search node, then succeeded cleanly on the next poll a minute later: self healing, but only by accident.

ROOT CAUSE

An earlier reliability pass had added automatic retry to every write that follows a won idempotency claim, the highest value place for it, since a stuck claim silently blocks forward progress forever. But that scope, defined precisely as "downstream of a claim," didn't include the read only search nodes that run *before* a claim is even attempted. Auditing every workflow directly turned up the identical gap on four more search nodes across three other workflows.

FIX AND VERIFICATION

Retry added to all five affected nodes, one workflow at a time, each proven with a real trigger before being published live, not just configured and assumed. The practical impact of the original gap was genuinely minor (a self healing read, worst case a few minutes' delay), but the audit habit it forced, checking every node against the same failure class and not just the one that happened to fail visibly, is what actually mattered.

SILENT CONTENT BUG: MISSED 3 TIMES

A real customer email arrived with every data field blank

FOUND BY READING A REAL EMAIL →ROOT CAUSED →FIXED →REVERIFIED ON CONTENT, NOT STATUS

WHAT HAPPENED

An internal "agreement signed" notification went out with the template intact but every variable empty: Customer, Email, Order ID, Product/Tier, all blank. The node had reported success.

ROOT CAUSE

The notification template read a field directly off its immediate upstream node's output. That upstream node was a direct API write to Airtable, a documented workaround for a separate platform bug in the vendor's native node (below), and a batch style write returns its data nested one level deeper than a single record read does. Reading the shallow shape against the nested response resolved to nothing, silently, on every single run.

THE PART THAT ACTUALLY MATTERS

This exact class of bug had already been found and fixed on a *different* workflow weeks earlier. That fix never propagated to this workflow's equivalent nodes, and three separate verification passes (an initial build test, a later retroactive function check, and this session's own retry fix testing) all made the identical mistake: each confirmed the node executed successfully and treated that as proof it worked, without opening the message it actually produced.

FIX AND VERIFICATION

Template corrected to read the correct nested path, matching exactly the pattern already proven elsewhere. Reverified with a fresh, real trigger, this time by actually reading the resulting email and Slack message, not just confirming they arrived. A system wide audit followed immediately: every notification node in every workflow traced back to its real upstream data shape, checked against this exact failure pattern. Clean everywhere else.

ROOT CAUSED BY ELIMINATION

A native platform node failed every real run, and the data was never the cause

FOUND BY A REAL FAILED WRITE

→THREE FIXES RULED OUT IN TURN

→CONFIRMED VIA CURL, OUTSIDE THE PLATFORM

→FIXED WITH AN ESTABLISHED PATTERN

WHAT HAPPENED

The Airtable node stamping a signed agreement's real completion data failed on every single real run with `422 INVALID_RECORDS`. Nothing about the specific record or table configuration explained it.

ROOT CAUSE

Three plausible fixes were tried and ruled out in turn, not guessed at: a genuine trailing newline character in the record ID expression, enabling the node's Typecast option, and deleting and fully recreating the node from scratch. None changed the result. The record was confirmed innocent by sending the identical payload directly to Airtable's own REST API with `curl`, bypassing the automation platform entirely: it succeeded immediately. That isolated the bug to the platform's own native node itself, not the data, not the table, not the configuration.

FIX AND VERIFICATION

Replaced the native node with an HTTP Request node making the identical call directly, the same pattern already established elsewhere in this system for this exact class of platform node unreliability. Two further issues surfaced in the same pass: a date value the native node had been silently reformatting was now rejected by Airtable's date only field until explicitly formatted, and a bearer token moved out of a node field into a proper stored credential. Verified against a real trigger, not assumed fixed once the write stopped erroring. This same pattern of calling the API directly is what produced the bug described above, where a batch

write returned its data nested one level deeper, a real tradeoff of an otherwise proven fix, not a free workaround.

SILENT CONTENT BUG

The alert built to explain a data problem couldn't describe its own most common case

FOUND DURING A CONTENT AUDIT →ROOT CAUSED →FIXED →REVERIFIED, REAL TRIGGER

WHAT HAPPENED

Found while pulling real execution data for the audit above, not while hunting for this specific bug. The date consistency guard's alert (described in Section 02) is meant to tell staff exactly what's wrong with a record so they can fix it. When the most likely real world trigger fired, a signed date field that's simply blank, the alert rendered the literal text "Invalid DateTime" and an empty reason. The one alert built specifically to be informative was, in its single most common case, informative about nothing.

ROOT CAUSE

The date formatting logic assumed the field would always contain a parseable date and never checked for blank. The reason message logic had exactly two branches, "too early" and "too late," with no branch at all for "missing."

FIX AND VERIFICATION

Added an explicit blank check: the alert now reads "not set" in plain English and states the reason as "Agreement Signed Date is not set." Retriggering the exact scenario required understanding that this alert's own duplicate suppression key is scoped to the specific date value involved. The first retry attempt was correctly blocked as a duplicate of the original broken alert, requiring a genuinely new date value to produce a fresh one. The second

attempt confirmed the fix, independently read and confirmed against the real Slack and email output.

SYSTEM WIDE, CAUGHT BY DIRECT QUESTION

A platform default quietly survived a fix that looked complete

CAUGHT BY READING A REAL MESSAGE

→ROOT CAUSED IN VENDOR SOURCE

→CONFIRMED SYSTEM WIDE

→FIXED ON ALL INSTANCES

WHAT HAPPENED

An earlier pass had turned off Gmail's default attribution footer, "This email was sent automatically with n8n," on every outbound email. Weeks later, a real Slack notification still carried its own version, "Automated with this n8n workflow," complete with a live link back to the workflow itself. The two looked like the same feature, already handled by the same earlier fix; they weren't.

ROOT CAUSE

Confirmed directly in the platform's own installed source rather than guessed: Gmail and Slack attribution are two entirely separate settings, differently named under the hood, `appendAttribution` for Gmail versus `includeLinkToWorkflow` for Slack. Slack's defaults to *on* whenever it isn't explicitly set. Turning off Gmail's setting was never going to touch Slack's. The original fix simply didn't know the second setting existed.

FIX AND VERIFICATION

Every Slack sending node across every workflow was checked directly against its saved configuration, confirmed the setting was unset (and therefore defaulting on) on all of them, not assumed from the one instance that surfaced it. Fixed uniformly, proven live on one workflow with a real trigger, the remaining instances

verified by direct configuration diff rather than each independently retriggered: a deliberate, explicit tradeoff for a change that is one generic platform toggle with identical behavior everywhere, not workflow specific logic, rather than a shortcut taken unilaterally.

ACCESS CONTROL GAP, CAUGHT BY A DOCUMENTATION QUESTION

The signature gate had a payment shaped hole in it

CAUGHT WHILE WRITING CLIENT FACING DOCUMENTATION

→ROOT CAUSED IN PLATFORM CONFIG

→CONFIRMED ON A REAL ORDER

→FIXED AND REVERIFIED END TO END

WHAT HAPPENED

Course access in this system is deliberately gated behind a signed agreement, not behind payment. For a hands on certification course like this one, releasing access to the course material before a signed waiver is on file is a genuine legal exposure risk, not a a nice extra. That rule lived correctly in the design documentation and in the board's own stage sequence: the "Access granted" stage genuinely only fires once a real signature has come back. But for the Online Training tier specifically, the commerce platform underneath the board had its own, entirely separate delivery mechanism for downloadable products, and it was still switched on. The moment an order was paid, the platform handed the customer a valid download link directly, right on the checkout confirmation page, before an agreement had even been sent to them, let alone signed. The board was telling the truth about what the automation had done. It just had no visibility into, or control over, what the customer could already do.

ROOT CAUSE

Two settings, confirmed directly rather than assumed: the product itself was flagged as a downloadable file, and the platform's own

global "grant access after payment" setting was switched on. Both predate the automation layer entirely and sit in the commerce platform's own configuration, not in anything the signature tracking workflow reads or writes. The gate the design called for existed only on the board; the actual file delivery was never wired through it at all.

FIX AND VERIFICATION

The commerce platform's native downloadable file delivery was switched off for this tier entirely, closing the leak at its source rather than trying to outrace the platform's own timing. A new step was added to the signature tracking workflow so the course material is emailed directly, and only once the signature is genuinely confirmed, the same point the board itself already advances to "Access granted." Before testing live, the change was audited on its own terms: confirmed it inherits the workflow's existing duplicate send protection, runs independently of the team's own internal alert so one failing can't block the other, and checked for and cleared any leftover access grants already issued under the old behavior, since closing a leak going forward isn't the same claim as confirming nothing already got through in the meantime. Verified on a real order end to end: no download appeared anywhere in the checkout flow, and after a real signature came back, the course material genuinely arrived by email, confirmed directly rather than assumed from a clean execution status.

The pattern across all nine cases: none of them were caught by a status check, a structural review, or a "the node ran without erroring" pass. Every one was caught by checking the actual real world result, real records, real messages, genuine concurrent load, rather than trusting that a clean execution status meant the outcome was correct. That's not a coincidence. It's the specific limitation Section 04 addresses directly.

04

Verification discipline

A "workflow tested" claim is only as strong as what triggered it and what was actually inspected afterward. Three specific habits, and what a real audit can honestly claim to prove.

An upstream "success" is not the outcome you need

A signing platform reporting an envelope as "sent" via its own API is not the same claim as "the recipient's inbox actually received it," a real, observed gap in a sandbox environment specifically. The fix wasn't to trust the platform's own notification more; it was to stop depending on it. A fresh, direct signing link is generated on demand and delivered through a channel already independently verified to work, rather than relying on the third party's native delivery. The general principle: when an upstream system's own success signal can't be fully trusted, own the delivery of the thing that actually matters.

"Not found" is a normal outcome, not an error

By default, the automation engine treats a search returning zero results as a failure worth halting the entire run for, which breaks any lookup where "this person is new" is the ordinary, common case, not an edge case. Every lookup node in this system is explicitly configured to output a clean, structured "nothing found" result instead of halting, with a downstream check that treats presence of a real record ID as the actual signal. The same idea recurs throughout this system in different forms: *not every "nothing happened" outcome is a failure.* The system needs an explicit way to represent "checked, found nothing, that's fine," separate from "something actually broke."

A field can silently store the wrong thing while looking fine

A numeric type field given a not numeric value doesn't error in this system's database layer; it silently stores zero, with no visible signal in either the database's own interface or the automation canvas. Caught only by checking the raw stored value directly via API, not through any UI. No coded guard was built for it, deliberately: the real production source for that field is always genuinely numeric, so the exposure only exists during manual test data entry. The right response to a risk that only exists in testing is a testing habit, not a workflow level guard against a threat that has no real path to production.

Proving a fix, not just configuring one

Every real trigger through the retry protected nodes in Section 02 had, up to this point, happened to succeed on the first attempt: genuine proof the retry configuration doesn't break anything, but zero direct proof retry actually *fires* under a real failure. Closing that gap meant deliberately breaking something on purpose: a genuinely invalid credential was swapped onto a live node that was already published, and the real scheduled trigger was left to hit the real API with it, no synthetic payload, no mocked response.

```
REAL RESULT: FORCED FAILURE, LIVE SCHEDULE TRIGGER

Two consecutive real failures: genuine 401 from the live API, n
Each failed run: 5–7 seconds
  (a single normal attempt completes in well under 1 second eve
   in this system; the multi-second duration is exactly what 3
   spaced 2 seconds apart would produce)
Node's own error snapshot: retryOnFail + waitBetweenTries confi
  during the failure, not inferred from timing alone

Credential restored -> next real scheduled run: ~2.2 seconds, b
```

"The retry setting is present and correctly configured" and "the retry mechanism actually retries when something really fails" are different claims requiring different evidence. This is what supplies the second one.

What a targeted audit actually proves, stated precisely

After the blank fields bug in Section 03, every notification node in every workflow was traced back to its real upstream data shape and checked against that exact failure pattern, system wide. It came back clean everywhere else. That is a true, meaningful, and fully verified claim, but it is a narrower claim than "the system has no more bugs," and the difference matters. An audit scoped to one specific, well understood failure pattern cannot speak to a failure pattern nobody has hypothesized yet. Overstating what a targeted audit proves is its own quiet form of the same mistake this whole section is about: mistaking "the check passed" for "the thing is actually fine."

9	**2**	**1**
real bug cases in Section 03 alone, found	independent system wide audits run after the first one	deliberate live failure injection test, forced on purpose

by checking actual results, not status		

05 Left unfixed, on purpose

Fixing every finding reflexively isn't more thorough. It's a different kind of carelessness, one that can introduce a new problem while chasing a theoretical one. Triage is itself the skill.

Seven real findings are documented here rather than fixed or hidden. Four are deliberate triage calls: real, low priority tradeoffs weighed and consciously left as is. Three are open gaps: real limitations of this system at its current scale, honestly named along with exactly what would need to change before they'd matter. The point of naming all seven is the same either way: a real security and reliability review always turns up more than gets fixed on the first pass, and being precise about which findings were triaged versus which are still genuinely open is more trustworthy than a report that implies everything got addressed.

Plain comparison instead of constant time, on the webhook checks

`THEORETICAL, LOW PRACTICAL RISK`

The signing link verification (Section 02) uses a constant time comparison to close a timing side channel: a naive equality check stops comparing at the first mismatched character, so a wrong guess that happens to share more characters with the real secret takes a fraction of a second longer to reject than one that's wrong from the very first character. Comparing a guess of `XBCDE` against a real secret of `ABCDE` fails instantly at position 1; comparing `ABCDX` fails only at position 5, after four correct characters in a row. Measured precisely enough, across enough attempts, that timing difference can let an attacker recover a secret one character at a time instead of guessing the whole string at once, which is exactly what a constant time comparison prevents by always checking every character regardless of an early mismatch. The separate webhook signature check does not use this protection. It appears in two workflows, both order flows, and in each one it is a plain equality comparison inside a visual condition node.

Why: the signing link check already required custom code for its HMAC math, so constant time comparison was effectively free to add there. Rebuilding those checks as custom code *solely* to add this one protection would mean duplicating the comparison in two places and keeping them in sync, since n8n code nodes cannot import each other, trading two simple, readable visual checks for real added complexity, against a risk that requires an attacker with unrealistically precise timing measurement against a low value target. Documented and left as a known, explainable inconsistency rather than silently different for no stated reason.

Form field mapping by type, not by ID

`NOT A LIVE BUG: A TRIPWIRE`

The inquiry form's mapping from field to data matches by field *type* (name, email, text, select) rather than the form builder's internal numeric field ID. The form builder in use is known to silently reassign those numeric IDs whenever the form is edited, with no warning: an based on ID mapping ("grab field #14 for the email") would misroute data the next time someone makes a routine edit, with no visible error. Based on type mapping ("grab whatever field is the Email type") survives that renumbering, since it never depends on the number at all.

Why: based on type mapping trades that failure mode for a narrower one. It only works cleanly because the form currently has exactly one field of each type. If a second field of the same type were ever added, for instance a new "How did you hear about us?" text field alongside the existing Message text field, the mapping would no longer have one obvious answer for "the text field" and could silently mismatch the two, again with no visible error. The fix that would close *that* gap (going back to based on ID mapping) reopens the original one it was built to avoid, so this is left as a documented tripwire for if the form is ever restructured, not treated as something to preemptively rebuild against a risk that doesn't exist in the form as it stands today.

No reconciliation if the automation process dies mid execution

`REAL GAP AT PRODUCTION SCALE`

Every guard in this system assumes the automation engine's own process stays alive between the moment a claim is won and the moment the guarded action (sending an envelope, stamping a record, issuing a certificate) actually finishes. Concretely: the claim gets marked "handled" in one step, then the real action, say, emailing a certificate, happens in a later step. If the whole process were killed in the gap between those two steps, for instance by a host's out of memory killer reclaiming resources on a shared server, the "handled" mark is already permanently saved, but the certificate never actually went out. The process restarts normally afterward and picks up new events fine; the one interrupted mid flight is left silently marked done forever, with nothing revisiting it and no error thrown anywhere.

Why: this has never actually happened on this system, and forcing it deliberately (killing the process mid execution, repeatedly, to test recovery) is a meaningfully more invasive test than the credential based failure injection already run in Section 04. At this system's real scale and traffic, the odds of a process crash landing inside that specific narrow window are low enough that building a reconciliation mechanism now would be solving a problem that has never once occurred, at the cost of real design and testing effort. Named here rather than left silent, because a real paying client running meaningful volume through a system like this would need two real answers before launch: an alert the moment the process itself dies or restarts, since that's the harder won information, nobody watching for it means it's discovered from a customer complaint instead of in real time, and a periodic job that separately reconciles any claim older than a threshold with no completed action. That alert also needs its own routing and noise handling to actually be useful: sent to whoever provides technical support, not the client's own staff facing channels, since "the process restarted" isn't something ordinary staff can act on the way "a new inquiry came in" is, and triggered on genuine state changes rather than every individual restart, so a process dying repeatedly in a short window produces one clear escalation instead of a flood of identical pings nobody reads.

That routing and noise handling now exists, and was proven under a real outage, but for execution failures only (Section 02): a dedicated alerts channel separate from the staff facing ones, and an hourly

suppression key so a repeatedly failing workflow produces one message rather than a stream. Neither of the two answers above is closed by it. An error handler that lives inside the automation engine cannot fire when the engine itself is what died, so catching a process death still requires a watcher outside the engine, and the reconciliation sweep remains unbuilt.

This exact category of risk was consciously accepted again, not overlooked, when the access granted fix in Section 03 was built: if the stored course file ever became unreachable mid delivery, that new step would fail after its retries with the same silent, unwatched outcome described above. Building one off recovery just for that single step, while every other step in this system carries the identical exposure, would make that one path inconsistently safer rather than actually safer, so it was deliberately left to the same, single future fix named here.

Rate limits and OAuth refresh under real burst load

`UNTESTED AT SCALE`

None of Airtable's, DocuSign's, or Google's rate limit (429) responses, or the token refresh behavior of their OAuth based integrations, have been tested under genuine concurrent burst load. Every real test this system has seen involved one or two overlapping people, never dozens hitting the same integration in the same second.

Why: building and load testing for burst traffic this system has never actually experienced, and has no near term reason to, would be solving for a scale that doesn't exist yet rather than the scale that does. Documented as an open question rather than assumed fine, since a real client launch (a course cohort opening enrollment all at once, for instance) is exactly the kind of event that would create genuine burst load for the first time, and that's the point at which this would need real answers: honoring a 429's `Retry-After` header directly instead of the current flat retry interval, and confirming each OAuth integration refreshes cleanly under concurrent use rather than serializing or failing.

Google’s test mode OAuth tokens expire every 7 days

`REAL INCIDENT, PLATFORM CONSTRAINT`

The Gmail credential this system uses genuinely stopped working mid session, mid test, with a "needs to be reconnected" error, and the same thing happened again days later to the Google Drive and Slides credentials as well. Root cause, confirmed directly in Google Cloud Console rather than guessed: the backing OAuth app is in Google’s *Testing* publishing status, and Google documents that refresh tokens issued to test users under that status expire after 7 days, requiring a manual reconnect. Not a bug in anything built here, a real platform limitation of the environment this demo runs in, and one that isn’t specific to any single Google integration: every credential built against this OAuth app is on the same 7 day clock.

Why: the actual fix isn’t "remember to reconnect every week." For a real client, it’s getting the OAuth app through Google’s formal verification process (a real, nontrivial step required for any app requesting Gmail, Drive, or Slides scopes in production) or switching to a based on a service account auth method that isn’t subject to the test user limit at all. Neither was warranted for a portfolio demo; both are the correct next step the moment this pattern gets reused for an actual paying client. Left here specifically because a credential that quietly stops working on a schedule is exactly the kind of gap easy to miss until it’s already broken something a customer was waiting on, and here it happened more than once, across more than one credential.

Customer order emails never leave the sandbox

`PLATFORM CONSTRAINT, DEMO ONLY`

WordPress sends its own mail through Resend, separate from the Gmail account n8n uses for everything the automation sends itself. That WordPress path has one job here: WooCommerce's order confirmation to the buyer. It has never been delivered, not once, across every order this demo has taken. The separate new order alert WooCommerce sends the shop admin has delivered every time, thirty for thirty. Root cause, confirmed against Resend's own API rather than inferred: the site sends from `onboarding@resend.dev`, Resend's shared unverified sender, which delivers only to the exact address registered on the account.

Why: the fix is verifying a sending domain and moving the from address onto it, which needs a domain this demo does not have. Nothing in the automation depends on it, since n8n deliberately never sends that message, because WooCommerce already does. It is worth stating plainly because the split is easy to miss: mail keeps arriving throughout testing, from n8n and from the admin alert, so the one message that never arrives is the one nobody is watching for.

No reminder tells staff someone is ready to be marked attended

`DELIBERATE OMISSION, NOT A LIMITATION`

Marking a person "Attended" and issuing their certificate is, by design, left as a genuine human judgment call: nobody automates deciding that a real student actually completed a real requirement, and that part of the design is correct as built. What's missing is anything that tells staff it's time to make that call. A notification *is* sent at the moment someone reaches "Access granted," but it is the signature alert, subject line "Agreement signed," announcing what just happened rather than prompting what needs to happen next. Nothing later says "this person has had access for a while and still isn't marked attended." Staff are expected to notice that on their own, by checking the column.

Why: this is a scope choice, not a technical limitation, and worth being explicit about which it is. A reminder, for instance, alerting the team if a card sits at "Access granted" past some threshold with nothing marked, would sit directly on top of the exact same notify branch pattern already used at every other stage in this system. Nothing about the platform or the architecture stands in the way of adding it. It simply wasn't needed to prove the point this demo exists to prove, and was left out on purpose rather than half built or forgotten.

In person tiers get no automated event details message

`DELIBERATE OMISSION, NOT A LIMITATION`

At Access granted, the Online Training tier gets an automated Welcome Packet email with a PDF attachment pulled from Google Drive: login details, what to expect, everything a remote student needs to start. The in person tiers get nothing at that stage. No email fires, no PDF, no automated trigger of any kind. Staff are left to manually reach out with the details an in person student actually needs: where to show up, what time, and who to report to.

Why: this is a scope choice, not a technical limitation. The branch that sends the Welcome Packet was built to close a real security bug specific to the Online Training tier (Section 03); the equivalent in person branch was never built because nothing forced it, not because it's hard. The automation itself would mirror the exact same pattern already used for the Welcome Packet: the same gate, a false branch instead of a true one, sending an event details email instead of a Drive PDF. The one genuine blocker is data, not logic: today's board has nowhere to hold per cohort event details, only a generic label field, so before this gets built the real question to answer is whether in person training happens at one fixed location or varies by cohort and city, since that decides the schema. Sequenced deliberately last on the roadmap rather than built now, since it doesn't change whether the demo proves what it's meant to prove.

06

Closing

The standard this system is held to

Security and reliability aren't a pass added at the end here. They're default behavior: secrets kept out of anywhere a browser or database dump could expose them, every public entry point authenticated before it's trusted, and, once a real race condition actually produced a duplicate record, every path where two runs could write at once rebuilt to be safe by construction rather than by hoping the timing works out. That standard held for everything built after that point.

The more useful claim is the second half: nine real bugs made it past earlier checks anyway, every one of them invisible to "did the node report success," and every one of them was still found by checking the actual result instead of trusting a clean status, and by deliberately going back over every place a bug class already proven once could recur. That's the actual discipline on display here, not a system that never broke.

For a plain English version of how this system actually runs day to day, stage by stage, see the companion client handoff walkthrough.

Tony Ngo, AI Automation Specialist Avyxen LLC

Source, workflow exports and screenshots on GitHub

All documents