Squire Install-and-Use Guide

Deploy SuiteCentral 2.0, connect it to your NetSuite sandbox, and put Sync Error AI Assist and AI Field Mapping into daily use — step by step, with copy-pasteable commands and no assumed familiarity with this codebase.

Who this is for

This guide is written for a Squire engineer who has **never worked with SuiteCentral 2.0 before**. It assumes you know SyncCentral the way a Squire operator or consultant knows it — you've seen it turn a failed sync into a NetSuite error record, and you've hand-built a field mapping before — but it does not assume you know anything about this repository's architecture, its terminology, or its code. It also assumes limited hands-on NetSuite administration experience: every NetSuite step below is an exact menu click-path, not "go configure the integration record." You will need NetSuite administrator access available to you (yours or a colleague's) and a server or cloud account to deploy to.

By the end of this guide you will have: a running SuiteCentral 2.0 server connected to your NetSuite sandbox; AI-suggested fixes appearing for real sync errors, held for your approval before anything writes back; and an AI mapping studio generating field-to-field mappings you can review and save. Nothing here auto-applies to your NetSuite data — every AI output in this guide is reviewed by a human before it touches anything.

*Watch the companion walkthrough: SuiteCentral Enhance Deployment Guide (**https://demo.kstratmdconsulting.com/squire-v2-media-demo/watch/videos/player.html?video=enhance-deployment-guide**) (5:50) — follows this lowest-disruption deployment path from prerequisites through AI Field Mapping and Sync Error Assist.*

What you're installing

SuiteCentral 2.0 sits BESIDE your existing SyncCentral pipeline — it does not replace it. SyncCentral keeps doing exactly what it does today: NetSuite scripts call out on a schedule, data moves, failures become NetSuite error records. SuiteCentral 2.0 is an add-on layer that reads from and writes back to that same pipeline, adding AI assistance and governance around it. This is what Squire's own planning docs call the Tier-2 "Enhance" wedge — the *lowest-disruption* option, not a rebuild (see `33-SUITECENTRAL-DEPLOYMENT-OPTIONS.md` and `30-SUITECENTRAL-FOR-SQUIRE.md` for the business framing). Nothing about the 40+ existing SyncCentral deployments changes when you install this — you are adding a new server that watches and assists, not swapping out the pipeline that already runs.

The two features this guide walks you through, described in SyncCentral's own terms:

- **Sync Error AI Assist.** Today, when a SyncCentral sync fails, it writes a NetSuite error record, and a person reads that record and figures out the fix by hand. With Sync Error AI Assist installed, the moment that error record is written, NetSuite calls out to the SuiteCentral server, an AI reads the error and drafts a suggested fix, and that suggestion sits in a queue until a human operator approves, rejects, or escalates it. Nothing writes back to NetSuite without that approval.
- **AI Field Mapping.** Today, when you set up a new sync, you hand-pick which source field maps to which NetSuite field, one dropdown at a time. With AI Field Mapping, you give the tool your source and target field lists and it proposes the mapping — including a confidence score and a plain-language reason for each pairing — and you review, edit, or reject each one before it's used.

Architecture at a glance:

NetSuite	SuiteCentral 2.0 Server	Operator
Sync fails; error record written		
SuiteScript webhook calls out (HMAC-signed)	1. Verify HMAC signature	
	2. AI reads error + DLP-redacts sensitive data	
	3. Suggested fix stored in suggestion queue	Operator reviews suggestion
	guardedWrite (only after human approval)	Operator approves fix
Error record updated		

(Unfamiliar terms above — HMAC, DLP, guardedWrite — are defined in the Glossary just below.)

Everything left of "operator reviews" happens automatically in seconds; everything right of it waits for a person. That approval step is not optional and cannot be turned off — Part 5 Chapter C shows exactly what the operator sees.

Glossary

Read this once before Part 1 — every later part uses these terms without re-explaining them.

Term	Plain-English definition
Tenant	The isolation boundary inside SuiteCentral 2.0 — every request, stored credential, and setting belongs to exactly one tenant ID (a string you choose, e.g. `squire_pilot`). Tenants never see each other's data.
Connector	The code module that knows how to talk to one specific system's API — NetSuite, Business Central, Salesforce, and so on. It handles that system's authentication and read/write calls.
Connector config	The stored record (credentials + settings) that tells a connector which account to talk to and how to log in — created once per integration, then referenced by an ID afterward.
HMAC signature	A cryptographic checksum computed from a shared secret plus the message body. The receiver recomputes it and compares — this proves the sender knows the secret without the secret ever being sent over the wire, and is how NetSuite's webhook proves it's really NetSuite.
Webhook	An HTTP call one system makes to another the instant something happens — here, NetSuite calls SuiteCentral's server the moment a sync-error record is created, instead of SuiteCentral having to repeatedly ask NetSuite "anything new?"
Embedded session	A short-lived, ERP-scoped login for the operator screens that (in a real deployment) run inside NetSuite or Business Central's own UI. It's identified by a session ID, not a username and password.
Operator role	The set of permissions attached to an embedded session — for example, whether that session can approve or reject an AI suggestion. Assigned when the session is created.
guardedWrite	The single code path every AI-approved change to NetSuite goes through. It re-checks governance and data-loss-prevention

Term	Plain-English definition
	rules immediately before writing, so an approval can never bypass those checks even if something upstream was stale.
Governance posture	The per-tenant policy settings — what to redact, what to block outright, what to allow — that guardedWrite and other decision points consult before acting.
DLP redaction	Automatic masking or removal of sensitive data (names, SSNs, account numbers, and similar patterns) from anything sent to an AI provider or written to a log, before it leaves the server.
Reasoning trace	A saved, retrievable record of the steps an AI took to reach a suggestion — lets an operator (or auditor) see *why* the AI proposed a specific fix, not just the fix itself.
Confidence threshold	A setting (`high` / `mid` / `low`) that controls which AI suggestions are prominently surfaced to operators. A low-confidence suggestion is still recorded; it may just be less visible in the default queue view.
TBA (token-based authentication)	NetSuite's long-lived-token authentication scheme (an OAuth-1.0a variant) — the credential type SuiteCentral uses to call NetSuite's APIs, instead of a username and password.
Integration record	A NetSuite object you create under *Setup* > *Integration* > *Manage Integrations* that identifies an external application and issues the Consumer Key/Secret pair used to request access tokens.
Access token	The Token ID + Token Secret pair a NetSuite administrator issues to a specific user and role under an integration record. Combined with the Consumer Key/Secret, these five values are what TBA signs every request with.
Secret manager	An external service (Azure Key Vault, AWS Secrets Manager) that stores encryption keys and secrets outside the application's own database. SuiteCentral requires one configured before it will store the Sync Error Assist webhook secret.
JWT (JSON Web Token)	A signed token carrying identity claims — who you are, which tenant, what role. SuiteCentral uses JWTs for server-to-server

Term	Plain-English definition
	API calls; there is no login page, so tokens are minted directly with a script (Part 4).

Prerequisites checklist

Gather these before starting Part 1. Nothing here is optional for the NetSuite-only path; the Business Central row is only needed if you also plan to walk Part 3.

Item	Who provides it	Where it's used
NetSuite sandbox account with Administrator role	Your NetSuite admin (or you, if you have it)	Part 2 — enabling features, creating the integration record and access token
Ability to create NetSuite integration records + access tokens	Same NetSuite admin	Part 2 — TBA credential provisioning
A server or cloud account (Docker host, managed Postgres, managed Redis)	Squire IT / your cloud account	Part 1 — running the SuiteCentral 2.0 server
Azure Key Vault instance (or AWS Secrets Manager)	Squire IT / your cloud account	Part 1 — required before the encrypted webhook secret can be stored (Part 5)
Anthropic API key (console.anthropic.com) and/or OpenAI API key (platform.openai.com)	You (either provider account)	Parts 5 and 6 — without at least one, AI features silently fall back to rule-based suggestions
Optional: Business Central sandbox + Microsoft Entra admin access	Your BC/Entra admin	Part 3 only — skip if you're NetSuite-only

Time estimate per part (hands-on time, assuming prerequisites above are already in hand):

Part	What it covers	Estimated time
Part 1 — Deploy the server	Docker Compose deployment, env vars, health check	~half a day
Part 2 — Connect NetSuite	Enable features, create integration record + access token, verify	~1–2 hours (includes a 5–10 minute NetSuite feature-activation lag)
Part 3 — Connect Business Central *(optional)*	Entra app registration, BC permission sets, verify	~1–2 hours
Part 4 — Tenant, users, tokens	Provision the pilot tenant, mint an operator session, mint a JWT	~30 minutes
Part 5 — Sync Error AI Assist	NetSuite custom records, HMAC secret, SuiteScript deploy, verify, daily use	~1 day
Part 6 — AI Field Mapping	Provider setup, editor walkthrough, API path, verify	~2 days (including review time)
Part 7 — Master checklist	Final end-to-end verification and troubleshooting	~30 minutes

Beta disclosure — read before you start installing

Sync Error AI Assist is beta. There is no production credential test on file for it today — everything proven so far is sandbox and fixture-based. **This guide is itself the pilot path.** Running it against your NetSuite sandbox, and later a production instance, is how that production evidence gets produced — not a prerequisite you're missing.

Known gaps you should know about going in, so nothing here surprises you mid-pilot:

- **Webhook secret rotation has a brief delivery gap.** Between updating the secret on the server and updating it in NetSuite, a webhook signed with the old secret is rejected. The 30-minute polling reconcile catches anything dropped during that window — nothing is silently lost, but a rotation should be done in a low-error window.

- **The pre-pilot error-fixture corpus is Claude-drafted, not drawn from Squire's real error history.** It proves the mechanism works end-to-end; it does not prove accuracy on your data.
- **The ≥50% AI-suggestion accept-rate target has never been measured on live data.** Treat your first weeks of real pilot data as calibration, not a pass/fail grade — a slow start is data, not failure.

AI Field Mapping's accuracy figures quoted later in this guide (Part 6) are fixture-benchmark numbers, not production numbers — they are quoted with that caveat every time they appear, and your own accept rate on your real field set is the number that actually matters.

Part 1 — Deploy the SuiteCentral 2.0 server

By the end of this part you'll have a server running and answering `https://<host>/health/ready` with a `200`. Every later part of this guide assumes that's true.

1.1 Recommended path: Docker Compose (production)

The fastest supported path is the production Docker Compose file already checked into this repository: `docker-compose.prod.yml`. It defines two services — **`integration-hub`** (the SuiteCentral 2.0 server itself; this is the exact service name every `docker compose` command below targets — it is never called `app`) and `nginx` (an optional reverse proxy/TLS terminator in front of it; skip configuring it if you already have load-balancing/TLS termination elsewhere, e.g. behind existing Squire infrastructure).

1. Get the code onto your deploy host — clone this repository, or receive a release archive. Either way you need `docker-compose.prod.yml`, `Dockerfile`, and the `src/` tree the image is built from.
2. Copy the example production env file and fill it in:

   ```
   cp .env.production.example .env
   ```

   Edit `.env` with real values, using the environment variables table below as your checklist. Docker Compose reads a `.env` file in the same directory as the compose file automatically — you do not need to `export` these into your shell.

3. Build and start:

```
docker compose -f docker-compose.prod.yml up -d --build
```

`--build` tells Compose to build the `integration-hub` image from `Dockerfile`'s `production` stage using your local source, instead of expecting a pre-built `integration-hub:latest` image to already exist locally. (If you're deploying a pre-built image from a registry instead, `docker pull`/`docker tag` it as `integration-hub:latest` first and drop `--build`.)

4. Database migrations run automatically on boot — the server calls its migration runner during startup (56 migrations as of this writing) — there is no separate "run migrations" command to remember for a fresh deploy.
5. Confirm it's actually up and ready: see 1.3 Verify the deployment below.

Port convention: the container listens on port 3000 internally; `docker-compose.prod.yml` maps that to **host port 3003** (`"3003:3000"`, chosen to sidestep a WSL2/Windows port-forwarding conflict on 3000 — see this repo's `docs/operations/PORT-CONVENTION.md`). Every command in this guide that talks to the server directly (bypassing `nginx`) uses `http://localhost:3003` or `https://<your-host>:3003`.

Other deployment paths (not walked step-by-step here — each is a legitimate alternative; pick based on what your team already operates):

- **Terraform (AWS)** — `terraform/environments/production` provisions VPC/RDS/Redis/EC2/ALB/Secrets Manager. See `terraform/README.md`.
- **Helm (Kubernetes)** — `helm/preston-test/values-prod.yaml`, autoscaling 2–10 replicas. See `helm/README.md`.
- **Bare Node, no containers** — Node 22.13+ required: `npm run build && npm start`. You're responsible for Postgres/Redis reachability and process supervision yourself on this path; migrations still run automatically on boot the same as the Docker path.

1.2 Environment variables

Three tiers, ordered by what happens if you get them wrong. Generate commands are at the end of this section.

Boot-blocking — the server exits immediately if these are wrong:

Variable	Fail-closed behavior	What to set
`NODE_ENV=production`	Unset/empty is automatically canonicalized to `production` internally, so the guards below stay on either way — but set it explicitly for clarity.	`production`
`JWT_SECRET`	Exits if it equals the built-in development default, or matches a disallowed placeholder pattern (case-insensitive `placeholder`, `change-me`/`change_me`/`changeme`, `local-only`/`local_only`/`localonly`). No code-enforced minimum length, but 64+ random characters is the shipped recommendation.	`openssl rand -base64 48`
`DATABASE_URL`	Exits if `DATABASE_URL` has no embedded credentials (i.e. isn't `postgresql://user:realpassword@host/db` form) **and** `DB_PASSWORD` is unset or literally the string `password`. If your `DATABASE_URL` already carries a real password inline — the normal case — this check is skipped entirely and you don't need a separate `DB_PASSWORD`.	A real Postgres connection string with a real password
`RATE_LIMIT_ENABLED`	Defaults to `true` if omitted, so leaving it unset is safe. The guard only fires if	Leave unset, or `true`

Variable	Fail-closed behavior	What to set
	something *explicitly* sets it to `false` in production.	
`DEMO_MODE`	Must stay unset or `0`. If `NODE_ENV=production` and `DEMO_MODE=1` (and `HOSTED_DEMO` isn't also set), the server exits — demo mode bypasses authentication, so this isn't a lint rule, it's a real safety trip-wire.	Leave unset

Feature-blocking — the server boots, but a specific feature silently doesn't work:

Variable	What breaks without it	What to set
`SECRET_MANAGER_PROVIDER` + `AZURE_KEY_VAULT_NAME` + **identity (see below)**	Without a configured secret manager, the Sync Error Assist webhook's HMAC secret has nowhere durable to live — Part 5 of this guide dead-ends.	`SECRET_MANAGER_PROVIDER=azure`, `AZURE_KEY_VAULT_NAME=<your vault name>`. Provisioning a vault: Azure Key Vault quickstart (portal).
`ANTHROPIC_API_KEY` and/or `OPENAI_API_KEY`	Without at least one, AI features silently fall back to rule-based/mock suggestions — you still get answers, just not AI-generated ones, and nothing flags that it happened unless you already know to check which provider produced a given suggestion.	A key from console.anthropic.com and/or platform.openai.com
`ENCRYPTION_KEY`	Connector credentials (NetSuite tokens, etc.) can't be encrypted or decrypted without it — every encrypt/decrypt call throws. **This is a different variable**	`openssl rand -base64 32`

Variable	What breaks without it	What to set
	from `CREDENTIAL_ENCRYPTION_KEY` (a separate key used by a different code path); this guide's `ENCRYPTION_KEY` specifically feeds connector-credential storage. Must decode to exactly 32 raw bytes.	
`REDIS_URL`	Session/cache backing for production; omit only if you've also set `DISABLE_REDIS=1`, which this guide does not recommend for a pilot.	Your managed Redis connection string

How the container actually authenticates to Azure Key Vault. Setting the two `SECRET_MANAGER_PROVIDER`/`AZURE_KEY_VAULT_NAME` vars above is not enough by itself — `SecretManager`'s Azure provider calls `DefaultAzureCredential()` with no explicit parameters, which means it walks the Azure Identity SDK's standard credential chain looking for *something* to authenticate with. Two paths, pick whichever matches your hosting:

- **Hosting on Azure compute** (an Azure VM, Container Instances, App Service, AKS, etc.) **with a managed identity attached**: nothing else to set — `DefaultAzureCredential` finds the managed identity automatically. This is the simpler path if you're deploying to Azure.
- **Hosting anywhere else** (on-prem, another cloud, a plain Docker host): create a service-principal app registration in Entra (same admin center as Part 3's Business Central app registration, if you did that part) and set three more env vars: `AZURE_CLIENT_ID`, `AZURE_TENANT_ID`, `AZURE_CLIENT_SECRET`. Guide: service principal with a client secret (Microsoft Learn).

Either way, the identity you use needs an **RBAC role or access policy on the vault itself** that permits reading and writing secrets (e.g. the built-in **Key Vault Secrets Officer** role, or an access policy with Get/Set/List secret permissions) — assign this in the vault's **Access control (IAM)** blade. Without it, `SECRET_MANAGER_PROVIDER=azure` is set correctly but every secret read/write will fail with an authorization error, not a "provider not configured" error — a confusing distinction if you don't know to look for it.

Recommended — nothing breaks immediately, but you'll want these:

Variable	Why	What to set
`METRICS_SCRAPE_TOKEN`	Without it, `GET /api/metrics` returns `403 metrics scraping requires METRICS_SCRAPE_TOKEN in production` in production. Only matters if something scrapes Prometheus metrics from this deployment.	A random token, presented as `Authorization: Bearer <token>` by the scraper
`PGSSLMODE`	Defaults to `prefer`; the shipped example production env file recommends `require` once you have a real managed Postgres with TLS.	`require`
`LOG_LEVEL`	Default is fine for a pilot; useful to bump temporarily while troubleshooting Parts 5–6.	`info` normally, `debug` while troubleshooting
`AI_COST_ALERT_THRESHOLD`, `AI_COST_HARD_LIMIT`	**Currently do nothing.** These two names appear in `.env.example` but are not read by any code in this repository — the actual per-session AI cost cap is hardcoded at $0.30 (alert) / $0.40 (hard limit) in the cost-tracking service. Listed here only so you don't spend time trying to tune a cap that isn't wired up.	Not actionable today — omit, or set for future-compatibility knowing they're inert

Generate the two keys above with OpenSSL (available on virtually any Linux/macOS host, and via Git Bash on Windows):

```shell
# JWT_SECRET — recommended 64+ random characters
openssl rand -base64 48

# ENCRYPTION_KEY — must decode to exactly 32 raw bytes
openssl rand -base64 32
```

Why some vars in `docker-compose.prod.yml` look different from others: the required vars above (`DATABASE_URL`, `JWT_SECRET`, `NETSUITE_*`, …) are written as `VARNAME=${VARNAME}` in the compose file's `environment:` list. The optional/feature-blocking vars in the table above (`SECRET_MANAGER_PROVIDER`, `ANTHROPIC_API_KEY`, `ENCRYPTION_KEY`, etc.) are written as bare `- VARNAME` entries instead, deliberately — Docker Compose renders `${VARNAME:-}`-style interpolation as an **empty string** when the variable is unset in your shell/`.env`, not as an absent key, and an empty `SECRET_MANAGER_PROVIDER` fails its allowed-values check outright instead of falling back to its default. A bare `- VARNAME` entry only injects the variable into the container when it's actually set on the host side; when it's unset, the key is simply absent from the container's environment, letting the application's own default/fallback logic take over as intended. You can see this for yourself: run `docker compose -f docker-compose.prod.yml config` with one of those vars unset and you'll see it render as `null` (absent) rather than `""` (empty string) in the output.

1.3 Verify the deployment

Readiness check:

curl -fsS http://localhost:3003/health/ready

Expected: HTTP `200` with body:

```json
{  "status": "ready",  "timestamp": "<ISO 8601 timestamp>"}
```

Migrations ran:

docker compose -f docker-compose.prod.yml logs integration-hub | grep -i "migrations"

Expected two lines, in order: `Running database migrations...` followed by `Database migrations completed`. If you only see the first line, or instead see `Failed to run migrations`, the server started but couldn't reach or write to the database — recheck `DATABASE_URL` and Postgres connectivity before moving on to Part 2.

1.4 Troubleshooting: boot-abort messages

If the container exits immediately after starting — `docker compose -f docker-compose.prod.yml ps` shows `Exited` instead of `Up` — check `docker compose -f docker-compose.prod.yml logs integration-hub` for one of these exact messages:

1. `JWT_SECRET is using a default or placeholder value in production. Set a unique, secure value.` — your `JWT_SECRET` is empty/unset (silently falling back to the built-in development default) or matches a disallowed placeholder pattern. Fix: generate a real value with `openssl rand -base64 48` and put it in `.env`.
2. `DB_PASSWORD is missing or insecure in production. Set DB_PASSWORD or use DATABASE_URL with embedded credentials.` — your `DATABASE_URL` has no embedded credentials and `DB_PASSWORD` is unset or literally `password`. Fix: use a `DATABASE_URL` in `postgresql://user:realpassword@host:5432/db` form, or set `DB_PASSWORD` separately.
3. `RATE_LIMIT_ENABLED must be true in production. Set RATE_LIMIT_ENABLED=true` — something explicitly set `RATE_LIMIT_ENABLED=false`. Fix: remove that override, or set it to `true`.

A fourth, related failure isn't one of the three `env.ts` guards above but shows up the same way (container exits, message in the logs) — the demo-mode guard writes a longer banner and also exits with status 1:

FATAL: DEMO_MODE=1 cannot be used in production environment. This is a security risk.

Fix: unset `DEMO_MODE`, or set it to `0`.

Part 2 — Connect NetSuite

By the end of this part, running `npx ts-node scripts/test-netsuite-connection.ts` against your deployed server's credentials will print `NetSuite connection test complete.` — meaning SuiteCentral 2.0 can authenticate to your NetSuite sandbox and read data through its REST API. Parts 5 and 6 both depend on this working first.

This part has two sides: four steps you do **inside NetSuite** to mint credentials (2.1–2.4), and one step you do **on your deploy host** to hand those credentials to the server you deployed in Part 1 (2.5), followed by verification (2.6).

NetSuite calls this credential scheme **Token-Based Authentication (TBA)** — an OAuth-1.0a variant using a long-lived token pair instead of a username and password. For Oracle's own click-through documentation behind Steps 2.1–2.3, see Token-based Authentication and the

step-by-step TBA tutorial. The walkthrough below is a complete, self-contained summary of those two pages — you shouldn't need to leave this guide to finish Part 2.

2.1 Step 1 — Enable the required features

Log into your NetSuite sandbox as an Administrator and navigate to:

> **Setup > Company > Enable Features > SuiteCloud tab**

Check these three boxes (they're independent checkboxes on the same tab):

- **SuiteTalk (Web Services)**
- **REST Web Services**
- **Token-Based Authentication**

Click **Save**. NetSuite can take **5–10 minutes** to fully activate these services after saving — this is the single most common source of "it should be working but isn't" in Step 2.6 below, so don't be alarmed if a connection attempt made immediately after this step fails; wait a few minutes and retry before troubleshooting further.

2.2 Step 2 — Create the integration record (Consumer Key + Secret)

Navigate to:

> **Setup > Integration > Manage Integrations > New**

1. Give it a name (e.g. `SuiteCentral 2.0 Pilot`) and an optional description.
2. Set **State** to **Enabled**.
3. Under the **Authentication** section, check **Token-Based Authentication** only — leave the other authentication checkboxes (e.g. OAuth 2.0) unchecked; SuiteCentral's connector speaks TBA, not OAuth 2.0.
4. Click **Save**.

Immediately copy the Consumer Key and Consumer Secret shown on the confirmation screen — NetSuite displays them exactly once and cannot show them again. Paste them somewhere safe now (a password manager, not a scratch file you'll lose). If you navigate away before copying them, you'll need to delete this integration record and create a new one — there's no "reveal secret again" option.

2.3 Step 3 — Create an access token (Token ID + Token Secret)

Navigate to:

> **Setup > Users/Roles > Access Tokens > New**

1. **Application Name** — select the integration record you created in Step 2.2.
2. **User** — select the NetSuite user this token acts as (see the least-privilege note below for which role that user should have).
3. **Role** — select the role for this token (again, see the least-privilege note).
4. **Token Name** — a descriptive label (e.g. `SuiteCentral Pilot Token`).
5. Click **Save**.

Immediately copy the Token ID and Token Secret shown on the confirmation screen — same one-time-only rule as Step 2.2. These four values (Consumer Key, Consumer Secret, Token ID, Token Secret) are what TBA uses to cryptographically sign every request SuiteCentral makes to NetSuite; NetSuite never sees a password.

2.4 Step 4 — Find your Account ID

You need one more value that isn't generated by a form — your NetSuite **Account ID**. Two equivalent ways to get it:

- **From the URL.** Your NetSuite sandbox's browser address bar looks like `https://1234567.app.netsuite.com` (or, for a sandbox, something like `https://1234567-sb1.app.netsuite.com`). The Account ID is the part before `.app.netsuite.com` — but written with an **underscore**, not the hyphen the URL uses: `1234567` for production, `1234567_SB1` for that sandbox (NetSuite's own convention: hyphen in the browser URL, underscore in the Account ID value you configure elsewhere).
- **From Setup.** Navigate to **Setup > Company > Company Information** and copy the **Account ID** field directly — this is the authoritative source if you're ever unsure which form to use.

Least-privilege note (role permissions). A role with **Administrator** is the simplest path and is fine for a sandbox pilot — use it if you want Part 2 done quickly and aren't worried about a sandbox's blast radius. For a tighter-scoped role instead, the user/role pair in Step 2.3 needs, at minimum: the **REST Web Services** permission (required for any TBA-authenticated REST call at all), plus **view/create/edit** permissions on the two custom record types Sync Error Assist reads and writes (`customrecord_suitecentral_sync_error` and `customrecord_suitecentral_fix_suggestion` — you'll create these record types in Part 5, so if you're setting up the role now, add those permissions once Part 5's record types exist, or come back and add them then). Everything in this Part (2.5–2.6) only needs REST Web Services and read access to the `customer` record type used by the connection test below.

2.5 Configure SuiteCentral with the five credentials

You should have five values from Steps 2.1–2.4:

Env var	Where it came from
`NETSUITE_ACCOUNT_ID`	Step 2.4
`NETSUITE_CONSUMER_KEY`	Step 2.2
`NETSUITE_CONSUMER_SECRET`	Step 2.2
`NETSUITE_TOKEN_ID`	Step 2.3
`NETSUITE_TOKEN_SECRET`	Step 2.3

`docker-compose.prod.yml` already passes all five of these into the `integration-hub` container (they're in the compose file's `environment:` block today, so there's no compose-file editing needed here — just fill in the values). Open the `.env` file you created in Part 1 §1.1 step 2 and set them:

NETSUITE_ACCOUNT_ID=1234567_SB1

NETSUITE_CONSUMER_KEY=<consumer key from Step 2.2>

NETSUITE_CONSUMER_SECRET=<consumer secret from Step 2.2>

NETSUITE_TOKEN_ID=<token ID from Step 2.3>

NETSUITE_TOKEN_SECRET=<token secret from Step 2.3>

Base URL auto-derivation. SuiteCentral does not require you to set a base URL — if `NETSUITE_BASE_URL` is absent, the connector derives one itself from `NETSUITE_ACCOUNT_ID`, DNS-normalizing it on the way: NetSuite's DNS convention for sandbox subdomains uses a **hyphen**, so an Account ID of `1234567_SB1` derives `https://1234567-sb1.suitetalk.api.netsuite.com` automatically. (The OAuth realm keeps your original underscore form `1234567_SB1` — the two are *supposed* to differ; that's NetSuite's own convention, not a mismatch. Earlier builds substituted the Account ID literally, underscore and all, which broke sandbox DNS — fixed on this branch.) Production and sandbox accounts alike work without any extra configuration.

`NETSUITE_BASE_URL` remains available as an explicit override, and it's already in `docker-compose.prod.yml`'s `environment:` passthrough list alongside the five credential vars above, so setting it in `.env` is all you need:

NETSUITE_BASE_URL=https://1234567-sb1.suitetalk.api.netsuite.com

Most deployments never need it — reach for it only if your account resolves under a nonstandard domain.

Re-create the container after editing `.env`. Compose reads `.env` when a container is *created*, not on restart — `docker compose restart` reuses the existing container's environment, so neither the edit alone nor a plain restart reaches a running container. Use `up -d`, which recreates the container when its configuration changed:

docker compose -f docker-compose.prod.yml up -d integration-hub

2.6 Verify the connection

Run these from the same host/checkout where your `.env` lives (they read `.env` directly with `dotenv`, independent of whether the server is running in Docker — you're validating the credentials themselves here, not calling the running container).

Command 1 — validate the env vars are present and well-formed:

npx ts-node scripts/validate-netsuite-env.ts

On success, the script prints a per-variable `[OK]` line and ends with:

All NetSuite credentials are present and look valid.

Next steps:

1. Run the connection test: npx ts-node scripts/test-netsuite-connection.ts
2. (Sandbox) Run CRUD validation: npx ts-node scripts/test-netsuite-crud.ts

and exits `0`. If anything is missing, still a template placeholder, or shorter than expected, the corresponding line reads `[!] <Name> (<ENV_VAR>)` with an `Issue:` explanation underneath, the script ends with `Some credentials are missing or still contain placeholders.` instead, and it exits `1` — fix whichever lines are flagged `[!]` and re-run before moving on.

Command 2 — live connection smoke test (only run this once Command 1 is fully `[OK]`):

npx ts-node scripts/test-netsuite-connection.ts

A fully successful run authenticates, fetches system info, lists up to 5 customers, and runs a basic search, ending with:

NetSuite connection test complete.

Next steps:

1. Sandbox CRUD test: npx ts-node scripts/test-netsuite-crud.ts

2. Review docs/tutorials/NETSUITE-SETUP-GUIDE.md for deeper guidance.

The script treats the last two steps (listing customers, running a search) as best-effort — a failure there prints an `[!]` line and a hint but does **not** abort the script or make it exit non-zero, since some tenants restrict search endpoints or start with zero customer records. What matters for "Part 2 is done" is that **Step 2: Authenticating...** prints `[OK] Authentication successful` and **Step 3: Fetching system information...** prints `[OK] System info retrieved:` — those two are the ones that abort the script (`process.exit(1)`) and mean TBA itself isn't working yet if they fail.

2.7 Troubleshooting

Symptom	Likely cause	Fix
`[!] Authentication failed: ...` (`401` / "Invalid login attempt" in the underlying error)	Credential mismatch, or the token/integration record was deactivated or deleted	Re-verify all five values match exactly what NetSuite showed you (no extra whitespace from copy-paste). If in doubt, delete and recreate the access token (Step 2.3) — the old one can't be un-revoked once you suspect it's wrong.
`[!] Failed to retrieve system info: ...` mentioning "Invalid account"	`NETSUITE_ACCOUNT_ID` format is wrong	Production accounts are a bare number (`1234567`); sandboxes append `_SB1`, `_SB2`, etc. with an **underscore**, matching Step 2.4 exactly — no `https://`, no trailing slash.
`[!] Failed to retrieve system info: ...` or `[!] Failed to list customers: ...` mentioning permissions	The role attached to the access token lacks REST Web Services or record-level permissions	Revisit the least-privilege note in Step 2.4 — either switch the token's role to Administrator for now (sandbox only), or add REST

Symptom	Likely cause	Fix
		Web Services + the relevant record permissions to the custom role.
Connection refused / timeout / DNS failure (not an auth or permission error)	Most often the 5–10 minute feature-activation lag from Step 2.1 hasn't elapsed yet. (Sandbox base-URL derivation now DNS-normalizes `_SB1` → `-sb1` automatically, so the old underscore-vs-hyphen mismatch is no longer a cause on current builds.)	Wait a few minutes after Step 2.1 and retry. If it persists, verify the derived base URL from 2.5 resolves from your host; set `NETSUITE_BASE_URL` explicitly only if your account resolves under a nonstandard domain.

Once Command 2 in 2.6 ends with `NetSuite connection test complete.`, Part 2 is done — move on to Part 3 (Business Central, optional) or skip ahead to Part 4 to provision your tenant.

Part 3 — Connect Business Central (optional)

> **Skip this part if you're NetSuite-only.** Business Central is a second, independent connector — nothing in Parts 4–7 requires it. If you have no Business Central tenant to connect to, jump straight to Part 4.

Honesty check before you start. `BusinessCentralConnector` carries `production` status in the connector registry — it's one of the platform's five production connectors, not a demo stub — and its OAuth2 client-credentials flow, OData v4 CRUD, and outbound-governance checks are real, tested code (`src/connectors/BusinessCentralConnector.ts`). What's **not** on file is a recorded live-credential test against a real BC tenant — the same gap NetSuite had before this guide's Part 2 became the pilot path for it. **This part is that live-credential test for Business Central.** The two `curl` commands in §3.4 below are what turn "production-status code" into "verified against a real tenant" — treat a clean run through them as real evidence, not a formality.

By the end of this part you'll have five values (Directory/tenant ID, Application/client ID, client secret, environment name, and — optionally — a company GUID) and two `curl` commands that prove SuiteCentral can authenticate to your Business Central tenant and list its companies. Part 6's Business Central mapping scenario depends on this working first.

3.1 Register an app in Microsoft Entra

Business Central authenticates callers via **Microsoft Entra ID** app registrations (Entra is the current name for what was Azure Active Directory — you may still see "Azure AD" in older screenshots). This section follows Microsoft's own service-to-service authentication guide; the full reference is Automation APIs using S2S authentication (and, for the OAuth mechanics generally, Authenticate web services using OAuth) — read either if a click-path below doesn't match what you see (Microsoft periodically reshuffles portal navigation).

1. **Register the app.**

   Entra admin center (entra.microsoft.com) > App registrations > New registration

   - **Name**: something identifiable, e.g. `SuiteCentral 2.0 BC Integration`.
   - **Supported account types**: **Accounts in this organizational directory only** (single tenant) — SuiteCentral's connector authenticates as your own tenant's app, not a multi-tenant ISV app.
   - Leave **Redirect URI** blank — the client-credentials flow this connector uses (§3.3 below) never redirects a browser.
   - Click **Register**.

2. **Create a client secret.**

   Certificates & secrets > New client secret

   Give it a description and an expiry (24 months is a reasonable default for a pilot — you'll rotate it before then). Click **Add**, then **copy the secret's Value column immediately** — like NetSuite's Consumer Secret in Part 2, Entra shows a client secret's value exactly once and cannot display it again. If you navigate away before copying it, delete the secret and create a new one.

3. **Grant API permissions.**

   API permissions > Add a permission > APIs my organization uses > search "Dynamics 365 Business Central"

   Select **Application permissions** — not Delegated permissions. This matters: the connector authenticates with the OAuth2 **client-credentials** grant (§3.3), which runs with no signed-in user, so Entra only issues app-role claims from *Application* permissions; Delegated permissions require an interactive user and would leave those claims empty for this flow. Add:

- `API.ReadWrite.All`
- `Automation.ReadWrite.All`

Click **Add permissions**, then click **Grant admin consent for <your tenant>** and confirm. Until you grant consent, every token request in §3.4 will succeed (Entra hands out a token either way) but every Business Central API call with that token will fail authorization — consent is easy to forget because nothing in the registration UI forces it.

4. **Copy the two IDs you'll need next.**

   App registrations > <your app> > Overview

   Copy **Application (client) ID** and **Directory (tenant) ID** — you'll use both in §3.3.

3.2 Grant the app access inside Business Central

Registering the app in Entra only proves *who* is calling — Business Central still needs to be told that app is allowed in, and with what permissions.

1. **Register the application inside Business Central.** In the Business Central web client, use the search icon (top right, magnifying glass) and search for:

   Microsoft Entra Applications > New

   - **Client ID**: paste the Application (client) ID from §3.1 step 4.
   - **Description**: something identifiable, e.g. `SuiteCentral 2.0 Pilot`.
   - **State**: **Enabled** — a Disabled entry looks configured but silently rejects every call.

2. **Assign least-privilege permission sets.** On the same Microsoft Entra Application Card, open the **Permission Sets** subgrid and add a purpose-scoped permission set covering the entities this pilot touches (Customer, Vendor, Item, Sales Order, Purchase Order, and the invoice tables). **Never assign `SUPER`** — it grants unrestricted access to every table and setting in the company, which is far more than an integration credential should ever hold, sandbox or not.

3. **Confirm you can reach the company list.** You already have everything needed to make your first authenticated call — jump to §3.4 below and run curl #1 (token) followed by curl #2 (companies) once you've filled in §3.3's values. The `id` field of the company you want to sync is your `companyId` (optional — see §3.3).

3.3 Configure SuiteCentral with the five credentials

You should now have four required values plus one optional one:

Env var (registry requirement label)	Where it came from
`BC_TENANT_ID`	§3.1 step 4 — Directory (tenant) ID
`BC_CLIENT_ID`	§3.1 step 4 — Application (client) ID
`BC_CLIENT_SECRET`	§3.1 step 2 — client secret value
`BC_ENVIRONMENT`	Your Business Central environment name (`production`, or a sandbox name like `sandbox`)
(optional, no registry label)	The company `id` GUID from §3.2 step 3's companies call — omit it and the connector auto-selects the tenant's first company on first authenticate()

Two different naming layers — don't conflate them. `BC_TENANT_ID` / `BC_CLIENT_ID` / `BC_CLIENT_SECRET` / `BC_ENVIRONMENT` (above) are the connector **registry's** requirement labels — `credentialRequirements: ['BC_TENANT_ID', 'BC_CLIENT_ID', 'BC_CLIENT_SECRET', 'BC_ENVIRONMENT']` in `src/connectors/connectorRegistry.ts` — a naming convention for where these values come from (your `.env` file), not literal JSON property names. What `BusinessCentralConnector.initialize()` actually reads at runtime is a **`credentials` object with camelCase keys** (`src/connectors/BusinessCentralConnector.ts` lines 74–82, reading `credentials.tenantId`, `credentials.environment`, `credentials.companyId`; lines 120–121 read `credentials.clientId` / `credentials.clientSecret` — the shape is `AuthenticationConfig.credentials` in `src/types/index.ts`). A JSON payload with a key literally named `BC_TENANT_ID` would not be read by anything — the connector has no idea that string exists.

Here is one complete connector-config JSON, using the camelCase keys the code actually reads, each one commented with the `BC_*` env var it's sourced from:

```
JSON
{
```

```
  "type": "oauth2",
  "credentials": {
    "tenantId": "<value of BC_TENANT_ID>",
    "clientId": "<value of BC_CLIENT_ID>",
    "clientSecret": "<value of BC_CLIENT_SECRET>",
    "environment": "<value of BC_ENVIRONMENT>",
    "companyId": "<optional>"
  }
}
```

Whether this JSON reaches the connector via a stored configuration record (`POST /api/configurations`) or a short initialization script in the style of the connector's own setup notes (`docs/connectors/business-central-production-setup.md` §"Initialize Connector in Code"), the shape above — `type: "oauth2"` plus the five camelCase `credentials` keys — is fixed either way, because it's what `initialize()` reads on the other end.

Token endpoint and scope (used internally by `authenticate()`, and by the verification `curl` in §3.4): the connector requests a token from `https://login.microsoftonline.com/{tenantId}/oauth2/v2.0/token` with `scope=https://api.businesscentral.dynamics.com/.default` and `grant_type=client_credentials` (`src/connectors/BusinessCentralConnector.ts` lines 122–125).

3.4 Verify the connection

There is no repo script for this the way `scripts/test-netsuite-connection.ts` covers NetSuite in Part 2 — verify Business Central with two raw `curl` commands instead. Run them from any machine with network access to Microsoft's endpoints (this doesn't need to be your deploy host).

Curl 1 — get an access token (fill in your three values):

Shell

```
curl -sS -X POST
"https://login.microsoftonline.com/<BC_TENANT_ID>/oauth2/v2.0/token
" \
  -H "Content-Type: application/x-www-form-urlencoded" \
  -d "client_id=<BC_CLIENT_ID>" \
```

```
  -d "client_secret=<BC_CLIENT_SECRET>" \
  -d "scope=https://api.businesscentral.dynamics.com/.default" \
  -d "grant_type=client_credentials"
```

Expected: HTTP `200` with a JSON body containing `access_token`, `token_type` (`"Bearer"`), and `expires_in`. Copy the `access_token` value for curl #2 — it's a long opaque JWT string.

Curl 2 — list companies with that token (fill in your tenant, environment, and the token from curl #1):

```shell
curl -sS -X GET
"https://api.businesscentral.dynamics.com/v2.0/<BC_TENANT_ID>/<BC_E
NVIRONMENT>/api/v2.0/companies" \
  -H "Authorization: Bearer <ACCESS_TOKEN_FROM_CURL_1>"
```

Expected: HTTP `200` with a JSON body shaped `{"@odata.context": "...", "value": [{"id": "<guid>", "name": "...", "displayName": "...", ...}, ...]}` — one entry per company in that environment. This is your live-connectivity proof: a 200 here means the app registration, client secret, admin consent, and BC-side permission set from §3.1–3.2 are all correctly wired together. The `id` of the company you intend to sync is the `companyId` value for §3.3's JSON if you want to pin it explicitly.

What this does and does not prove — read before Part 6. These two curls exercise Business Central's real OAuth2 token endpoint and real `/companies` OData endpoint directly — that's genuine live connectivity, not a fixture, and it's proof your Entra app/client-secret/consent/permission-set setup is correct. What they do **not** prove by themselves is that **SuiteCentral's own `BusinessCentralConnector` class successfully authenticates with these same credentials** — the curls never call `BusinessCentralConnector.initialize()` (`src/connectors/BusinessCentralConnector.ts:67`); they talk to Microsoft directly, bypassing SuiteCentral's connector code entirely.

To close that gap, the server exposes an endpoint that runs the real connector: **`POST /api/integrations/<config-id>/test`** (`src/routes/integration.ts`, mounted at `/api/integrations` behind `authMiddleware` — so it needs Part 4 §4.4's `$JWT`). It resolves your stored connector config, constructs the registered `BusinessCentralConnector` through

the connector registry, calls `.initialize()` on it with the config's `sourceAuthentication` — §3.3's JSON — and runs the connector's own `testConnection()`, returning per-side pass/fail. Two values in the config must be exact for the right connector to be selected:

- `sourceSystem` must be the string **`BusinessCentral`** — exactly that PascalCase form. `IntegrationService.getConnector()` maps that literal (and only that literal) to the registry's `businesscentral` entry; any other spelling (`businesscentral`, `business_central`, `BC`) throws `Unsupported system type` and the test reports a failure that has nothing to do with your credentials.
- `sourceAuthentication` must be §3.3's shape — `type: "oauth2"` plus the camelCase `credentials` keys.

First create the config (every field below is required by the config schema; `targetSystem` must differ from `sourceSystem`, so this uses `NetSuite` as a placeholder target with no credentials, and `isActive: false` sidesteps the at-least-one-field-mapping rule for active configs):

Shell

```
curl -sS -X POST https://<host>/api/configurations \
  -H "Authorization: Bearer $JWT" \
  -H "Content-Type: application/json" \
  -d '{
    "id": "bc_connector_test",
    "name": "BC connector live-auth test",
    "sourceSystem": "BusinessCentral",
    "targetSystem": "NetSuite",
    "sourceEntity": "customers",
    "targetEntity": "customer",
    "syncDirection": "source_to_target",
    "syncMode": "manual",
    "isActive": false,
    "sourceAuthentication": {
      "type": "oauth2",
      "credentials": {
        "tenantId": "<BC_TENANT_ID>",
        "clientId": "<BC_CLIENT_ID>",
        "clientSecret": "<BC_CLIENT_SECRET>",
        "environment": "<BC_ENVIRONMENT>"
      }
    }
  }'
```

Then run the connector test:

```shell
curl -sS -X POST
https://<host>/api/integrations/bc_connector_test/test \
  -H "Authorization: Bearer $JWT"
```

The route returns `200` with `IntegrationService.testIntegrationForTenant`'s result object, shaped:

```json
{
  "isValid": false,
  "sourceConnection": { "isConnected": true },
  "targetConnection": {
    "isConnected": false,
    "errorMessage": "<target-side auth failure>"
  },
  "errors": ["Target system connection failed: ..."],
  "warnings": ["No field mappings defined - data may not sync
properly"]
}
```

`sourceConnection.isConnected: true` is the pass signal — it means `BusinessCentralConnector.initialize()` plus its `testConnection()` authenticated against your tenant with §3.3's credentials, through SuiteCentral's own connector code this time. `targetConnection.isConnected: false` (and therefore `isValid: false`, plus a target-side entry in `errors`) is expected with the credential-less placeholder NetSuite target above — ignore it, or supply a real `targetAuthentication` if you want both sides green. If `sourceConnection` comes back `false` instead, its `errorMessage` carries the connector's own failure reason — the same underlying causes as §3.5's table.

Two narrower caveats survive even after a green run:

- **Schema browsing still isn't live.** `MetadataClient.fetchFromAPI()` (`src/connectors/businessCentral/MetadataClient.ts` lines 143–165) logs `Live BC metadata fetch not yet implemented, falling back to fixture` and unconditionally falls through to bundled fixture `$metadata` XML — a server-side connector limitation a passing connector test doesn't change. (Separately,

Part 6's mapping editor never consults the connector or its metadata at all — its per-system field catalogs are server-side fixture schemas, not connector output, until you upload your own data; see Part 6 §6.3 and the still-open schema-catalog gap in §7.3.)

- **CRUD isn't proven.** A `create`/`update` against a real sandbox company (which exercises `formatDataForBusinessCentral` and the outbound-governance gate, not just auth) is what would prove CRUD end-to-end — that's a larger step than this guide's verification scope, and the same "next evidence to produce" gap the readiness brief calls out for Business Central generally.

3.5 Troubleshooting

Symptom	Likely cause	Fix
Curl #1 returns `400` with `error: "invalid_client"` or `unauthorized_client`	Wrong `BC_CLIENT_ID`/`BC_CLIENT_SECRET`, or the secret expired/was deleted	Re-copy both values from §3.1; if in doubt, create a fresh client secret (§3.1 step 2) — the old one can't be un-revoked once suspect.
Curl #1 succeeds (`200`, you get an `access_token`), but curl #2 returns `401` or `403`	Admin consent (§3.1 step 3) was never granted, or the app isn't registered/Enabled inside Business Central (§3.2 step 1)	Revisit §3.1 step 3's **Grant admin consent** button — it's easy to skip since nothing else in the flow forces it. Then confirm the Microsoft Entra Application record in BC has **State = Enabled**.
Curl #2 returns `200` with an empty `value: []`	The Entra app has no permission set assigned in BC (§3.2 step 2), or you're pointed at the wrong `BC_ENVIRONMENT` name	Confirm the permission set assignment, and double-check the environment name matches exactly what's shown in the Business Central admin center (environment names are case-sensitive).
Curl #2 returns `404`	`BC_TENANT_ID` or `BC_ENVIRONMENT` doesn't match a real environment	Re-verify both against the Business Central admin center — a typo in either segment of the URL 404s rather than 401s, which can be misleading.

Once curl #2 in §3.4 returns a non-empty company list, Part 3 is done — move on to Part 4 to provision your tenant.

Part 4 — Create your tenant, users, and tokens

By the end of this part you'll have a provisioned tenant (`squire_pilot`), a working operator embedded session, and a working API JWT — the three prerequisites Parts 5 and 6 assume are already in hand. **There is no login page anywhere in SuiteCentral 2.0.** Every credential below is minted by running a script or calling an endpoint directly — that's expected, not a sign something is missing. This part is widely the most confusing area for a newcomer, so it moves slowly and names every value explicitly.

4.1 Three credential kinds — read this before running anything

SuiteCentral 2.0 has three distinct kinds of credential, and mixing them up is the single most common way to get stuck in Parts 5–6. None of them are interchangeable with either of the others.

Kind	What it actually is	Where you use it	How you get it
Tenant	Not a credential at all — just an ID string that's the isolation boundary. Every request, stored credential, and setting belongs to exactly one tenant.	Everywhere, implicitly — it's baked into the other two credentials below and into every HMAC key-id, config record, and audit row from here on.	You choose the string yourself when you provision (§4.2). This guide uses `squire_pilot` for every example from here through Part 7 — use that exact value unless you have a specific reason not to.
Embedded session ID (a.k.a. operator session)	A short-lived (8-hour) session identifier, `es_<random>`, scoped to one ERP platform + account + set of operator roles.	`X-Embedded-Session-Id` header on operator-UI calls, or `?embeddedContextId=<sessionId>` on the triage page URL itself. This is what Part 5's Sync Error Assist queue and Part 6's mapping editor run under.	Minted by `POST /api/embedded/host-bootstrap` (§4.3) — a *different* endpoint from tenant provisioning, and not something the tenant-provisioning script produces directly.

Kind	What it actually is	Where you use it	How you get it
API JWT	A signed, stateless identity token carrying `sub`/`tenantId`/`roles`/`permissions` claims, valid until it expires (you choose the lifetime when you sign it).	`Authorization: Bearer <jwt>` header on direct server-to-server `/api/*` calls — the kind of call you'd make from a script, not from a browser session inside NetSuite.	Minted with a short Node script and your server's `JWT_SECRET` (§4.4) — no network round trip to the server at all; you're signing it locally with the same secret the server will use to verify it.

One more value shows up along the way and deliberately isn't a fourth row in this table: provisioning (§4.2) also mints an **embedded *service* token**. That token is a machine credential, not a session — its only job is to authenticate the one bootstrap call in §4.3 that mints the real operator session above. Don't try to use it anywhere Parts 5–6 ask for `X-Embedded-Session-Id`; it will not work there. The relationship is spelled out again in §4.3 once you have both values in hand.

4.2 Provision the pilot tenant

Tenant provisioning is driven by `scripts/provision-pilot-tenant.mjs` (documented in `docs/pilot/PILOT_PROVISIONING_RUNBOOK.md`), wrapped by the `provision:pilot-tenant` npm script. It accepts **exactly one** of three mode flags — `--dry-run`, `--apply`, or `--verify` — never more than one at a time, and two of the three modes require an extra flag the script refuses to run without:

- `--apply` **requires** **`--output <path>`** — without it the script exits immediately with `--output is required for --apply` before it mints anything.
- `--verify` **requires** **`--provisioning-output <path>`** — without it the script exits immediately with `--provisioning-output is required for --verify`.

Run all three in order, from your checkout on the deploy host (or anywhere with `npm` and network access to this server's dependencies):

Step 1 — dry run. Prints the deterministic provisioning plan and touches nothing:

```shell
npm run provision:pilot-tenant -- \
  --tenant squire_pilot \
```

```
  --platform netsuite \
  --platform-account-id <NETSUITE_ACCOUNT_ID from Part 2 §2.4> \
  --dry-run
```

Review the printed JSON (`mode`, `tenantId`, `platform`, `platformAccountId`, and an `actions` list). Nothing is written to disk or to token storage yet.

Step 2 — apply. Mints a real embedded service token and writes the provisioning artifact:

Shell
```
npm run provision:pilot-tenant -- \
  --tenant squire_pilot \
  --platform netsuite \
  --platform-account-id <NETSUITE_ACCOUNT_ID from Part 2 §2.4> \
  --apply \
  --output ./pilot-provision.json
```

This is the step that actually mints something — under the hood it delegates to `npm run rotate-embedded-service-token`, the same CLI that owns all embedded-token storage writes elsewhere in this codebase. `./pilot-provision.json` is written atomically at file mode `0600` and contains `{rawToken, tokenHash}` for the new **service token**, the provisioning plan, and the four `governance.*` posture keys below. Stdout only ever prints a redacted summary (`"rawTokenRedacted": true`) — the raw bearer token never lands in your terminal scrollback or shell history. Treat `pilot-provision.json` like any other credential file: don't commit it, and if you're handing the token to someone else (e.g. whoever runs §4.3), use the same secure channel you'd use for a password.

This service token is not an operator session. It cannot be used as `X-Embedded-Session-Id` anywhere in Parts 5–6. Its only job is authenticating the host-bootstrap call in §4.3, which is what actually mints a usable operator session.

Step 3 — verify. Reads the artifact back and fails closed on any mismatch or tampering:

Shell
```
npm run provision:pilot-tenant -- \
  --tenant squire_pilot \
```

```
  --platform netsuite \
  --platform-account-id <NETSUITE_ACCOUNT_ID from Part 2 §2.4> \
  --verify \
  --provisioning-output ./pilot-provision.json
```

This confirms the artifact's `tenantId`/`platform`/`platformAccountId` match the CLI arguments you just typed, and recomputes `sha256(rawToken)` to confirm it still equals the stored `tokenHash` — i.e. the file hasn't drifted or been hand-edited since Step 2. Prints `"embeddedTokenPresent": true` on success.

Finish with the two companion audits the runbook calls out — they don't touch this tenant specifically, but they're the standing regression net for the storage guarantees §4.2 depends on (encrypted secret rows, fail-closed posture reads):

npm run audit-secret-key-encryption

npm run audit-governance-posture-reads

The four `governance.*` posture keys — manual follow-up, not automated by this script. The `--apply` artifact lists these four keys under `manualGovernanceKeys` as a reminder, but provisioning does not set any of them — an operator sets them separately (via the tenant configuration store) when a pilot needs a posture other than the fail-closed default. Leaving all four unset is safe for a first pilot run; SuiteCentral falls back to a frozen, conservative default posture rather than an open one.

Key	What it controls	Fail-closed default if unset
`governance.allow_pii`	Whether PII is allowed to reach an AI provider at all for this tenant.	`false`
`governance.block_on_detection`	Whether a PII detection blocks the request outright, instead of just redacting and continuing.	`false`
`governance.auto_redact`	Whether detected PII is automatically masked before use.	`true`

Key	What it controls	Fail-closed default if unset
`governance.pii_types_csv`	Which specific PII types the posture above applies to (comma-separated list).	empty (no types singled out)

4.3 Mint an operator embedded session

The service token from §4.2 authenticates exactly one thing: a call to `POST /api/embedded/host-bootstrap` (`src/routes/embedded/hostBootstrapRouter.ts`). That call is what actually mints the operator session Parts 5–6 use — it's a two-stage process, and the two tokens it produces are not interchangeable:

1. **§4.2's service token** proves *this call is coming from an authorized embedding integration* for the `squire_pilot` tenant + NetSuite account.
2. **The `sessionId` this call returns** is the actual operator credential — it carries the operator's roles and is what the triage UI and mapping editor check on every subsequent request.

Shell

```
curl -sS -X POST https://<host>/api/embedded/host-bootstrap \
  -H "Authorization: Bearer <rawToken from pilot-provision.json>" \
  -H "x-embedded-platform: netsuite" \
  -H "Content-Type: application/json" \
  -d '{
    "platformAccountId": "<NETSUITE_ACCOUNT_ID from Part 2 §2.4>",
    "userId": "squire-operator-1",
    "userRoles": ["ops"],
    "modulePath": "/embedded/sync-error-triage.html",
    "expectedHostOrigin":
"https://<your-account-id>.app.netsuite.com"
  }'
```

A few of these values need to match exactly or the call 400s/401s:

- `Authorization` — the `rawToken` field (not `tokenHash`) from `pilot-provision.json`.
- `x-embedded-platform` — must match the platform the service token was minted for (`netsuite`, here).

- `platformAccountId` (body) — must match the same `--platform-account-id` value used in §4.2.
- `expectedHostOrigin` — must match `^https://[^/]+\.netsuite\.com$` for a real NetSuite account, or `^https://[^/]+\.dynamics\.com$` for Business Central; use your actual NetSuite account's origin here.
- `userRoles` — the operator roles this session carries. Sync Error Assist's triage queue only accepts exactly three role strings — **`ops`**, **`admin`**, **or `finance`** (`OPERATOR_ROLES` in `syncErrorAssistRoutes.ts`) — anything else (including the generic-sounding `"operator"`) gets every triage endpoint a `403 {"ok":false,"code":"forbidden_role"}`. Use `["ops"]` for a standard triage session, or `["ops","admin"]` for a session that also needs approval rights. This is the field that actually grants the roles Sync Error Assist's triage queue checks — it isn't inherited from the service token.

Expected response — `200 OK`:

```
JSON
{
  "sessionId": "es_<22-ish base64url chars>",
  "csrfToken": "csrf_<32-ish base64url chars>",
  "embedSrc":
"/embedded/sync-error-triage.html?embeddedContextId=<urlencoded
sessionId>",
  "sessionExpiresAt": "<ISO 8601 datetime>"
}
```

`sessionId` is the value Part 5 uses two ways: as the `X-Embedded-Session-Id` header on API calls made from the triage UI's own backend calls, and embedded directly in the triage page's URL via `embedSrc`'s `?embeddedContextId=` query parameter. The session is valid until `sessionExpiresAt` (8 hours from minting); re-run this call to mint a fresh one once it expires — the service token from §4.2 stays valid for repeated use.

Error	Cause
`401 missing_or_invalid_authorization`	Bad or missing `Authorization: Bearer` header.
`400 missing_x_embedded_platform`	Missing `x-embedded-platform` header.
`400 missing_platform_account_id`	Missing `platformAccountId` in the body.

Error	Cause
`401 invalid_service_token`	Token/platform/account don't all match what was minted in §4.2 (deliberately collapsed so it doesn't leak which one is wrong).
`403 standalone_host_disabled_in_production`	You used `platform: standalone` with `NODE_ENV=production` — not applicable to the NetSuite path above.
`400 invalid_expected_host_origin`	`expectedHostOrigin` doesn't match the allowed pattern for the platform.

Why you're doing this by hand. In a real embedded deployment, SuiteCentral runs *inside* NetSuite (or Business Central) as a sidecar, and the ERP's own host page calls `host-bootstrap` automatically the moment it loads — an operator never sees this request, let alone types a `curl` command for it. You're doing it manually here because there's no ERP host page in this pilot to do it for you yet; the two-stage curl process above is the standalone/pilot equivalent of that automatic call, using the same endpoint and the same contract the real integration will use later.

4.4 Mint an API JWT

Server-to-server API calls (the kind Part 6 makes when driving the mapping engine directly, rather than through the browser editor) authenticate with a JWT, not an embedded session. There's no endpoint that issues one — you sign it yourself with the server's `JWT_SECRET`, using the exact claim names the server's auth middleware reads (`src/middleware/auth.ts`): subject from `sub` (or `id` as a fallback), tenant from `tenantId` (or `tid`/`tenant_id` as fallbacks).

JavaScript

```
// mint-jwt.js
const jwt = require('jsonwebtoken');

const token = jwt.sign(
  {
    sub: 'squire-admin',
    tenantId: 'squire_pilot',
    roles: ['admin'],
    permissions: ['*']
  },
  process.env.JWT_SECRET,
```

```
  { expiresIn: '24h' }
);

console.log(token);
```

Run it with the **same** `JWT_SECRET` value your server is using (Part 1 §1.2) — signing with a different secret produces a token the server's `verifyJWT` will reject as invalid. Don't type the secret value directly on the command line — it gets recorded in shell history and is briefly visible to other processes on the host via `ps`. It already lives in your `.env` from Part 1, so source it from there instead, and unset it from your shell once you have the token:

set -a; source .env; set +a

export JWT=$(node mint-jwt.js)

unset JWT_SECRET

Note that `source .env` exports **every** value in the file into your shell (the `set -a`/`set +a` pair auto-exports them all), not just `JWT_SECRET` — on a shared host, close the terminal when you're done, or `unset` the other credentials too.

Smoke-test it:

curl -H "Authorization: Bearer $JWT" http://localhost:3003/api/statistics

Expected: HTTP `200` with a JSON body of cross-tenant aggregate statistics. (`/api/statistics` deliberately requires `authMiddleware` rather than the usual per-tenant scoping — it's an admin/ops diagnostic endpoint, not tenant data, so a `200` here just proves your JWT is well-formed and signed with the right secret. Part 6's field-mapping API calls use this same `Authorization: Bearer $JWT` pattern against tenant-scoped endpoints instead.)

4.5 What you're carrying into Parts 5–6

Value	Where it came from	Where it's used next
`squire_pilot`	Chosen in §4.2	Part 5's HMAC key-id, and every tenant-scoped call from here on

Value	Where it came from	Where it's used next
Service token (`pilot-provision.json`'s `rawToken`)	§4.2 `--apply`	Re-run §4.3 with it any time you need a fresh operator session
Operator session ID (`es_...`)	§4.3 response	`X-Embedded-Session-Id` header / `?embeddedContextId=` on Part 5's triage page
API JWT (`$JWT`)	§4.4	`Authorization: Bearer $JWT` on Part 6's direct API calls

With all three in hand, move on to Part 5.

Part 5 — Install and use Sync Error AI Assist

By the end of this part you will have: both NetSuite custom record types Sync Error Assist depends on, created by hand; a webhook HMAC secret generated, stored encrypted on the server, and mirrored into NetSuite as an API Secret; the SuiteScript webhook deployed and firing on real sync errors; a verified round trip from a signed synthetic webhook through to a `202 accepted` response; and a working understanding of the operator triage screen — Accept, Reject, and Escalate — including two write-back defects this guide's own testing surfaced, both since fixed on this same branch (Chapter C3 tells that story and what to expect now).

No SDF (SuiteCloud Development Framework) objects ship this feature. `platform/netsuite-suiteapp/` in this repository contains only a manifest and a host Suitelet — there is no bundle, no custom-record XML, nothing you can install through NetSuite's bundle installer. Both custom record types below must be created by hand, once, in your NetSuite account. That is not an oversight this guide is working around — it is the actual current state of the codebase, and Chapter A1 is the authoritative procedure for creating them.

The primary technical reference for everything in this part is `docs/operations/SYNC-ERROR-ASSIST-WEBHOOK-SUITESCRIPT.md` in this repository — it carries the full SuiteScript source, the secret-bootstrapping commands, and the rotation runbook. This guide summarizes and links to it rather than duplicating the script inline; if something here and that file ever disagree, the runbook file is source of truth for the script itself, and this guide is source of truth for the parts of the walkthrough (NetSuite click-paths, end-to-end verification, operator UI) the runbook doesn't cover.

Chapter A — Install

A1. Create the two NetSuite custom record types (manual — no SDF object exists)

Do this once, in your NetSuite sandbox, as an Administrator. You are creating two separate record types. Do both before moving on to A2.

Record type 1: `customrecord_suitecentral_sync_error` — this is the record SyncCentral already writes to when a sync fails; Sync Error Assist reads it. The field IDs below are **confirmed** — they are the literal `fieldId` strings the shipped SuiteScript reads (`docs/operations/SYNC-ERROR-ASSIST-WEBHOOK-SUITESCRIPT.md` lines 124-129), so create the fields with these exact IDs, not IDs of your choosing.

1. Navigate to **Customization > Lists, Records & Fields > Record Types > New**.
2. **Name**: `SuiteCentral Sync Error` (the label NetSuite shows in menus).
3. **ID**: NetSuite auto-suggests an ID from the name — overwrite it so it reads **exactly** `customrecord_suitecentral_sync_error`. This exact string is what every part of this guide, the SuiteScript, and the server's webhook route all key off of; a different ID here breaks everything downstream silently (no error until the first webhook 400s).
4. Leave **Access Type** at its default (or restrict to specific roles/employees if your NetSuite governance requires it — the access-token role from Part 2 §2.3 needs at least view/create/edit on this record type either way).
5. Click **Save**.
6. On the record type's own page, use the **Fields** subtab (or **Customization > Lists, Records & Fields > Record Fields > New**, selecting this record type) to add each field below:

Field ID (NetSuite)	Label (your choice)	NetSuite field type	Required?	Purpose
(built-in `id`)	*(none — built-in)*	Built-in Integer, read-only	—	Not created by you; every NetSuite record has this. The SuiteScript reads it as `r.id` and sends it as `errorRecordId`.

Field ID (NetSuite)	Label (your choice)	NetSuite field type	Required?	Purpose
(built-in `lastmodified`)	*(none — built-in)*	Built-in Date/Time, read-only	—	Not created by you. Used as the watermark for both the webhook and the 30-minute polling reconcile.
`custrecord_error_type`	Error Type	Free-Form Text (64 char max)	Yes	Error category shown in the AI prompt and the operator queue.
`custrecord_error_message`	Error Message	Free-Form Text — **Text Area** (long text; up to 8192 chars)	Yes	The raw sync-failure message; the source text the AI reads to draft a fix.
`custrecord_source_payload`	Source Payload	Free-Form Text — **Long Text** (up to 32KB)	No	JSON-serialized source-record fields, given to the AI as context. The SuiteScript writes this as a JSON *string* (it calls `JSON.stringify` on the source object before storing it — NetSuite itself doesn't understand JSON, it's just a long text field holding JSON text).

Field ID (NetSuite)	Label (your choice)	NetSuite field type	Required?	Purpose
`custrecord_attempt_count`	Attempt Count	Integer Number	No	Retry counter; defaults to 0 if left blank. Influences confidence/back off messaging.

> **Trap that "silently looks right but 400s":** `errorRecordId` in the webhook payload must be a **string**, not a NetSuite number. NetSuite's own `record.id` is a read-only number internally — the SuiteScript in A3 wraps it as `String(r.id)` before sending. If you ever customize that script, keep the `String(...)` wrap; without it, every real webhook 400s with `invalid_payload` because the server's schema requires a string matching `/^[a-zA-Z0-9_-]{1,128}$/` (`src/routes/syncErrorAssistWebhookSchema.ts`).

Record type 2: `customrecord_suitecentral_fix_suggestion` — this is a *new* record type Sync Error Assist creates one row in per AI-drafted suggestion, so an operator (and NetSuite's own audit trail) can see what the AI proposed. **These field IDs are not read from anywhere in the code today — no SDF object and no other file in this repository names a literal `custrecord_*` ID for this record type.** That's because the write-side code builds a plain JavaScript object with generic keys (`confidence`, `suggestion_text`, and so on) and hands it to the NetSuite connector without ever specifying a NetSuite field ID. **This guide is the source of truth for the field IDs below** — they're a proposed, sensible naming convention (the `custrecord_` prefix plus the JS key), not something extracted from the code. Create the fields with these IDs; nothing will break if you use different ones, but nothing in the code expects any particular ID either, so there's no reason to deviate.

1. **Customization > Lists, Records & Fields > Record Types > New** again.
2. **Name**: `SuiteCentral Fix Suggestion`.
3. **ID**: `customrecord_suitecentral_fix_suggestion` — exact string, same reasoning as above.
4. **Save**, then add these fields:

Proposed field ID (`custrecord_` prefix — guide-authoritative, see note above)	**Label (your choice)**	**NetSuite field type**	**Required?**	**Purpose**
`custrecord_error_record_id`	Error Record ID	Free-Form Text	Yes	Back-reference to the Table 1 record this suggestion is for.
`custrecord_confidence`	Confidence	List/Record or Free-Form Text, values `high` / `mid` / `low`	Yes	The AI's confidence; drives operator triage sort and the `confidence_threshold` setting from A2.
`custrecord_suggestion_type`	Suggestion Type	List/Record or Free-Form Text, values `create_missing_record` / `fix_field_value` / `manual_review`	Yes	What kind of fix the AI is proposing.
`custrecord_suggestion_text`	Suggestion Text	Free-Form Text — Text Area (1–2048 chars)	Yes	Human-readable fix description shown in the operator queue.
`custrecord_references_field`	References Field	Free-Form Text (128 char max)	No	Names the specific source field the suggestion is about, when applicable;

Proposed field ID (`custrecord_` prefix — guide-authoritative, see note above)	Label (your choice)	NetSuite field type	Required?	Purpose
				blank/empty means "not applicable," not "unknown."
`custrecord_reasoning_trace_id`	Reasoning Trace ID	Free-Form Text	Yes	Links back to the explainable-AI reasoning trace for this suggestion (same session ID the AI orchestration layer uses).
`custrecord_provider_used`	Provider Used	Free-Form Text	Yes	Which AI backend produced the suggestion — observed values are `cloud-api`, `local`, `local-llm`, `rule-based` (an internal provider-mode label, not a vendor name like "anthropic").
`custrecord_cost_estimate_usd_cents`	Cost Estimate (cents)	Integer Number	No	Cost of the AI call, in whole US cents.

> **History note — this write path was broken when this guide was first drafted, and is now fixed.** The guide's own testing surfaced a defect where the NetSuite create call for this record type sent an empty payload (none of the eight field values above landed); that defect was fixed on this same branch, and the write now carries all eight values. Chapter C3 has the full story. On your first real suggestion, it's still worth opening the created NetSuite record once to confirm the field values populated in *your* account — ordinary first-run verification, not a workaround.

A2. Generate the HMAC secret and store the tenant configuration

The webhook proves it's really coming from your NetSuite account using an HMAC signature — a shared secret that both sides know, that never travels over the wire itself. Two copies of the same secret need to exist: one on the SuiteCentral server (encrypted), one in NetSuite (as an API Secret, so the SuiteScript can read it without hardcoding it in script source).

Step 1 — generate a secret.

openssl rand -hex 32

Copy the 64-character hex output somewhere safe; you'll paste it in two places below and then never need to see it again.

Step 2 — store it on the server, encrypted. This *requires* the `SECRET_MANAGER_PROVIDER` you configured in Part 1 §1.2 (`aws`, `azure`, or `hashicorp` — never `env`, which can't write secrets at all). Run this from the repo root, on the deploy host, after `npm run build`:

```shell
node --input-type=module -e "
  const { container } = await
import('./dist/inversify/inversify.config.js');
  const { TYPES } = await import('./dist/inversify/types.js');
  const repo = await
container.getAsync(TYPES.TenantConfigurationRepository);
  await repo.upsert('squire_pilot',
'sync_error_assist.webhook_hmac_secret', '<hex from Step 1>', {
isEncrypted: true });
  await repo.upsert('squire_pilot', 'sync_error_assist.enabled',
'true', { isEncrypted: false });
  await repo.upsert('squire_pilot',
'sync_error_assist.webhook_enabled', 'true', { isEncrypted: false
});
"
```

This is lifted directly from `docs/operations/SYNC-ERROR-ASSIST-WEBHOOK-SUITESCRIPT.md` — see that file for why it must be `node --input-type=module -e` (top-level `await` needs ESM context) and why `container.getAsync` (not `.get`) is required (`TenantConfigurationRepository` is async-bound behind an async `DatabaseService`). **Do not substitute direct SQL for the first line.** The encrypted-read path recomputes an expected secret name from `(tenantId, settingKey)` and rejects any row whose stored value doesn't match it exactly (an anti-exfiltration guard) — a hand-inserted SQL row can't produce that value, and a mismatched row silently 401s every webhook as `unknown_tenant` rather than telling you why. Direct SQL is fine for the two plaintext feature-flag rows only, if you need it for an audit-trail reason; the runbook file has that form too.

The three keys, in one place:

Key	Value	What it means
`sync_error_assist.webhook_hmac_secret`	the hex secret, **encrypted**	The shared signing secret.
`sync_error_assist.enabled`	`true`	Master switch for the feature — both this and `webhook_enabled` must be `true`, or an authenticated webhook still gets accepted at the HTTP layer but is a no-op (`200 {"ok":false,"reason":"webhook_disabled"}` rather than `202`).
`sync_error_assist.webhook_enabled`	`true`	Specifically gates the webhook ingest path (as opposed to the 30-minute polling reconcile, which is controlled separately).

`sync_error_assist.confidence_threshold` is a fourth key you'll likely also want, though it's not required for the webhook to function — it controls which suggestions are prominently surfaced in the operator queue by default:

Value	Behavior
`high`	Only high-confidence suggestions surface by default; mid/low are recorded but less visible. Fewer false positives shown to operators, more suggestions sit quietly.
`mid`	High- and mid-confidence suggestions both surface by default. A reasonable starting point for a pilot — enough signal to be useful without burying operators in low-confidence noise.
`low`	Everything surfaces, including low-confidence suggestions. Useful early in a pilot when you're still calibrating what "good" looks like (see the beta disclosure at the top of this guide), noisier day to day.

```
node --input-type=module -e "

  const { container } = await import('./dist/inversify/inversify.config.js');

  const { TYPES } = await import('./dist/inversify/types.js');

  const repo = await container.getAsync(TYPES.TenantConfigurationRepository);

  await repo.upsert('squire_pilot', 'sync_error_assist.confidence_threshold', 'mid', { isEncrypted: false });

"
```

Step 3 — store the same secret in NetSuite as an API Secret, so the SuiteScript in A3 can read it without ever hardcoding it in script source:

Setup > Company > API Secrets > New

- **Script ID**: `custsecret_sync_error_hmac_<your tenant id>` (e.g. `custsecret_sync_error_hmac_squire_pilot`).
- **Value**: paste the same hex string from Step 1.
- Save.

You now have the same secret in two places, matched by tenant. If you ever rotate it, update both — the rotation runbook (in `docs/operations/SYNC-ERROR-ASSIST-WEBHOOK-SUITESCRIPT.md`, also summarized in the beta disclosure at the top of this guide) covers the brief delivery gap that opens between updating one side and the other.

A3. Deploy the SuiteScript

The full script — a User Event script, ~90 lines, triggered `afterSubmit` on `customrecord_suitecentral_sync_error` — lives in `docs/operations/SYNC-ERROR-ASSIST-WEBHOOK-SUITESCRIPT.md` under "SuiteScript code." This guide doesn't reproduce it here; open that file and copy the script verbatim. Save it locally as `customscript_suitecentral_sync_error_assist_ue.js` before uploading.

1. **File Cabinet.** Upload the `.js` file somewhere under **SuiteScripts** in NetSuite's File Cabinet (Documents > Files > File Cabinet, or upload it directly from the script-record screen in the next step).
2. **Customization > Scripting > Scripts > New.**
3. **Script File**: select the file you just uploaded.
4. **Script Type**: NetSuite reads the `@NScriptType UserEventScript` annotation at the top of the file and should default to **User Event Script** — confirm it did.
5. Click **Create Script Record**. On the resulting script record:
    - Under **Deployments**, click **New Deployment** (or NetSuite may prompt you to create one automatically).
    - **Applies To**: `customrecord_suitecentral_sync_error` (your Record Type 1 from A1).
    - **Event Type**: leave the default trigger set — the script itself checks `context.type` for CREATE/EDIT internally (see the `afterSubmit` function), so it's safe to leave NetSuite's own trigger checkboxes at their defaults.
    - **Status**: `Released` (not `Testing` — a `Testing` deployment only fires for the developer account, not for real sync-error creates).
6. **Set the three script parameters** (under the deployment's **Parameters** subtab — the script declares these via `runtime.getCurrentScript().getParameter(...)`):

Parameter ID	Value
`custscript_sync_error_endpoint`	`https://<your host>/api/sync-error-assist/ingest`
`custscript_sync_error_hmac_secret`	`custsecret_sync_error_hmac_squire_pilot` — the API Secret **script ID** from A2 Step 3, not the raw hex value. The script

Parameter ID	Value
	reads the secret's contents at runtime via `N/crypto`; it never sees the hex string in script source.
`custscript_sync_error_tenant_id`	`squire_pilot`

7. **Save.** The script now fires automatically every time a `customrecord_suitecentral_sync_error` record is created or edited — no further manual trigger is needed. Move on to Chapter B to verify it end-to-end.

The script is deliberately fire-and-forget from NetSuite's side: it POSTs once, logs a 4xx as an assumed misconfiguration, logs a 5xx or network failure and relies on the 30-minute polling reconcile to catch it — it does **not** retry inside the script (NetSuite governance tracks CPU wall-time per script execution; busy-waiting or retrying inside `afterSubmit` would risk hitting that limit on an unrelated record save).

Chapter B — Verify end-to-end

Two independent paths prove Sync Error Assist is wired correctly: a synthetic webhook you send by hand (fast, no NetSuite record needed, proves the server side), and a real NetSuite record (slower, proves NetSuite's side too). Do the synthetic one first.

B1. Synthetic webhook

This is the exact request the SuiteScript in A3 sends — same headers, same signing scheme, same payload shape (`src/routes/syncErrorAssistWebhookSchema.ts`'s Zod schema is the canonical contract). Save as `send-sync-error-webhook.js` and run with `node send-sync-error-webhook.js`:

JavaScript

```
// send-sync-error-webhook.js
const crypto = require('crypto');
const http = require('http'); // use require('https') + adjust
below for a real https:// host
const SECRET = '<hex secret from A2 Step 1>';
const TENANT_ID = 'squire_pilot';

const payload = {
  tenantId: TENANT_ID,
```

```
  errorRecordId: '999001',                // NetSuite internal ID,
as a STRING
  lastModified: new Date().toISOString(), // ISO 8601
  errorType: 'field_mapping_error',
  errorMessage: 'Customer email failed validation on sync to
NetSuite',
  sourcePayload: { customerId: 'CUST-001', email: 'not-an-email' },
// optional
  attemptCount: 1,                         // optional
};

const rawBody = JSON.stringify(payload);
const timestamp = Math.floor(Date.now() / 1000);
const signedString = `${timestamp}.${rawBody}`;
const signature = crypto.createHmac('sha256',
SECRET).update(signedString).digest('hex');

const req = http.request({
  hostname: 'localhost',
  port: 3999,
  path: '/api/sync-error-assist/ingest',
  method: 'POST',
  headers: {
    'Content-Type': 'application/json',
    'Content-Length': Buffer.byteLength(rawBody),
    'x-suitecentral-key-id': TENANT_ID,
    'x-suitecentral-timestamp': String(timestamp),
    'x-suitecentral-signature': signature,
  },
}, (res) => {
  let body = '';
  res.on('data', (c) => (body += c));
  res.on('end', () => console.log('STATUS', res.statusCode, 'BODY',
body));
});

req.write(rawBody);
req.end();
```

Adjust `hostname`/`port` (or swap `http` for `https`) to point at your actual deployment — the values above target a local dev server on port 3999, which is what this guide's own testing used (see the verification note at the end of this chapter).

Expected response: `202` with:

```
JSON
{  "status": "accepted",  "claimId": "<uuid>"}
```

Error responses, all body-shape `{"ok": false, ...}` unless noted:

Status	Code / body	Cause
`415`	`unsupported_media_type`	`Content-Type` header isn't `application/json`.
`401`	*(no code field)*	Missing one of the three `x-suitecentral-*` headers.
`401`	*(no code field)*	`x-suitecentral-key-id` is the reserved system sentinel — not a real tenant ID you'd ever legitimately send.
`401`	*(no code field)*	Malformed timestamp (non-digit), timestamp more than 5 minutes off from server time (replay-window guard, works in both directions), or a malformed signature (not 64 lowercase hex chars).
`401`	*(no code field)*	Signature computed correctly but doesn't match — either the secret is wrong, or the tenant has no secret configured at all (both collapse to the same response deliberately, so a

Status	Code / body	Cause
		prober can't distinguish "wrong secret" from "unknown tenant").
`400`	`invalid_body`	Body isn't valid JSON.
`400`	`invalid_payload`	Valid JSON, but fails the Zod schema (wrong types, `errorRecordId` sent as a number instead of a string, `sourcePayload` over the 32KB/6-deep caps, etc.).
`400`	`tenant_mismatch`	The body's `tenantId` field doesn't match the `x-suitecentral-key-id` header.
`429`	`rate_limited`	Per-IP limiter tripped (30 requests/minute per source IP, pre-authentication).
`429`	`tenant_rate_limited`	Per-tenant limiter tripped (100 requests/minute per `x-suitecentral-key-id`, post-authentication).
`200`	`{"ok": false, "reason": "webhook_disabled"}`	HMAC verified fine, but `sync_error_assist.enabled` or `.webhook_enabled` is not `true` for this tenant — a deliberate soft no-op, not an error.
`500`	`internal_error`	Server-side fault (DI resolution failure, etc.) — not a contract violation; check server logs.

This guide's own testing verified the snippet above is byte-correct against a locally booted dev server (SQLite-backed, `SECRET_MANAGER_PROVIDER=env` with a manually-seeded encrypted-row configuration for the pilot tenant, since the `env` provider can't perform the real

KMS round-trip Step A2 describes — production always uses `aws`/`azure`/`hashicorp`). The request above returned `202 {"status":"accepted","claimId":"..."}` on the first real run; `415` and `401` (missing headers) were also reproduced directly. See Chapter C3 for the two write-back defects that same testing surfaced *downstream* of acceptance — both since fixed on this branch — and what the post-acceptance path should look like now.

B2. Real path — a genuine NetSuite error record

Once B1 passes, prove the whole chain with a real record:

1. In NetSuite, create (or wait for SyncCentral to create) a record of type `customrecord_suitecentral_sync_error` — filling in `custrecord_error_type` and `custrecord_error_message` at minimum.
2. Saving the record fires the `afterSubmit` script from A3, which POSTs to your server the same way B1's synthetic script did.
3. Within a few seconds, open the operator triage page (Chapter C) and look for a new suggestion card referencing that record's ID.

If nothing appears: check the script deployment's execution log (**Customization > Scripting > Script Deployments**, open your deployment, **Execution Log** subtab) for the `log.error`/`log.audit` lines the script emits on failure — a 4xx there means misconfiguration (wrong endpoint, wrong tenant ID, wrong secret script ID), a 5xx or network failure means the server-side request didn't complete, and either way the 30-minute polling reconcile (next paragraph) is your backstop, not something you need to manually retry.

B3. The polling backstop

Sync Error Assist doesn't depend on the webhook firing reliably. A scheduled polling job re-reads `customrecord_suitecentral_sync_error` on a 30-minute cadence and catches anything the webhook missed — a dropped 5xx, a network blip, a secret rotated mid-flight (see the beta disclosure's rotation-gap note). This means a webhook failure is a *latency* problem (up to ~30 extra minutes before a suggestion appears), not a *data-loss* problem. You do not need to build any additional retry logic around B1/B2 above.

Chapter C — Daily operator use

Before anything else: `public/sync-error-assist.html` is a static demo page. It is not this tool. If you navigate to it directly, you'll see a page that looks similar but is disconnected from the real pending-suggestions queue, the real Accept/Reject/Escalate actions, and the real audit trail. The actual operator tool is the embedded page **`public/embedded/sync-error-triage.html`**, and it only works when loaded with a valid embedded session — the same kind of session Part 4 §4.3 walked you through minting.

Open it at:

https://<host>/embedded/sync-error-triage.html?embeddedContextId=<operator session ID from Part 4 §4.3>

In a real embedded deployment this URL is constructed and opened for the operator automatically by the ERP's own host page (see Part 4 §4.3's "why you're doing this by hand" note) — for this pilot, you're pasting the URL in yourself using the session ID your `host-bootstrap` call returned.

C1. The triage screen

Each pending suggestion renders as a card:

- A **confidence badge** (`high` / `mid` / `low`, color-coded — the same three values as the `confidence_threshold` setting from A2) next to the error record ID.
- The AI's **suggestion text** in plain language.
- If applicable, the specific source **field** the suggestion references.
- An expandable **"Suggestion metadata"** panel with the suggestion type, which AI provider produced it, its reasoning-trace ID, and the AI cost in dollars.
- Three buttons: **Accept…**, **Reject…**, **Escalate…** — each expands an inline form; only one form is open per card at a time.

Accept. Choose **create** or **update**, fill in the entity type (e.g. `customer`, `invoice`) and, for update, the record ID, then provide the payload/patch as JSON in the text box, and click **Approve & write back**. This calls `POST /api/sync-error-assist/suggestions/<errorRecordId>/accept`, which runs the write through `guardedWrite` — the same governance-and-DLP re-check every AI-approved NetSuite write in this system goes through, described in the Glossary at the top of this guide. A few outcomes you'll see reflected as a banner rather than a generic error:

Outcome	What happened
Success	The write landed in NetSuite; the card disappears from the queue.
`409 already_dispositioned`	Another operator (or a stale tab) already actioned this suggestion — the queue refreshes automatically.
`503 connector_unavailable`	The target NetSuite connector isn't available — **nothing was written**. Two causes: the five `NETSUITE_*` env vars from Part 2 §2.5 are missing from the container (the connector can't initialize; the server log names exactly

Outcome	What happened
	which keys are missing — see C3), or NetSuite is genuinely unreachable right now. Fix the env vars, or retry once connectivity is back. Distinct from either `502` cause below.
`502 write_failed`	Two distinct causes share this one response today, and the endpoint does not distinguish them: **(a)** the connector reached NetSuite and NetSuite rejected the write — review the payload before retrying; or **(b)** governance policy for the target entity routed the write to the human-in-the-loop **approval queue** instead of writing immediately (see `docs/review/proof-cards/guarded-write-ownership-enforcement.md`) — the queue path currently surfaces as this same `write_failed` error even though your accept was recorded and NetSuite's write is waiting on a second, independent sign-off. To tell them apart: if a `502` appears and a new approval row shows up under `GET /api/governance/approvals`, the write was queued, not failed.

Reject. Requires a reason (the form won't submit without one) — calls `POST .../reject` with `{reason}`. No NetSuite write happens; the disposition is recorded and audited.

Escalate. Requires a note, same shape as Reject but calls `.../escalate`. Use this when a suggestion needs a second opinion or falls outside your own authority to decide.

C2. Cost and audit

Every AI call behind a suggestion is cost-tracked (the `cost_estimate_usd_cents` field you saw in the metadata panel and in the Table 2 field list back in A1 is not decorative — it's read from the same `CostTrackingService` this codebase uses everywhere else AI is called). Every operator disposition — Accept, Reject, or Escalate — writes an audit log row, independent of whether the underlying NetSuite write itself succeeded. Between the two, you get a full accounting of both what the AI cost and what a human decided, even on the rows where the eventual write to NetSuite failed or was queued for further approval.

C3. Two write-back defects this guide's own testing surfaced — both fixed on this branch

Testing Chapter B's synthetic webhook against a real dev server surfaced two real defects in the fix-suggestion write-back path. An earlier draft of this chapter disclosed both as open known gaps; **both have since been fixed on this same branch.** The history is kept here because it explains one behavior you can still encounter (`connector_unavailable`) and what it means now.

1. The AI-drafted fix-suggestion record used to write to NetSuite with an empty payload — FIXED. `NetSuiteConnector.formatDataForNetSuite()`, the method every `create()`/`update()` call routes through, only reads `data.fields` — but `SyncErrorAssistService.ts` handed the connector a flat object with no `.fields` wrapper, so the wire payload NetSuite actually received was `{}` (verified at the time by executing the real `mapCommonFields` helper against the real payload shape — an empty object, zero keys, confirmed by execution, not just by reading the code). The fix wraps the create payload as `{fields: ...}`, matching the *operator-accepted* write path in C1 — the `guardedWrite` call inside `SyncErrorAssistOperatorService.accept()` — which always wrapped its payload correctly. **What to expect now:** a created `customrecord_suitecentral_fix_suggestion` record carries all eight field values from A1's Table 2. On your first real suggestion, open the record once and confirm the fields populated — that's ordinary first-run verification now, not a workaround for a known bug.

2. The NetSuite connector Sync Error Assist resolves was never initialized with credentials — FIXED. Both the fix-suggestion write and the operator's accept-and-write-back resolve their connector via `ConnectorManager.getConnector('netsuite', 'netsuite_<tenantId>')`, and nothing ever called `.initialize()` on that cached instance — reproducing the write threw `TokenError: Missing required OAuth1 credentials` before any network call was made. The fix (`src/services/syncErrorAssist/netsuiteEnvAuth.ts`) builds an OAuth1 config from the same five `NETSUITE_*` env vars Part 2 §2.5 put in your `.env`, and all three Sync Error Assist call sites now initialize the connector with it before use. **This makes Part 2 §2.5's env vars load-bearing for the write-back path** — they are no longer just inputs to the connection-test scripts; they're the credentials every approved write authenticates with. **What to expect now:** with the five vars set inside the container, the write-back path authenticates with the same credentials Part 2 already verified. If they're missing from the container, behavior degrades to the same observable outcomes as before the fix — the accept endpoint returns `503 connector_unavailable` (C1's table) — but the server log now names exactly which keys are missing (`NetSuiteEnvCredentialsMissingError`) instead of the opaque `TokenError`. The fix for that state is configuration, not code: set the five vars in `.env` and recreate the container (`up -d`, not `restart` — Part 2 §2.5).

One more behavior rides along with this fix, added after review flagged the multi-tenant implication of deployment-wide credentials: at initialize time, Sync Error Assist now compares the tenant's *provisioned* NetSuite account — the `--platform-account-id` you passed in Part 4 §4.2 — against the deployment's `NETSUITE_ACCOUNT_ID`, and **fails closed on mismatch**. The comparison tolerates the underscore/hyphen and case spelling differences from §2.5's DNS note (`1234567_SB1` and `1234567-sb1` are the same account). A tenant with **no provisioning record at all** fails closed too (`NetSuiteTenantUnprovisionedError` — a second review round caught that letting unprovisioned tenants through would recreate the hole for exactly the tenants that skipped Part 4): enabling `sync_error_assist` for a tenant now requires the Part 4 §4.2 provisioning step to have run first, which is the order this guide already walks you through. For the single-account pilot the account IDs always match and you will never see either error fire. The guard exists so a second tenant provisioned for a *different* NetSuite account — or never provisioned at all — can't silently read from or write fix-suggestions into the first tenant's ERP through the shared env credentials: per-tenant ERP credentials don't exist yet (the five env vars are the only credential store), so until they do, one deployment serves one NetSuite account, and the guard enforces that instead of trusting you to remember it. When it fires, the outcome is the same `503 connector_unavailable` you already know, with the typed error in the server log naming the tenant and the account IDs involved.

Part 6 — Install and use AI Field Mapping

By the end of this part you will have: generated a batch of AI field-mapping suggestions in the primary editor UI and via the raw API, learned to read the one response field (`isDemo`/`isLiveAI`) that tells you whether a suggestion actually came from an LLM or from the built-in rule-based fallback, saved a reviewed mapping as a server-side template — from the editor's own Save Template button or via the direct API automation route, both of which write to the same store — and read the fixture-benchmark accuracy numbers with the caveat that applies to them every time they're quoted. (An earlier draft of this part disclosed several editor defects — a broken Save Template button, Apply/Reject never reaching the server, a split load/save template system — all fixed on this same branch; §7.3 records the full before/after.)

6.1 How it decides: live AI vs. rule-based fallback

Every mapping-suggestion request — whether triggered by clicking **Generate AI Suggestions** in the editor or called directly via API — lands on the same endpoint, `POST /api/ai/proxy/mapping/suggestions`. If a provider key (`ANTHROPIC_API_KEY` and/or `OPENAI_API_KEY` from Part 1 §1.2) is configured and reachable, that call returns real LLM-generated suggestions with `"providerId": "openai"` (or `"claude"`, `"lmstudio"`, …), `"isDemo": false`, and `"isLiveAI": true`. If no key is configured — or the configured provider fails — the exact same endpoint returns `"providerId": "rule-based"`, `"isDemo": true`, and `"isLiveAI": false`, with suggestions generated by exact/partial field-name matching instead of an LLM. (`isDemo`/`isLiveAI` are the badge fields to check; a `fallbackReason` string also appears, but only when your request explicitly asked for a

provider via `preferredProvider` — don't rely on it in the general case.) **Both cases return `success: true` and a normal-looking `suggestions` array — nothing about the HTTP status or the shape tells you which one happened.** Checking `isDemo`/`isLiveAI` (API callers) or the yellow "🧪 Demo Mode" badge next to the provider pill in the editor's AI Assistant panel (UI users) is the only way to tell, and skipping that check is the single most common way to misread this feature — mistaking a confident-looking rule-based suggestion for an AI one.

Provider selection, when a live key is available, follows this order: an explicit `preferredProvider` in your request, then your per-user setting for the `field_mapping` task in the AI Configuration dashboard (§6.2 below), then the account's default provider. This guide's own local testing (§6.4) reproduced the no-key case directly: with neither key set, `POST /api/ai/proxy/mapping/suggestions` returned `providerId: "rule-based"`, `isDemo: true`, `isLiveAI: false` on every call, confirming the fallback is silent and automatic, not an error.

6.2 Setup

1. **Confirm at least one provider key is set** — `ANTHROPIC_API_KEY` and/or `OPENAI_API_KEY`, from the feature-blocking tier in Part 1 §1.2. Without one, §6.1's fallback applies to every request from this server: still usable, but never AI-generated.
2. **Open the AI Configuration dashboard** at `https://<host>/ai-configuration-dashboard.html`. Under the Tasks tab, find the `field_mapping` task entry and pick a provider/model for it explicitly, or leave it unset to fall through to whichever provider is marked "Default Route" in the Providers tab — both are legitimate choices, but knowing which one you picked matters when you're troubleshooting an unexpected `providerId` later.
3. **Cost guardrails — read this as informational, not configurable.** `AI_COST_ALERT_THRESHOLD` and `AI_COST_HARD_LIMIT` appear in `.env.example` but are dead: no code in this repository reads them (Part 1 §1.2 already flagged this). The actual per-session cap is hardcoded in `CostTrackingService.ts` at **$0.30 (alert)** / **$0.40 (hard limit)**. There is nothing to set here today.
4. **Auth truth — send your JWT anyway.** `AI_PROXY_AUTH_REQUIRED` (also in `.env.example`) is read by zero code in this repository — no environment variable enforces authentication on `/api/ai/proxy/*`. An unauthenticated call is accepted and runs as the server's internal `SYSTEM_IDENTITY`, not rejected. This guide's instruction is: **always send `Authorization: Bearer $JWT`** (the token from Part 4 §4.4) on every call in this part regardless, because it's what makes the request carry your tenant's identity and governance posture (cost attribution, DLP posture, audit trail) instead of the generic system identity — not because anything will reject you if you omit it.

6.3 Editor walkthrough

The primary production UI is `https://<host>/ai-field-mapping-editor.html`. (`AI-Integrated-Mapping-Studio.html` still exists but is legacy and redirects here — if

you land on it, that's expected, not a bug; just note you were redirected and continue on this page.)

1. Pick source and target systems. The two dropdowns at the top of the page (**Source System**, **Target System**) are populated from `GET /api/configurations` — the connector configs you created in Parts 2/3 — plus a small built-in demo system list (Salesforce, NetSuite, Shopify, HubSpot, QuickBooks). You can still create a config here for organizational purposes (it's what labels a saved template with real system names):

```shell
curl -sS -X POST https://<host>/api/configurations \
  -H "Authorization: Bearer $JWT" \
  -H "Content-Type: application/json" \
  -d '{
    "id": "sf_to_ns_customers",
    "name": "Salesforce to NetSuite Customer Sync",
    "sourceSystem": "salesforce",
    "targetSystem": "netsuite",
    "sourceEntity": "customers",
    "targetEntity": "customer",
    "syncDirection": "source_to_target",
    "syncMode": "batch",
    "isActive": false,
    "sourceAuthentication": {
      "type": "api_key",
      "credentials": { "apiKey": "placeholder" }
    }
  }'
```

Validation is two layers, and both matter. The route itself only prechecks `name`, `sourceSystem`, and `targetSystem` — omit any of those and you get a 400 listing exactly which are missing. But every request that clears that precheck is then validated against the full Zod config schema (`IntegrationConfigSchema` in `src/schemas/configurationSchemas.ts`), which also requires `sourceEntity`, `targetEntity`, `syncDirection`, `syncMode`, `isActive`, and `sourceAuthentication` (this guide live-tested a request with only the three precheck fields and confirmed it 400s at this second layer: `"sourceEntity: Invalid input: expected string, received undefined"` — the example above is what actually saves, live-verified 201). `id` is optional (auto-generated if you omit it) and `tenantId` is bound automatically from your JWT, not read from the body. One more trap worth knowing before you experiment: flip `isActive` to `true` on

a config with no `fieldMappings` entries and you'll hit a schema refine — `"fieldMappings: Active configurations must have at least one field mapping"` — so keep `isActive: false` until you're ready to add real mappings. Once saved, reload the editor and the new system appears in both dropdowns.

> **Know this before you proceed — the fields you see are generic catalogs, not YOUR schema.** Picking NetSuite, Salesforce, or SuiteCentral in either dropdown now fetches a per-system field catalog from the server (`GET /api/ai/proxy/mapping/schemas/:systemType`), so different systems show different, system-appropriate field lists. (Earlier builds hardcoded one identical mock list for every system regardless of your pick — fixed on this branch.) But be honest with yourself about what these catalogs are: **fixture-based generic ERP schemas bundled with the server, NOT a live fetch of your own tenant's fields.** No connector is consulted — a custom field you added in your NetSuite account will not appear here. This gap narrowed on this branch but is not closed; it's the one still-open editor gap in §7.3.
>
> **To get YOUR real fields into the tool**, use the **Upload data file** control next to each panel (accepts `.csv`/`.json`/`.txt`) and upload a sample of your real source/target records — that remains the one path that reflects your actual schema. Loading a previously-saved template (step 4 below) also brings in the real field names it was saved with. For a real pilot mapping, upload your actual fields first; treat the catalog fields as a realistically-shaped smoke test of the Generate flow, not a preview of your real data.

2. Load your fields (upload your data file, per the callout above, or work from the per-system catalog fields for a first smoke test), then click **Generate AI Suggestions** in the AI Assistant panel (right-hand side; click the brain icon if it's collapsed). Each suggestion card shows a **confidence percentage** (color-coded red/orange/yellow/green), a plain-language **reason**, the **transformation type**, and — once at least one suggestion has come back — a **provider badge** in the panel header (§6.1's live-vs-demo signal).

3. Review each suggestion, then **Apply** (accept) or **Reject** it individually, or use **Apply All** with a confidence-threshold slider for batch acceptance. Beyond updating your working `currentFieldMappings` array in the page's own state, these buttons now report your decision server-side (a fix on this branch — earlier builds kept Apply/Reject purely in-browser): **Apply** fires the `POST /api/ai/proxy/suggestions/:id/accept` telemetry endpoint (§6.4) plus `POST /api/ai/proxy/mapping/feedback` with an accepted decision (the endpoint that feeds the rule-based engine's learned confidence weights), **Reject** fires feedback with a rejected decision, and **Apply All** covers every suggestion it accepts. All of these are fire-and-forget — the UI never blocks or errors if a server call fails — so treat them as the telemetry/learning signal they are, not as persistence: saving the reviewed mapping itself is the Save Template step below or §6.4's API call.

> **Save Template now performs a real server-side save.** Clicking **Save Template** with demo mode off POSTs your reviewed mappings to the real create endpoint (`POST /api/templates/`) with the `fields` key the server's contract requires, and on success shows the returned template key — the slug you'd use with `GET /api/templates/:key`. (This button was broken when this guide was first drafted — it POSTed a nonexistent route with a wrong body key and always failed with an unhelpful toast; fixed on this same branch.) **Demo mode remains a separate, deliberate path**: with the Demo toggle (top-right pill) on, Save Template writes to the browser's `localStorage` instead — fine for exploring the editor, but browser-local only, gone if you clear site data, and never reaches the server. §6.4's direct API call persists to the same server-side store this button writes to — use it when you're scripting rather than clicking.

4. Load a saved template via **Load from Template** — this now reads the same unified template system Save Template writes to (`GET /api/templates/`), so a template you saved with the editor's button or via §6.4's `POST /api/templates/` call shows up in this list. (Another fix on this branch — earlier builds read a different, older template router, and live-saved templates never appeared in the Load list.) The list additively merges the eight built-in templates from the older library, deduplicated by key with the unified entry winning on collision — so the built-in browsing experience is unchanged, just with your own saved templates alongside.

6.4 API automation path

This is the automation path for scripting AI Field Mapping end-to-end — generate, record decisions, persist — with no browser involved. It writes to the same unified template store the editor's Save Template button uses (§6.3), so anything you persist here also appears in the editor's Load-from-Template list.

Generate suggestions. This is the exact request `MappingRouter.ts`'s Zod schema expects (`sourceSystem`, `targetSystem`, and at least one field in each of `sourceFields`/`targetFields`, where every field must carry both a `name` **and** a `type` — the schema requires both, as the example below shows); this guide ran it against a locally booted dev server with no provider key configured to confirm the shape below byte-for-byte:

```shell
curl -sS -X POST https://<host>/api/ai/proxy/mapping/suggestions \
  -H "Authorization: Bearer $JWT" \
  -H "Content-Type: application/json" \
```

```
  -d '{
    "sourceSystem": "salesforce",
    "targetSystem": "netsuite",
    "sourceFields": [
      { "name": "Email", "type": "string" },
      { "name": "Phone", "type": "string" },
      { "name": "BillingCity", "type": "string" }
    ],
    "targetFields": [
      { "name": "email", "type": "string" },
      { "name": "phone", "type": "string" },
      { "name": "city", "type": "string" }
    ]
  }'
```

Abridged response (this guide's own no-key test run — with a real provider key configured, `providerId`/`isDemo`/`isLiveAI` flip as described in §6.1 and `reason` text comes from the LLM instead of a string-match description):

JSON


```
{
  "success": true,
  "suggestions": [
    {
      "id": "suggestion_1783626055002_33xk96ksu_0",
      "sourceField": "Email",
      "targetField": "email",
      "confidence": 0.646,
      "reason": "Exact field name match: \"Email\" = \"email\";
Provider: rule-based",
      "transformationType": "direct"
    }
  ],
  "providerId": "rule-based",
  "providerName": "📋 Rule-Based Engine",
  "isDemo": true,
  "isLiveAI": false,
```

```
  "cost": {
    "estimatedCost": 0.0081,
    "tokensUsed": 180
  },
  "governance": {
    "approved": true,
    "flags": [],
    "riskLevel": "low"
  },
  "metadata": {
    "suggestionsCount": 3,
    "avgConfidence": 0.6205,
    "timestamp": "2026-07-09T19:40:55.017Z"
  }
}
```

Record an accept decision — telemetry only, not persistence. `POST /api/ai/proxy/suggestions/:suggestionId/accept` records that a suggestion was accepted (for the telemetry/cost-accounting stream) and returns `{"success": true, "message": "Suggestion accepted", ...}`. **It does not save, write, or persist the mapping anywhere** — it's an event log entry, nothing more. This guide ran it directly against a local server and confirmed the response is exactly that one-line acknowledgement, with no mapping data stored as a side effect:

Shell

```
curl -sS -X POST
"https://<host>/api/ai/proxy/suggestions/suggestion_1783626055002_3
3xk96ksu_0/accept" \
  -H "Authorization: Bearer $JWT" \
  -H "Content-Type: application/json" \
  -d '{}'
```

-H "Authorization: Bearer $JWT" \

-H "Content-Type: application/json" -d '{}'

Persist the mapping. This is `POST /api/templates/` (mounted at `/api/templates`; a plain POST to the mount root works with or without a trailing slash) — the same endpoint the editor's Save Template button now calls — reading `UnifiedTemplateService.createTemplate` directly for its contract: `name` (required, non-empty) and `fields` (required, non-empty array — note the key is `fields`, not `fieldMappings`) are the only required fields; `description`, `sourceSystem`, `targetSystem`, `category`, `tags`, `configuration`, and `metadata` are all optional. Each entry in `fields` is `{source, target, transformation, params?, required?, validation?, description?}`. This guide ran both variants below against a local server to confirm the contract byte-for-byte:

Wrong body key — the server requires `fields`, not `fieldMappings` — confirmed 400:

Shell

```
curl -sS -X POST https://<host>/api/templates/ \
  -H "Authorization: Bearer $JWT" \
  -H "Content-Type: application/json" \
  -d '{
    "name": "test",
    "fieldMappings": [
      { "source": "a", "target": "b" }
    ]
  }'
```

-H "Authorization: Bearer $JWT" -H "Content-Type: application/json" \

-d '{"name":"test","fieldMappings":[{"source":"a","target":"b"}]}'

-> {"error":"Template must have a name and at least one field mapping"}

The real body shape — this is the one to actually use:

Shell

```
curl -sS -X POST https://<host>/api/templates/ \
  -H "Authorization: Bearer $JWT" \
  -H "Content-Type: application/json" \
  -d '{
```

```
    "name": "Salesforce to NetSuite Customer Pilot",
    "description": "Core contact fields, generated by AI Field
Mapping and reviewed by an operator",
    "sourceSystem": "salesforce",
    "targetSystem": "netsuite",
    "fields": [
      { "source": "Email", "target": "email", "transformation":
"direct" },
      { "source": "Phone", "target": "phone", "transformation":
"direct" },
      { "source": "BillingCity", "target": "city",
"transformation": "direct" }
    ]
  }'
```

Shell

```
  -H "Authorization: Bearer $JWT" \
  -H "Content-Type: application/json" \
  -d '{
    "name": "Salesforce to NetSuite Customer Pilot",
    "description": "Core contact fields, generated by AI Field
Mapping and reviewed by an operator",
    "sourceSystem": "salesforce",
    "targetSystem": "netsuite",
    "fields": [
      { "source": "Email", "target": "email", "transformation":
"direct" },
      { "source": "Phone", "target": "phone", "transformation":
"direct" },
      { "source": "BillingCity", "target": "city",
"transformation": "direct" }
    ]
  }'
```

The second call returns `201` with the created template, including a generated `key` (`salesforce-to-netsuite-customer-pilot` — slugified from `name`) you'll use to fetch it back later via `GET /api/templates/:key`. One more thing worth knowing before you rely on this for a multi-tenant pilot: **`/api/templates` has no `authMiddleware` and templates are**

not tenant-scoped — they're stored process-wide in a JSON file (`config/custom-templates.json`) on the server, visible to — and writable by — any caller who can reach the endpoint, unlike `/api/configurations` (Part 4-scoped, tenant-isolated) and the AI proxy's governance/audit trail. Still send `Authorization: Bearer $JWT` for consistency and in case that changes, but don't treat a saved template as tenant-private today.

6.5 Accuracy — read this caveat every time these numbers appear

Metric	Value
Top-1 accuracy (OpenAI `gpt-5.4-mini`)	**95.1%**, Wilson 95% CI [86.5%, 98.3%]
Top-1 accuracy (Anthropic `claude-haiku-4-5`)	**96.7%**, Wilson 95% CI [88.8%, 99.1%]
Hallucinations	**0**, both providers
Fixture	Salesforce Account → NetSuite Customer, 22 test cases with 61 hand-labeled mappings total
Secondary fixture	Salesforce → Business Central Customer, 11 mappings — **100%** top-1 on both providers
Measured cost	**≈$0.04** for the full 61-mapping NetSuite run (OpenAI); **≈$0.07** (Anthropic)

These are **fixture-measured numbers from a small, hand-labeled, curated benchmark** (`docs/review/ai-accuracy-benchmark.md`, reproducible via `npm run benchmark:ai`) — not a population-level accuracy guarantee. The Wilson confidence intervals quantify sampling noise at this fixture's size (deliberately wide, by construction, for an n=61 set); they say nothing about how representative the fixture is of your actual field data. **Your own accept-rate on your real field mappings — visible over time via the accept/reject decisions you record through §6.4's endpoints — is the number that actually matters for your pilot**, and the `POST /api/ai/proxy/mapping/feedback` endpoint mentioned in §6.3 is how those decisions feed back into improving the rule-based engine's confidence weighting for future suggestions.

Part 7 — Master verification checklist, troubleshooting, known gaps

You've now walked every part of this guide once. This part is the closer: one table to confirm the whole chain end-to-end, one troubleshooting table that consolidates every symptom scattered

across Parts 1–6, and one honest ledger of the defects this guide surfaced (most since fixed on this same branch) and what remains open or unmeasured — so nothing here surprises you mid-pilot.

7.1 Master verification checklist

Run these in order. Each row assumes every row above it already passed. If you skipped Part 3 (Business Central), skip that row too — nothing later in the table depends on it.

#	Check	Command	Expected result
1	Server healthy	`curl -fsS http://localhost:3003/health/ready`	`200` with `{"status":"ready",...}` (Part 1 §1.3)
2	Boot guards clean (no abort messages in the logs)	`docker compose -f docker-compose.prod.yml logs integration-hub | grep -iE "FATAL|must be true|placeholder|insecure"`	No output — a hit here means one of Part 1 §1.4's four boot-abort messages fired
3	NetSuite validated	`npx ts-node scripts/test-netsuite-connection.ts`	Ends with `NetSuite connection test complete.` (Part 2 §2.6)
4	*(optional)* Business Central validated	Curl #2 from Part 3 §3.4 (`GET .../api/v2.0/companies`)	`200` with a non-empty `value: []` company list
5	Tenant provisioned	`npm run provision:pilot-tenant -- --tenant squire_pilot --platform netsuite --platform-accou`	Prints `"embeddedTokenPresent": true` (Part 4 §4.2 Step 3)

#	Check	Command	Expected result
		`nt-id <id> --verify --provisioning-output ./pilot-provision.json`	
6	JWT works	`curl -H "Authorization: Bearer $JWT" http://localhost:3003/api/statistics`	`200` with a JSON statistics body (Part 4 §4.4)
7	HMAC secret stored encrypted	Re-run the `repo.upsert(...)` read-back, or attempt a synthetic webhook (row 9) — a wrong/unreadable secret 401s there	Row 9's `202` is the practical proof; a direct DB check should show the `sync_error_assist.webhook_hmac_secret` row with `is_encrypted=true` (Part 5 A2 Step 2)
8	SuiteScript deployed	**Customization > Scripting > Script Deployments** in NetSuite — open your deployment	**Status: Released**, **Applies To: `customrecord_suitecentral_sync_error`** (Part 5 A3)
9	Synthetic webhook accepted	`node send-sync-error-webhook.js` (Part 5 B1)	`202` with `{"status":"accepted","claimId":"<uuid>"}`
10	Suggestion appears in the operator queue	Open `https://<host>/embedded/sync-error-triage.html?embeddedContextId=<sessionId>`	A card referencing the error record ID you just sent appears within a few seconds (Part 5 B2/C1)

#	Check	Command	Expected result
11	Accept writes back	Click **Accept** on a card, or `POST /api/sync-error-assist/suggestions/<errorRecordId>/accept`	A success banner and the card disappears — on your first run, open the created NetSuite record once to confirm the field values populated (Part 5 C1, C3 — both former write-back defects are fixed on this branch; a `503 connector_unavailable` here means the `NETSUITE_*` env vars aren't inside the container)
12	Mapping suggestions return live AI	`curl -X POST https://<host>/api/ai/proxy/mapping/suggestions ...` (Part 6 §6.4) with a provider key configured	Response has `"isDemo": false` and `"isLiveAI": true` (Part 6 §6.1) — `isDemo: true` here means no key reached the container; see the troubleshooting table below

Rows 1–8 are infrastructure/wiring proof; rows 9–12 are the two features actually working. A pilot is "installed" once row 8 passes and "in daily use" once rows 9–12 pass repeatedly, not just once.

7.2 Consolidated troubleshooting

This table folds together every symptom/cause/fix scattered across Parts 1–6 into one lookup. Each row also names the Part/section that covers it in full — use this table to find the right place fast, not as a replacement for reading that section once.

Symptom	Likely cause	Fix	Full detail
Container exits immediately (`docker compose ps` shows `Exited`) — log reads `JWT_SECRET is using a default or placeholder value in production...`	`JWT_SECRET` unset, or matches a disallowed placeholder pattern	`openssl rand -base64 48`, put the value in `.env`, then `docker compose -f docker-compose.prod.yml up -d` (not `restart` — `.env` is only read at container create, Part 2 §2.5)	Part 1 §1.4
Container exits — log reads `DB_PASSWORD is missing or insecure in production...`	`DATABASE_URL` has no embedded credentials and `DB_PASSWORD` is unset/`password`	Use a `DATABASE_URL` with a real embedded password, or set `DB_PASSWORD`	Part 1 §1.4
Container exits — log reads `RATE_LIMIT_ENABLED must be true in production...`	Something explicitly set `RATE_LIMIT_ENABLED=false`	Remove the override, or set it to `true`	Part 1 §1.4
Container exits — log reads `FATAL: DEMO_MODE=1 cannot be used in production environment.`	`DEMO_MODE=1` in production	Unset `DEMO_MODE`, or set it to `0`	Part 1 §1.4
NetSuite connection test: `[!] Authentication failed` (`401`)	Credential mismatch, or the token/integration record was deactivated	Re-verify all five TBA values against what NetSuite showed you; recreate the access token if in doubt	Part 2 §2.7
NetSuite connection test: `Invalid account`	`NETSUITE_ACCOUNT_ID` format wrong (missing the sandbox `_SB1` suffix, or has a	Match Part 2 §2.4 exactly — underscore, no `https://`, no trailing slash	Part 2 §2.7

Symptom	Likely cause	Fix	Full detail
	hyphen instead of underscore)		
NetSuite connection test: permissions error	Access-token role lacks REST Web Services or record-level permissions	Use Administrator for a sandbox pilot, or add the specific permissions to the custom role	Part 2 §2.7
NetSuite connection test: connection refused / timeout / DNS failure	Most often the 5–10 minute feature-activation lag from Part 2 §2.1 hasn't elapsed (base-URL derivation now DNS-normalizes `_SB1` → `-sb1` automatically, so the old sandbox underscore mismatch is fixed)	Wait and retry; if it persists, set `NETSUITE_BASE_URL` explicitly in `.env` (already passed through by `docker-compose.prod.yml`) for a nonstandard domain	Part 2 §2.5, §2.7
Business Central curl #1 (`token`) returns `400 invalid_client`/`unauthorized_client`	Wrong `BC_CLIENT_ID`/`BC_CLIENT_SECRET`, or the secret expired	Re-copy from Entra; create a fresh client secret if in doubt	Part 3 §3.5
Business Central curl #2 (`companies`) returns `401`/`403` after curl #1 succeeded	Admin consent was never granted, or the Entra app isn't Enabled inside BC	Grant admin consent (§3.1 step 3); confirm **State = Enabled** in BC's Microsoft Entra Applications record	Part 3 §3.5
Business Central curl #2 returns `200` with `value: []`	No permission set assigned in BC, or wrong `BC_ENVIRONMENT` name	Confirm permission set assignment and exact (case-sensitive) environment name	Part 3 §3.5

Symptom	Likely cause	Fix	Full detail
Business Central curl #2 returns `404`	`BC_TENANT_ID` or `BC_ENVIRONMENT` doesn't match a real environment	Re-verify both against the BC admin center	Part 3 §3.5
Business Central field catalog in the mapping editor looks generic even after a live connection	The editor's per-system catalogs are fixture-based generic schemas served by `GET /api/ai/proxy/mapping/schemas/:systemType` — no connector is consulted (the still-open schema gap in §7.3). Separately, the server-side `MetadataClient.fetchFromAPI()` always falls back to a bundled fixture `$metadata`, so even connector-level schema browsing isn't live yet	Expected today; upload a data file to get your real fields into the editor (Part 6 §6.3). The §3.4 connector test proves the connection is real; no field catalog is sourced from it yet	Part 6 §6.3, Part 3 §3.4
Webhook (synthetic or real): `401`, bad secret	Signature doesn't match — wrong HMAC secret, or the tenant has no secret configured at all (deliberately collapsed into the same response)	Re-verify the hex secret matches on both sides (server config + NetSuite API Secret); re-run Part 5 A2	Part 5 B1
Webhook: `401`, clock skew	`x-suitecentral-timestamp` is more than 5 minutes off	Sync the sending host's clock; re-run	Part 5 B1

Symptom	Likely cause	Fix	Full detail
	server time (replay-window guard, either direction)		
Webhook: `401`, unknown tenant	`x-suitecentral-key-id` doesn't match a tenant with a configured secret (same response as "bad secret," deliberately)	Confirm Part 5 A2's `upsert` actually ran for the tenant ID you're sending	Part 5 B1, A2
Webhook: `401` from a plaintext-secret row	The HMAC secret was written without `isEncrypted: true`, or via a hand-inserted SQL row that doesn't match the recomputed secret name	Never hand-insert the encrypted secret row directly — always go through `repo.upsert(..., {isEncrypted: true})`; re-run Part 5 A2 Step 2	Part 5 A2
Webhook returns `200 {"ok":false,"reason":"webhook_disabled"}` (not `202`)	HMAC verified fine, but `sync_error_assist.enabled` or `.webhook_enabled` isn't `true` for this tenant	Set both keys `true` via the same `repo.upsert` pattern	Part 5 A2, B1
Operator queue is empty even though a webhook returned `202`	The embedded session's `userRoles` doesn't include one of the three accepted roles (`ops`/`admin`/`finance` — Part 4 §4.3), or `sync_error_assist.enabled`/`.web`	Re-mint the session with `userRoles: ["ops"]`; re-check the two feature-flag keys	Part 4 §4.3, Part 5 A2, C1

Symptom	Likely cause	Fix	Full detail
	`hook_enabled` flipped off after the fact		
Triage queue/accept/reject/escalate returns `403 {"ok":false,"code":"forbidden_role"}`	The session's `userRoles` contains a role string that isn't exactly `ops`, `admin`, or `finance` (e.g. the generic-sounding `"operator"` — not accepted)	Re-mint the host-bootstrap session with `userRoles: ["ops"]`	Part 4 §4.3
Mapping suggestions return `isDemo: true` even though a provider key IS set in your shell/`.env`	The key never reached the container (compose passthrough, or a typo in the var name), or the provider was skipped at registry init (e.g. malformed key format)	Confirm the key is actually inside the running container: `docker compose -f docker-compose.prod.yml exec integration-hub env | grep -i api_key`; check server startup logs for a provider-registration warning	Part 6 §6.1, §6.2
Sync Error Assist write-back returns `503 connector_unavailable`, or the server log shows `NetSuiteEnvCredentialsMissingError` (pre-fix builds: `TokenError: Missing required OAuth1 credentials`)	The five `NETSUITE_*` env vars from Part 2 §2.5 aren't set inside the container — Sync Error Assist's connector now initializes from those deployment-wide vars (a fix on this branch), and the log line names exactly	Set the five vars in `.env` and recreate the container (`up -d`, not `restart` — Part 2 §2.5); confirm they landed with `docker compose -f docker-compose.prod.yml exec integration-hub env | grep NETSUITE`	Part 5 C3, Part 2 §2.5

Symptom	Likely cause	Fix	Full detail
	which keys are missing		
Sync Error Assist returns `503 connector_unavailable` and the server log shows `NetSuiteTenantAccountMismatchError` naming two different account IDs	The tenant was provisioned (Part 4 §4.2) with a `--platform-account-id` that isn't the deployment's `NETSUITE_ACCOUNT_ID` — the write-back guard fails closed rather than let the shared env credentials touch a different NetSuite account (underscore/hyphen and case differences do NOT trigger this; those are the same account)	If the provisioning value was a typo, re-run `npm run rotate-embedded-service-token` with the correct account ID; if the tenant genuinely lives on another NetSuite account, it needs its own deployment — per-tenant ERP credentials don't exist yet	Part 5 C3, Part 4 §4.2
Sync Error Assist returns `503 connector_unavailable` and the server log shows `NetSuiteTenantUnprovisionedError`	The tenant has `sync_error_assist.enabled=true` but Part 4 §4.2's provisioning step never ran for it, so no NetSuite-account binding is on file — the guard fails closed rather than hand an unverified tenant the deployment's shared credentials	Run the Part 4 §4.2 provisioning flow (`npm run provision:pilot-tenant` or `npm run rotate-embedded-service-token`) with this tenant's `--platform-account-id`, then retry	Part 5 C3, Part 4 §4.2
A created `customrecord_sui`	You're running a pre-fix build — the	Upgrade to a build containing the fix; on	Part 5 C3

Symptom	Likely cause	Fix	Full detail
`tecentral_fix_suggestion` record has a non-empty `id` but no field values	empty-payload defect (write payload not wrapped in `.fields`) was fixed on this branch; current code carries all eight field values	current code this symptom should not occur (if it does, verify field values directly in NetSuite and report it)	

Troubleshooting Tip: Timing and Duplicates

(Continuing from above: `-fsS` makes `curl` exit non-zero on a non-2xx response, which is what you want in a script. A `503` with `{"status":"not-ready",...}` means the server process started but something it depends on — most often Postgres — isn't reachable yet; give it a few more seconds and retry.)

If you immediately replay the *exact* same signed request while the original claim is still in flight, the contract is `{"status":"duplicate"}` — this guide's own testing observed both requests come back `accepted` rather than `duplicate` in a from-scratch dev sandbox with no AI provider configured (the first claim failed fast and became retry-eligible before the replay landed), which is a timing artifact of that specific test setup, not a contract violation — the `duplicate` branch is real code (`SyncErrorAssistService.ts`'s `ingestWebhook`), just not the branch this guide's particular replay timing happened to hit.

7.3 Known gaps — what this guide surfaced, and where each stands now

Writing this guide doubled as a defect hunt: every flow was exercised against a running server and read against the real source, and that process surfaced eight real product defects. An earlier draft disclosed all eight as open known gaps; since then, **six were code-fixed on this same branch, one was resolved by documentation, and one was narrowed but remains open**. The lists below are the current truth — what was found, what's fixed, and what's still open. Nothing here blocks completing this guide.

Fixed on this branch (surfaced by writing this guide, then resolved — by code except where noted):

1. **Sync Error Assist's AI-suggestion write-back sent NetSuite an empty payload** — FIXED. The create payload is now wrapped in the `fields` key `formatDataForNetSuite()` reads; a created

`customrecord_suitecentral_fix_suggestion` record carries all eight field values from A1's Table 2. Story and current expectations: **Part 5, Chapter C3, item 1**.

2. **Sync Error Assist's NetSuite connector was never initialized with credentials** — FIXED. All three call sites now initialize the connector from the deployment's `NETSUITE_*` env vars (Part 2 §2.5) before use; if those vars are missing from the container, the accept path returns `503 connector_unavailable` and the server log names the missing keys. A review follow-up hardened the fix for multi-tenant deployments: initialize also fails closed (same `503`) when a tenant's provisioned NetSuite account (Part 4 §4.2's `--platform-account-id`) differs from the deployment's `NETSUITE_ACCOUNT_ID` (`NetSuiteTenantAccountMismatchError`) — or when the tenant has no provisioning record at all (`NetSuiteTenantUnprovisionedError`) — so the shared env credentials can never cross NetSuite accounts, and enabling the feature requires the Part 4 provisioning step first. Full detail: **Part 5, Chapter C3, item 2**.
3. **The AI Field Mapping editor's "Save Template" button was broken for a live (non-demo) save** — FIXED. It now POSTs the real `POST /api/templates/` contract with the required `fields` key and shows the returned template key on success; the demo-mode `localStorage` path is unchanged. Full detail: **Part 6, §6.3** (the Save Template callout); §6.4 remains the API route for automation.
4. **The editor's Apply/Reject buttons never called the telemetry or learning-feedback endpoints** — FIXED. Apply fires `POST /api/ai/proxy/suggestions/:id/accept` plus `POST /api/ai/proxy/mapping/feedback` (accepted); Reject fires feedback (rejected); Apply All is covered — all fire-and-forget, so the UI never blocks on a server failure. Full detail: **Part 6, §6.3**, step 3.
5. **The editor's "Load from Template" read a different, older router than "Save Template" wrote to** — FIXED. Loading now reads the unified `GET /api/templates/` (the same system Save writes to), additively merging the eight legacy built-ins (deduplicated by key; unified wins) — templates saved in the editor or via §6.4's API now appear in the Load list. Full detail: **Part 6, §6.3**, step 4.
6. **NetSuite sandbox base-URL auto-derivation didn't convert the `_SB1` underscore to the hyphen NetSuite's DNS actually uses** — FIXED. Derivation now DNS-normalizes the account ID (`1234567_SB1` → `1234567-sb1.suitetalk.api.netsuite.com`) while the OAuth realm keeps the original form; `NETSUITE_BASE_URL` remains an optional explicit override. Full detail: **Part 2, §2.5**.
7. **No documented first-class way to live-test a stored connector config** — resolved by documentation rather than code. `POST /api/integrations/<config-id>/test` already existed and runs the real connector's `initialize()` + `testConnection()` against your stored `sourceAuthentication`; **Part 3, §3.4** now documents the worked example. Still true and worth knowing: `ConnectorManager.testConnector()` itself is unrouted (the routed test goes through `IntegrationService` instead), creating a config via `POST /api/configurations` proves nothing about connectivity on its own,

and Part 6's mapping-suggestions path never touches a connector — suggestions work directly off the field lists you provide.

Still open:

8. **The mapping editor's field catalogs are generic fixtures, not your tenant's live schema.** Narrowed on this branch, not closed: picking NetSuite/Salesforce/SuiteCentral now fetches a per-system field catalog from the server (`GET /api/ai/proxy/mapping/schemas/:systemType`) instead of one identical hardcoded mock for every system — but those catalogs are fixture-based generic ERP schemas, no connector is consulted, and a custom field in your own account will not appear. Uploading a data file remains the way to work with YOUR real fields. Full detail: **Part 6, §6.3**, step 1.

Known going into the pilot (from the original plan, not discovered while writing this guide):

9. **Three standing calibration gaps**, all disclosed up front in the beta disclosure at the top of this guide:
    - **Webhook secret rotation has a brief delivery gap.** A webhook signed with the old secret is rejected between updating the server side and the NetSuite side; the 30-minute polling reconcile (Part 5 B3) is the backstop — nothing is silently lost, but rotate in a low-error window.
    - **The pre-pilot error-fixture corpus is Claude-drafted, not drawn from Squire's real error history.** It proves the mechanism works end-to-end; it does not prove accuracy on your data.
    - **The ≥50% AI-suggestion accept-rate target (from `35-SQUIRE-DEPLOYMENT-PLAN.md`) has never been measured on live data.** Treat your first weeks of pilot data as calibration, not a pass/fail grade.

Where to report pilot accept-rate data. `35-SQUIRE-DEPLOYMENT-PLAN.md`'s pilot checklist calls for a "Sync-error accepted-fix rate vs the ≥50% target (with the synthetic-corpus calibration caveat)" as part of the go/no-go memo with the Squire sponsor, alongside the mapping accept-with-no-edit rate and the DLP false-positive count from operator feedback (same file, pilot-checklist section). `13-PILOT-30-60-90.md` is the pilot-governance frame this rolls up into — sponsor, owner, and a weekly KPI pack. Use the audit-log export (Part 5 C2's per-disposition rows) and the AI cost roll-up (also C2) as your source data for that memo; neither this guide nor the codebase auto-generates the memo itself.

End of guide. If you've worked through Parts 1–7 in order, you have a running SuiteCentral 2.0 server connected to your NetSuite sandbox (and optionally Business Central), Sync Error AI

Assist producing reviewable suggestions from real sync failures, and an AI Field Mapping workflow you can drive from the editor or the API — with the defects this guide surfaced fixed on this same branch, and everything still open disclosed above rather than hidden. The next step is yours and Squire's: run a real pilot window, capture the KPIs Part 7.3 points at, and use that data — not this guide's fixture numbers — to decide whether the ≥50% accept-rate target is met.