SOVEREIGN · GOVERNED · SELF-HOSTABLE

Sovereign Agentic OS

The governed, EU-sovereign operating system for data, knowledge, agents and software — where AI gets real, safe hands on your work.

Orchestrated by Data Masterclass • datamasterclass.com • www.sovereign-agentic.com
Chart 0.2.11 (app 0.2.0-alpha.11 • os-ui 0.6.23) • generated 2026-07-29 from commit b17f73b

Contents

1 The one-paragraph version

The **Sovereign Agentic OS** is a self-hostable, EU-residency operating system for your data, knowledge, agents and software — a single **governed** stack where every action, taken by a person *or* by an AI, runs **as you**: OPA-policy-checked, row- and column-secured, and Langfuse-audited. It assembles roughly two dozen best-in-class, permissively-licensed open-source tools — a Trino/Iceberg lakehouse, a Cube semantic layer, OpenSearch retrieval, a LiteLLM model gateway to sovereign EU models, LangGraph agents — into one operating model you learn once and apply everywhere. Its promise is simple and, we think, important: it gives AI **real, safe hands on your data**. The web UI and the AI (over MCP) travel the *exact same governed path* — there is no privileged back door — so you can hand an agent the keys to your lakehouse and know, provably, that it can never do anything you couldn't do yourself.

> **Where to go next.** Curious what it feels like? Read *The guided tour*. Want to run it? Jump to *Quickstart*. Coming from a governance or security background? Start with *The governance model*. Contributing code? *How to contribute* and `os-ui/ARCHITECTURE.md` are your on-ramp.

2 Why this exists

For two years the industry has been stuck on the same tension. **Agents are only useful when they can act** — read the warehouse, write a table, ship an app, call a tool. But the moment an agent can act, the honest questions arrive: *Whose identity is it running as? What is it allowed to touch? Where did the data go? Can I prove, after the fact, exactly what it did?* Most stacks answer these by wrapping an autonomous agent in ad-hoc guardrails bolted on after the fact — and by shipping your prompts and data to a US-hosted model.

The Sovereign Agentic OS answers them structurally, with three ideas doing all the work:

1. **One governed path for humans and AI.** Every write — from a button click, from a tab's built-in assistant, from Claude or ChatGPT over MCP, or from an autonomous agent — flows through the *same* function: `authorize → act → trace`. Governance isn't a feature you can forget to turn on; it's the only road in.
2. **Sovereignty is the substrate, not a badge.** The production target is **STACKIT** Kubernetes with STACKIT Object Storage in **EU01 / Deutschland Süd**, and model calls route to **STACKIT AI Model Serving**, so prompts and completions never leave the EU boundary. Default-deny egress means an agent has *no* raw internet unless you grant it.
3. **Permissive open source, end to end.** Every bundled component is Apache-2.0 / MIT / BSD / ISC-class licensed — full auditability, no proprietary lock-in, the right to host and modify it forever. The core is **Apache-2.0**, and the permissive posture is *enforced*, not merely promised: an automated `check:licenses` gate (`license-checker --onlyAllow`) rejects any dependency outside a strictly permissive allowlist (MIT, Apache-2.0, ISC, BSD-2/3-Clause, 0BSD, and a few equivalents) — so no copyleft can slip in — with a CycloneDX SBOM (`sbom.cdx.json`) and a full attribution manifest (`THIRD-PARTY-LICENSES.md`) alongside.

2.1 Who it's for

- **Regulated organizations, the public sector, and EU enterprises** that need data residency, a complete audit trail, and zero dependency on US-controlled cloud or hosted LLMs — but still want production-grade agentic workflows.
- **Data and platform leaders** who want *one* place for the lakehouse, the semantic layer, the knowledge spine, BI, and software delivery — instead of a dozen disconnected tools and a governance story stitched across all of them.
- **Curious engineers and OSS contributors** who want to see how a governed agent runtime is actually built — and to extend it, one clean tab-module at a time.
- **Data Masterclass participants** building real agentic systems on the *same* production components used in the field, not a teaching fork.

2.2 The problems it solves

THE TENSION	HOW THE OS RESOLVES IT
Governance vs. agent autonomy	Agents are genuinely autonomous *inside* a policy envelope. Every tool call is OPA-authorized; anything out of scope doesn't fail silently — it queues to a human as an approval card.
Sovereignty vs. capability	The full stack runs in your EU region on open source. Models are sovereign (STACKIT) yet swappable; nothing phones home.
Ten tools vs. one source of truth	One lakehouse, one semantic layer, one knowledge spine, one identity, one audit — so BI, agents and dashboards can never disagree about "revenue."
Demos vs. production	The same Helm chart runs on a laptop (`kind`) and on a sovereign STACKIT cluster. The only difference is a values choice per backend.

3 The operating model — learn it once

One idea ties every screen together. Internalize it here and the whole product reads the same.

3.1 Everything is a governed artifact

Whatever you make — a dataset, a knowledge workflow, a file, an agent, an app, an ML model, a metric, a connection, a dashboard — is an **artifact** with the same four attributes: **owner** • **domain** • **type** • **visibility**. Whatever its type, it travels one lifecycle:

> **Create → Document → Use → Promote** — authored in the UI (which scaffolds the *real* tool underneath: a dbt model, a Cube metric, a Forgejo repo, a KServe service), preview-first, cataloged and audited.

3.2 One Builder Framework — every build tab reads the same

Creating any non-trivial artifact happens in a **staged builder**, and every builder is the *same* shape (one core primitive — `lib/core/stages.ts` + `components/core/StageShell.tsx`). Learn the Agents builder and you can drive the Data, Metrics, Dashboards, Software and Science builders, because they all share four traits:

- **A numbered five-stage flow** with a stepper rail. Each tab names its own five stages, but the machinery is identical: **Agents** — Define • Design • Build • Run • Evaluate; **Data** — Ingest • Define • Harmonize • Validate • Publish; **Metrics** — Define • Refine • Preview • Publish • Monitor; **Dashboards** — Define • Design • Build • View • Govern; **Software** — Define • Design • Build • Test • Publish; **Science** — Define • Train • Deploy • Predict • Monitor.
- **Honest, gated navigation.** A stage is reachable only when its precondition is met (you can't Harmonize to Gold before Silver exists), and a stage shows a ✓ only when you completed it *this session* **and** its live condition still holds — a ✓ clears the moment you invalidate it. A fresh artifact opens on its first incomplete stage with **no** pre-marked checks. No faked green.
- **A per-stage assistant.** The Sovereign-OS AI helper on the right of the rail is scoped to the *current* stage — it knows you're defining a metric, or harmonizing a Gold join — and, like every assistant, it acts through the same governed tools.
- **Simple ⇄ Developer, and lifecycle-in-header.** A **Simple** view keeps the guided flow calm and NL-first; a **Developer** view exposes the raw technical surface (the dbt SQL, the Cube YAML, the repo tree). The artifact's name, visibility badge and lifecycle controls (Archive • Restore • Delete • Promote) live in a **persistent header** above the stages, always reachable mid-flow.

3.3 The sharing ladder

Visibility widens **one rung at a time**, and each move is strictly **two-step** — the person who *triggers* a promotion is never the person who *approves* it:

The OS speaks **one scope vocabulary everywhere — My • Domain • Company** (defined once in core, `lib/core/scopes.ts`). "My" is your private draft space; "Domain" is your team; "Company" is the whole tenant. Promote reads **"Promote to Domain"** and certify **"…to Company"** — the same two verbs on every tab.

Your "My" work is yours — no approval, ever. A builder — *and the agents they build* — create and write their own **personal** artifacts of every type (data, files, knowledge, metrics, connections, software, dashboards, agents, science) **directly**, with no admin review. The person's own rights and ownership are the authority. Approval only appears the moment you widen the audience:

SCOPE	MEANING	WHO TRIGGERS	WHO APPROVES
My	you only — the default; **full rights, no approval**	—	—
Domain	usable across the owning domain	the **owner** files a promotion request	a **domain admin** of that domain
Company *(certified)*	discoverable and importable by *other* domains, listed in the Marketplace storefront	a **Builder / Domain admin** — the domain vouches for it	the **tenant Administrator** — the platform accepts it

You can only change what you originally built. Edit rights follow ownership, not tier — even a shared artifact is editable only by its original owner (and, for Domain/Company artifacts, an in-scope admin); a personal "My" artifact is strictly owner-only, invisible to admins.

Approving *is* the action. High-stakes actions never fail silently: when you promote, certify or request a deploy you get one calm confirmation — *"Request filed — awaiting approval to Domain/Company"* — with a **Go to Policies & Approvals →** button that deep-links to and highlights the exact request. An admin who *can* approve it sees an **Approve now** button that approves inline (fail-closed — non-approvers never see it, and the server re-checks). On approve, the platform executes the governed effect — for a dataset, a physical publish; the tier flips only once it verifies — and writes the audit. Nothing enters the governed store without **documentation + passing checks**: a transparency gate that turns green only when an artifact is documented and in the lineage graph.

3.4 Four roles, assigned per domain

The ladder is exactly **creator < builder < domain_admin < admin**.

ROLE	WHAT THEY DO
Creator	the base role — creates and runs their **own** artifacts (My scope by default) and consumes anything at Domain or Company scope. Files promotion requests; cannot approve.
Builder	the domain **approver** — everything a Creator can, plus review/approve My→Domain promotions, deploys, knowledge and connections. An approver, *not* a people-admin.
Domain admin	everything a Builder can, plus administering the users of their **own domain(s) only** — invite, edit, assign roles **up to Builder**. Never mints another Domain admin.
Administrator	tenant-wide — the only role that appoints **Domain admins**; sets policy, certifies to Company scope (the Marketplace storefront), sets cost caps; runs the Admin section.

Roles are assigned **per domain** and **compiled to OPA**, so a person who is a Builder in one domain and a Creator in another sees exactly the right controls in every tab, instantly.

3.5 Every assistant acts, and acts as *you*

Each tab has a built-in assistant, and there is one **overarching assistant** — **Ask the OS** — present on every tab. Neither is a chat box. Each runs one loop: **PLAN** with the reasoning model, then **ACT** in a bounded tool-calling loop with the worker model — calling that tab's governed tools, which are the *same* OPA-authorized, Langfuse-traced functions the UI uses. Crucially, **Ask the OS is a client of the OS's own MCP server**: it dispatches through the exact same governed path (`handleRpc`) that Claude Desktop uses, under *your* delegated identity. It inherits identical guardrails — role floor, approval gates, OPA/RLS, audit — and it is transparent: each answer lists the governed tools it invoked, and if governance *blocks* a call it says so plainly. There is **no privileged side-channel**.

4 The guided tour

Open the OS in a browser and you land on a left sidebar organized into named sections. Here is the whole map, in sidebar order — each surface in a few vivid sentences, with what you *do* there.

4.1 Entry

- **Home — the golden-path launcher.** The warm front door after you pick a domain. An illustrated launcher of the golden paths (Data, Knowledge, Agents, Software, Science, Metrics, Dashboards, Big Bets, Marketplace, Connections), each card with a role-aware action. It *only* orients and routes — the live view lives one click away in Cockpit.
- **Cockpit — what's moving, what needs you.** A persona-ordered live overview: a pulse strip (*Needs you • In progress • Your items • Spend* vs. cap), your work-in-progress, and a scannable "top items, by type" board. Cockpit *reads and routes* — it never recomputes another tab's numbers and never bypasses governance.
- **Marketplace — consume across domains.** The *Consume* counterpart to every tab's *certify to Company* step: discover and reuse Administrator-certified (Company-scope) products of every type across the tenant. The storefront keeps its name; importing is a **governed grant**, not a copy — the default is read-in-place, under your own identity and row-level security.

4.2 Plan

- **Strategy — pillars, value and adoption.** Where the company plans its agentic transformation, in exactly three calm sections: **Strategic Pillars** (each with a big value target; Big Bets are created under them), **Self Service** (how broadly your people build for themselves), and **Foundations** (the certified asset base every bet builds on).
- **Big Bets — initiative roadmaps.** A strategic AI bet as a **goal + dated roadmap** built from real artifacts across the platform. **Every bet sits under a Strategic Pillar** — creation requires one from both entry points (the Big Bets "New" panel and "New bet under this pillar" from Strategy) and via MCP. Status derives *live* from each artifact's real lifecycle; the roadmap rolls up on-track / at-risk.
- **Operating Model — how the company runs, at three scopes.** *My / Domain / Company Operating Model*, each a fixed set of sections — **General • Strategy • Business • Organization • Architecture • Data • Glossary** — governed per scope (My = owner, Domain = domain_admin+, Company = admin). It's the durable, structured backbone agents can be granted as context.
- **Workflows — the process spine.** A **workflow** per business process (ordered steps • rules • know-how, each step owned by a Human / Software / Agent / external actor), retrievable and grantable to agents.
- **MCP** *(Builder+)* — the setup surface for connecting external AI clients over MCP.
- **Tutorials.** An illustrated, hands-on tutorial for every tab — fourteen today, each kept in step with its tab's current journey — reached from Home or a tab header, that can spotlight the real

controls and let you practice in a sandbox before doing it for real.

4.3 Context

- **Knowledge — the domain's captured know-how.** Human-authored reference knowledge, made retrievable by a knowledge agent behind document-level security. Mark a decision rule **hard** and it compiles into an OPA guardrail. (The structured backbone — Operating Model, Strategy, Big Bets, Workflows — lives in the Plan section; Knowledge is the reference library.)
- **Files — a calm governed drive.** Any unstructured file — documents, images, audio, video — uploaded and auto-indexed (parse → embed → hybrid OpenSearch) so agents can search and cite it. Governed exactly like Data; *"Use as"* distils a file into Knowledge or Data.
- **Data — datasets, refined and governed.** A five-stage medallion builder turns a plain-language flow into real governed artifacts (a dlt pipeline, dbt models, a Cube cube), with no YAML: **Ingest** (land a file as a Bronze Iceberg table) • **Define** (document the columns, clean and conform to Silver) • **Harmonize** (join into a Gold business mart — a join is optional, so a single-table Gold is fine) • **Validate** (quality checks + lineage) • **Publish** (sharing first, then Talk-to-Data, then the metrics, dashboards and agent systems already built on this data — each list links out and offers a create pre-scoped to the dataset). Silver and Gold builds are never one-shot black boxes: after a build the definition stays visible and editable, so you explore the result, tweak, and **Rebuild in place** — and *you* choose when to Continue. The **Validate** stage is a real data-quality gate: author dropdown-driven rule checks (`not_null`, `not_blank`, `unique`, `accepted_values`, `range`) that compile to SQL and run *for real* against the built table, get a 0–100 **health score** and a passing/failing badge (honestly `unknown`, never a fake pass, when nothing ran), let the OS **suggest rules from the column profile**, and watch three heuristic **monitors** — freshness, row-volume and schema stability — that learn each dataset's normal band from its own run history. Then **Talk to your data**: governed NL→SQL, one validated read-only `SELECT`, executed under your row filters. A **Developer** view exposes the raw dbt SQL and the physical table behind the guided flow. A table imported from an external warehouse doesn't go stale either: a **"Keep this in sync"** panel schedules regular refreshes — **Full refresh**, **Add new rows** (incremental append over a cursor column, with a late-data lookback window), or **Update by key** (a merge) — Hourly / Daily / Weekly or your own cron. Every run executes **as the dataset's owner** through the same governed Trino path as the original import, with the cursor predicate pushed down to the source — no data ever flows through the app — and each slice lands in Iceberg carrying `_loaded_at` / `_batch_id` lineage columns. The dataset shows its **sync history and watermark**; ten consecutive failures **auto-pause** the schedule (with a one-click *Reset & full re-sync* recovery), and freshness in **Monitoring** reflects the last sync. Scheduled sync is built for *incremental slices*, not bulk backfills: the current per-run limits (statement timeout, single-node Trino sizing, object-storage capacity), the honest estimates behind them, and the concrete scale-up playbook — exact Helm values, backfill windowing, and when to switch to staged-file loads — are documented in `docs/data-sync-scaling.md`.
- **Connections — governed bridges to outside systems.** A Connection is `credentials + endpoint + a set of governed tools`, never a raw pipe — used to bring data in and to expose external APIs/MCPs as tools. You grant **use**, never the token; **reads are automatic, writes are**

approval-gated (destructive ops blocked), and secrets are write-only. The Supported Connectors gallery is **grouped by vendor stack** (Microsoft • Google • AWS • Databricks • Snowflake • Salesforce • Kajabi • Atlassian • Open source • Other) and searchable. When nothing in the gallery fits, a **Custom Connector** lets you add your own **REST/GraphQL API** or **MCP server** in one governed action: you name it, give the base URL and a write-only credential, and the OS **atomically files the egress-allowlist request** for that host — reads auto-allow, writes stay off until you enable them per-tool, and the far-side host cannot be reached until an Administrator approves the egress. The real catalogue now spans: **operational databases** (PostgreSQL • MySQL • SQL Server • MongoDB, federated through central Trino); **code & DevOps** (GitHub); **docs & knowledge** (Notion, Atlassian); a Supabase connector; **messaging & calendar** (Slack • Gmail • Google Calendar • Outlook • Teams — sending a message or email is always approval-gated, never automatic); **cloud governance / ML** (Microsoft Entra • Purview • Azure AI Foundry • AWS SageMaker, read-only); plus the established data-ingest (Google Drive / OneDrive), orchestration (Airflow) and catalog (**OpenMetadata**, read/discover of a customer's existing catalog, DLS-clamped to the tables the caller may already see — and now **active-only**: when you archive an OS dataset the integration best-effort **soft-deletes** its OpenMetadata entity, and unarchiving restores it, so the catalog reflects what's live rather than accumulating ghosts) connectors. Setting up the OAuth app / tokens on the far side is the **operator's step** (each connector ships an install guide). For teams already running a lakehouse elsewhere, an admin-enabled **external-warehouse** connector federates it through central Trino as a governed catalog — AWS Glue/Athena, Snowflake, BigQuery, Databricks/Delta, and (experimental) Microsoft Fabric/OneLake — so you can query it in place under the same OPA path, or import a core table as a governed data product into the sovereign lakehouse — and keep the copy fresh with a scheduled sync (see *Data* above), so an import is a living dataset, not a one-time snapshot. Operational sources are first-class here too: PostgreSQL / MySQL / SQL Server sync on a timestamp or id cursor, **Kafka** topics land append-only on a per-partition offset cursor (de-duplicate downstream), **Salesforce** objects sync incrementally by `SystemModstamp` over the REST API, and **Kajabi** resources sync over its public API with honest per-resource cursors (purchases incrementally by `updated_at`; contacts/customers/orders by `created_at` — new records only, edits need a full refresh; resources without a documented cursor are full-refresh only) — schedules run from every 15 minutes (append recommended at high frequency; frequent merges accumulate delete files) up to weekly. And for BI on your own desktop, a **one-click Power BI** button downloads a `.pbids` file that drops Power BI Desktop straight into the pre-filled PostgreSQL connector for the **Cube SQL API** — connecting as the `bi_<domain>` principal in **DirectQuery** mode, so per-domain row security re-runs on every query and no password is ever written into the file.

- **Metrics — one number, everywhere.** The KPI semantic layer. Define "Revenue" once and it resolves to the *same* number in the explorer, in dashboards, and in an agent's `metrics` tool — each under the viewer's own row-level security.

One folder UX on every context tab. Files, Data, Knowledge and Metrics all share the *same* folder experience (one core primitive, `lib/core/folders.ts`, with each tab registering a thin adapter — no per-tab divergence). Each tab shows a scope segment (My / Domain) and a single **folder rail tied to the active scope** — you only ever see the root that matches. You **create, rename and move**

folders and items through a **folder-tree picker** (browse-and-click, with inline New-folder — never a text field); moving a folder carries its contents. Lifecycle is the shared one too: **Archive** a folder and it cascades to the items inside (with a warning — move items out first to keep them active); **Restore** or **Delete** (physical delete is archived-only, per-item permission-checked).

Talk to any Context tab. Every tab above carries a read-only **"Talk to X"** copilot. It builds a security-scoped overview of what *you* can see on that tab, runs the tab's own governed retrieval **as you** (Data → NL→SQL, Knowledge → knn retrieval, Files → file search; Metrics and Connections grounded on their catalog), and packs it within the model window via the Context Assembler. Answers arrive with the model's **reasoning shown separately** (a collapsible "thinking" panel), real citations, and a "what ran" disclosure — and it degrades honestly rather than inventing an answer when retrieval comes back empty.

4.4 Build

- **Agents — compose, govern, run.** A domain's **agent systems** (instructions + tools + memory) move through **five phases — Define • Design • Build • Run • Evaluate**. In **Define**, **"What your team can use"** is the interactive grants surface, and a grant here is a **default-on capability**: **every agent in the system inherits the full set of the system's Define grants by default** — you *narrow* an agent to give it less, never scramble to add. Per item you choose **read-only • read + propose • read + write** (a clear labelled selector), capped by the system's overall access setting; a **read + write** grant provisions that agent's matching **write tools** (e.g. `upload_file`, `create_dataset`), and a hard invariant guarantees an agent can never exceed the team's grants. Grants split into two groups — **Plan Items** (Strategy • Big Bets • Operating Model • Workflows) and **Context** (Knowledge • Files • Data • Connections • Metrics) — and **all four Plan Items are grantable**: granting a pillar or bet provisions its governed read tools (`get_pillar` / `get_big_bet`), DLS-scoped to what the caller may view. Context items grant via a **folder-tree with tri-state checkboxes** — tick a folder to grant everything in it (and future contents, resolved at run time, budget-capped, every resolved item still per-item DLS/OPA-checked so a folder grant is provably a *subset*), or tick individual items. Each **data grant** can target the **medallion layer** the team reads — Bronze, Silver, or Gold — and the picker only offers layers that are actually built, defaulting to the highest (Gold, the curated default). **Design** composes the agents three equivalent ways — a React-Flow graph builder, Monaco YAML editing, or a chat assistant. **Build** (*Build = execute + verify*) provisions and checks the team behind a live progress stepper; **Run** executes it as you and offers a **"Download PDF Results Report"**; **Evaluate** attributes context per agent and offers a **"Download PDF Evaluation Report"** — both fully brand-styled (gold-lotus cover, embedded datamasterclass fonts). Every call routes through **LiteLLM → OPA → Langfuse**.
- **Software — a governed frontend over the OS API.** An app here isn't a black box you bolt on — it's a first-class client of the OS itself. It moves through the shared five-stage builder — **Define** (state the purpose and grant the app its context) • **Design** (epics + user stories) • **Build** (the Simple view puts the app's *structure* first: the **Epics & stories tree** from Design, each story with an honest status chip — *to do • building • done • blocked* — and a *"N of M stories built"* count; **click a story to make it the build target** for the run, click it again to build the whole app. Beside it, a **live streaming build**: the AI streams its plan, then one honest, human-readable line per

action — *"Committed 3 files"*, *"Provisioning preview…"* — with errors shown as warnings with the real reason and retries visible in the feed, plus a real before/after **file diff** of what was committed per run; the raw code panel lives one click away in the **Developer view**) • **Test** (a *"Verify & Improve"* pass — the reasoning model checks each built story against its spec across five dimensions — Functionality • User Experience • Code Structure • Security • Documentation — and turns any shortfall into a tracked refinement with a visible **Proposed → Designed → Built** state, beside a live in-cluster preview pod) • **Publish** (the Builder-reviewed deploy, promote / certify, the live tool surface, and lifecycle). A persistent **Build ▸ Preview ▸ Deploy status rail** keeps the honest, single-glance state in view throughout. At **Define** you pick one of **four scaffold templates** — **Application** (the default, full OS UX), **Website**, **APIs only** (no user interface), or **Empty app** — and each pre-shapes the epic structure the Build stage works through. The default is the **Sovereign standard app template** — a Vite + React + TypeScript SPA that already *is* an OS app before the first story is built. It boots *in the OS design*: it vendors `@sovereign-os/ui` (the gold-on-black AppShell and `.sb-*` primitives, no build step, no registry) and calls back into the platform through the **OS-client SDK** (`@sovereign-os/app-sdk` — `createOsClient().whoami()`, `.datasets.list()`, `.metrics`, `.knowledge`), so a brand-new app already renders *real governed data* under the signed-in user's own row security. Sign-in is **delegated to the OS session** — no local accounts or passwords, ever. The app shows its **owning-domain badge** and ships **My / Domain scope helpers** that filter every record by domain + user; an **Admin section** (OS `domain_admin` / Administrator only) carries a **read-only directory** of the OS users who can reach the app — managing users stays in the OS; and a top-bar **MCP button** links to the app's own MCP connection in Connections. The scaffolded **README is the build contract** — it documents how each story adds a page + section, and the Build assistant reads the same text as context. Each built story page is **auto-wired into the app's navigation**: the OS deterministically regenerates the section registry from the committed story pages on every commit, so a written story can never be left invisible ("builds fine but no feature shows"). Code commits to an in-cluster **Forgejo** repo (no GitHub, no tokens, your code never leaves). Commits are **sha-aware** — every write fetches the live blob sha first, so concurrent edits can never silently no-op or overwrite — and the pipeline status is **earned, not claimed**: the CI badge reflects the actual outcome of the latest Actions run on your commit (a missing repo secret even self-heals on the next push), and a build that *fails* is loud — the Test and Publish stages say plainly that the app is serving an *earlier* release with your recent changes undeployed, so a broken build never masquerades as "complete." An app can still **declare its surface** — `surface: ui | api | both` in `app.yaml`, which wins over auto-detection so a Streamlit/Gradio/Flask UI is never mislabelled "API." *Request deploy* assembles a review card — a security scan of the **live repo tree**, resource envelope, diff — that a human Builder decides in **Policies & Approvals**; on approve the in-cluster runner provisions a real Deployment + Service + Ingress with a live per-app URL (a `vite-os` app publishes as **static** files served by nginx). Apps carry the same lifecycle as every other tab — **Archive → Restore / Delete**.

- **Science — classic ML** *(opt-in, Layer 4)*. Take traditional ML (regression, forecasting, clustering — *not* LLMs) from a governed data product to a deployed model-as-service, exposed as both a REST `predict` API and a `predict` MCP tool. Off by default; GPU is cost-gated.

- **Dashboards — governed BI, rendered natively.** Dashboards are built and rendered *in the OS* — **Apache ECharts on the governed Cube semantic layer** — so BI and agents can never disagree. The staged flow: **Define** (name it and bind **one governed Cube view** via metric chips) • **Design** (the panel designer — metrics, dimensions, time grain, filters, and big-number / line / area / bar / pie / table viz types, with a live preview) • **Build** ("Save dashboard") • **View** (every panel queries Cube **as the viewer** — per-user row-level security, with a live/offline badge; switch *View as* and every panel re-queries as that viewer) • **Govern** (reports, promote / certify, and **Connect tools**). Connect tools are the Tier-2 BI bridge over the **Cube SQL API** as a domain-scoped read-only principal: **one-click Power BI** (a pre-filled `.pbids` file), **Tableau** connection fields, and — when the operator has configured it — an **"Open in Superset →"** link to Superset's own console in a new tab, never embedded.

4.5 Monitor & Admin

- **Governance** *(Builder+)* — the control plane: one Approvals inbox for every side-effectful action, the consolidated policy view, the hash-chained audit, cost caps, and Users & access.
- **Monitoring** *(Builder+)* — artifact observability, scoped to your identity and strictly read-only. Two tile boards, each My / Domain / Company: **Agent Monitoring** — every agent system with its real last-7-day telemetry (runs, last run, warnings/errors, and **Tokens truly measured**, captured from the model gateway per run; **Cost** appears only when model pricing is configured via `MODEL_PRICES_JSON`, otherwise an honest "—", never a fake 0) — and **Data Monitoring** — every dataset with freshness, pipeline health and its DQ status (a red *"N DQ rules violated"* is the cue). Open any tile for the full diagnosis view — it takes over the main window like every other tab's detail: run history with per-node drill-down, the system profile (agents, models, grants), governed tool-call traces, cost/token trends, and for datasets a **Data-Quality dashboard**. It also rolls up **data quality** across your datasets: a risk-ranked board (riskiest first) built from the same quality-run history, a domain health average, and an honest count of datasets that have **never been checked** — surfaced as a gap, not painted green.
- **Console** *(Builder+)* — the governed **Query** surface: Lakehouse SQL runs through Trino under the caller's own OPA row/document-level security, so a builder can explore data safely. The **raw Shell** and the unscoped Cube query mode stay **admin-only** (in the UI *and* the API). For developers who'd rather stay in their own terminal, the same governed door is a CLI: `sos` (`cli/sos/`, Phase 0) is a thin Go client that signs in with OAuth 2.1 PKCE and then runs `whoami`, `datasets list` / `get`, and `query "<NL or SQL>"` (or `--metric`) — **every call over the OS MCP front door, as you**, with role, domains, OPA and row/document security re-resolved live on the server. It holds only a short-lived token in your OS keychain; there is no privileged side-channel.
- **Components** *(Admin)* — the one operator surface: every platform service with live health and version, and one-click same-origin consoles (Superset, Forgejo, Dagster, …) via SSO.
- **Admin** *(Builder+, filtered)* — the tenant control room (domains, users, models, egress, cost, backups). A builder sees the tab too, filtered to a single **My Settings** self-service tile; every

tenant-admin tile stays admin-only and hidden, and deeper admin sub-pages redirect non-admins back (fail-closed, default-deny). **About** carries the license inventory.

5 The golden paths — walked end to end

Abstract governance is easy to nod at and hard to feel. So here is the OS at work on a single, concrete story that runs through the whole guide: **Northpeak Commerce**, a fictional mid-sized European omnichannel retailer, whose team wants to optimize marketing-campaign budget with agents. (It's the exact case study seeded into the live teaching cohort — see *The live teaching cohort* — so every step below is a real, governed flow, not a mock-up.)

5.1 Golden path 1 — Data: from a CSV to a queryable metric

Meet **Mara**, a Creator in the `sales` domain. She has a `campaign_master.csv`.

1. **Create & ingest (Bronze).** In **Data**, Mara creates a dataset and uploads the CSV. Her bytes land as a real Iceberg table in her *own* per-user schema (`iceberg.personal_mara.bronze_campaign_master`) — registered only when apply **and** a governed verify both pass. No fake green ✓.
2. **Clean it (Silver).** She presses *Turn into Silver* with guided ops (cast types, drop dupes, set the key). The OS compiles **one** allowlisted CTAS into her schema and runs it as her — OPA masks every read.
3. **Harmonize (Gold).** *Turn into Gold* joins the campaign, margin and CAC datasets on a reconciled key. On success the Gold **auto-registers as a Cube model**.
4. **Validate — quality & lineage.** In the **Validate** stage Mara authors a few checks (`not_null` on the key, `unique` on the campaign id, an `accepted_values` list for `channel`, a `range` on `spend`) — or lets the OS **suggest them from the column profile** — and runs them for real against the Gold table. She gets a **health score** and a passing badge, plus freshness/volume/schema **monitors** that will flag the table if its next load arrives late, comes in thin, or changes shape. The lineage graph shows exactly where the Gold came from.
5. **Document & promote.** Documentation is the gate: Mara adds a description and a tag, then files a promotion request. She's a Creator — she *cannot* approve her own work.
6. **A Builder approves — and the publish runs. Ben**, a Builder in `sales`, approves. The approval independently verifies the physical gold materialized in the domain schema (`iceberg.sales.gold_campaign`), then flips the tier and writes the audit.
7. **One number, everywhere.** In **Metrics**, `revenue`, `aov`, `conversion_rate` and `churn_rate` now resolve on the Gold cube, sliceable by `region`, `product` and `date` — no SQL. Anyone who asks "what's revenue?" — a dashboard, an agent, the explorer — gets the same answer, under their own row filters. (These metrics resolve *reliably* now: Cube recompiles deterministically whenever a model is written, so a freshly-promoted cube is queryable without a restart — see *The lakehouse & semantic layer*.)
8. **Talk to it.** Mara opens **Talk to your data** and asks a plain-English question. The model is shown only datasets she can see, generates one validated read-only `SELECT`, executes it through governed Trino under her masks, and answers grounded only in the returned rows.

(This entire path is also available over MCP — `create_dataset` · `ingest_dataset` · `transform_silver` · `build_gold_join` · `document_dataset` · `request_promotion`, then a Builder's `approve_promotion` runs the physical publish. Same governed functions, no back door.)

5.2 Golden path 2 — Knowledge: capturing how the work is done

Northpeak's campaign playbook lives in people's heads. Let's make it retrievable.

1. **Author a workflow.** In **Knowledge**, Ben authors a *"Campaign budget decision"* workflow: ordered **steps** (each owned by a Human / Software / Agent actor, with inputs/outputs), **rules**, and **tacit** know-how — the gotchas and the "why behind the why," which get indexed as first-class retrieval units.
2. **Mark a hard rule.** *"CAC above target for 14 days ⇒ never INCREASE budget"* is marked **hard**, so it compiles into an OPA guardrail an agent must respect.
3. **Index & verify.** `index_knowledge` chunks and embeds the workflow into OpenSearch; a quick `search_knowledge` confirms it surfaces. Indexing is *not* automatic — this step is what makes it findable.
4. **Publish.** A Builder publishes it to **Domain** scope, so every domain agent can ground on it. Tacit notes carry provenance, and agents must cite the source.

5.3 Golden path 3 — Agents: a governed team with real hands

Now the payoff. Mara builds an agent team that reads the campaign data, respects the playbook, and recommends a budget next-best-action.

1. **Compose.** In **Agents**, Mara drags an *analysis* agent and a *recommendation* agent onto the React-Flow canvas (or edits `system.yaml` directly — same versioned file). Each agent's `AGENT.md` grounds it in the published campaign knowledge.
2. **Grant resources + tools.** In "What your team can use" she grants the system the Domain-scope campaign datasets, the knowledge workflow, and the `query_data` / `search_knowledge` tools, each at read-only. A validation gate must pass; a sub-agent's grants are always a strict subset of the system's.
3. **Pick models.** The single **Auto / Reasoning / Execution** toggle shows the real gateway model names (`sovereign-reasoning`, `sovereign-default`) with an internal/external badge.
4. **Build = execute + verify.** *Build* runs the compiled system and checks it — every call routed **LiteLLM → OPA → Langfuse**.
5. **Run — governed all the way down.** *Run* executes the team **as Mara**. Every tool call the team makes dispatches through the same governed door as her own MCP calls: grant-scoped, OPA-pre-gated, role-floored. The team returns *INCREASE / CUT / HOLD budget for X days + reasoning*. A write pauses for approval and enqueues in **Governance** — the agent is *propose-don't-commit* by default.
6. **Evaluate — see what each agent actually used.** The **Evaluate** view attributes context *per agent*: exactly which datasets, docs, files, metrics and connections each agent read, how (tool +

read/retrieved/written + a short args hint), each a **clickable deep link** that opens the real artifact (switching scope so it's visible). A granted-vs-used strip flags dead grants. So Mara can prove the team grounded on the campaign knowledge, not on thin air.

7. **Promote.** Once it's good, Mara files a promotion; a Builder shares it (to Domain) so the whole domain can *run* it (but not edit it).

Two runtimes, one governed plane. A system picks **LangGraph** (the default — structured, replayable, human-in-the-loop) or the autonomous **Hermes** runtime for long-running work that compounds (persistent memory + self-improving skills). Both share one governed plane: Hermes reaches models **only** through LiteLLM and tools **only** through the same Platform MCP, so OPA still gates every call, Langfuse traces it, and code runs in a kernel-isolated sandbox (Kata microVM or gVisor — never host-local). Hermes ships **off by default**.

5.4 Golden path 4 — Big Bets & Strategy: tying it to value

Finally, the work connects to the plan. In **Strategy**, an Administrator defines a pillar — *"Marketing efficiency"* — with a value metric (say, EBIT, or a custom "CAC reduction %"). In **Big Bets**, Ben creates a *"Campaign budget optimization"* bet under that pillar, sets a target and a go-live date, and attaches the real artifacts built above — the Gold dataset, the metric, the agent team, the app. Status derives **live** from each artifact's real lifecycle, so the roadmap flags itself on-track or at-risk without anyone updating a spreadsheet. The pillar's value can be tracked as a governed Cube metric or entered monthly — either way it feeds the pillar's history chart.

6 The governance model — the honest details

Governance here is not a policy PDF; it's executable, and it's the same for a click and for an agent. Four guarantees hold throughout.

6.1 Run-as-user, always

The per-user token (UI session or MCP token) carries your identity. The **role floor is re-checked from the live session on every call** — never trusted from the request body. An agent runs as its owner; Ask the OS runs as you. Nobody and nothing gets a privileged path.

6.2 OPA tool gates

A principal may invoke a tool only if **granted**; unknown principals and ungranted tools are denied. Internet access is an explicit grant — `web_fetch` is ungranted by default, and the web is returned as **sanitized data, never instructions**. Policy checks **fail closed**: if OPA is unreachable, the gate *denies* (with an explicit `opa-unreachable` marker) rather than waving the call through.

6.3 DLS — row & column security, independent of tier

Document- and row-level security filter what you see **at query time, regardless of tier**. Promoting an artifact to a wider scope **never** widens row access — two viewers of the same Domain-scope dataset see different rows. Every data-proxy route requires a session and scopes results to the caller's domains.

6.4 Two policy layers, one inbox

- **Tenant guardrails** an Administrator sets and domains cannot override: default-deny egress, no plaintext secrets to agents, no cross-domain data without a grant, a model allowlist.
- **Domain policy** Builders set within them.

High-stakes actions don't fail silently — they queue as a **card** in the **Governance** inbox, where *approving is the action*: on approve the platform runs the effect (an Argo deploy, a policy grant, an egress allowlist entry, a promote, a queued run) and writes the hash-chained audit. Three planes stay deliberately separate and cross-link rather than duplicate: **Admin** *configures* the tenant, **Governance** *decides and records*, and **Monitoring** *observes the artifacts*.

6.5 Archive, delete, and version history

Every artifact carries the same lifecycle controls, and only its owner (or an in-domain Admin) may use them. **Archive** is a reversible soft-hide (a running agent is also stopped); **Delete** is available only on already-archived items, behind an explicit confirm; **Version history** snapshots the prior state on every meaningful edit, and *restore* itself snapshots the current state first — so you can always undo a restore. One reusable helper (`lib/core/versioning.ts`) gives every store identical behaviour, and history is mirrored to OpenSearch so it survives redeploys.

7 The architecture

The platform assembles ~two dozen open-source components you can reason about — and enable — in layers. On top sits the **OS UI**, the single front door; beside it, the **MCP servers**, the governed front door for AI.

```
flowchart TB
  subgraph front["Front doors — one governed path"]
    UI["OS UI (Next.js)<br/>server-side routes"]
    MCP["MCP servers<br/>/api/mcp + /api/mcp/&lt;tab&gt;"]
    EXT["Claude · ChatGPT · any<br/>Streamable-HTTP MCP client"]
    EXT --> MCP
  end

  UI --> SPINE
  MCP --> SPINE

  subgraph spine["The governed spine — lib/infra/governed.ts"]
    SPINE["authorize → act → trace"]
    OPA["OPA<br/>policy-as-code"]
    LF["Langfuse<br/>trace + cost"]
    SPINE --> OPA
    SPINE --> LF
  end

  SPINE --> L1 & L2 & L3 & CTX

  subgraph L1["L1 — Agent core"]
    LG["LangGraph agents"]
    LL["LiteLLM gateway<br/>(budget cap, allowlist)"]
    OS["OpenSearch<br/>(hybrid retrieval)"]
    LG --> LL
    LG --> OS
  end

  subgraph L2["L2 — Foundations"]
    CUBE["Cube (metrics)"]
    DBT["dbt · Dagster"]
    OM["OpenMetadata"]
    DOC["Docling · Haystack"]
  end

  subgraph L3["L3 — Self-service"]
    TRINO["central Trino"]
    ICE["Iceberg / Polaris / MinIO"]
    SUP["Superset (optional BI console)"]
    FORGE["Forgejo + Argo CD"]
    TRINO --> ICE
  end

  CTX["Context Assembler<br/>budget-aware prompt packing"]
  LL --> STK["STACKIT AI Model Serving<br/>(EU-sovereign models)"]
```

7.1 The layers

- **Layer 1 — Agent core.** The runtime: **LangGraph** agents calling **LiteLLM** (the one model + tool gateway, with per-key access control and cost caps), every action traced in **Langfuse**, retrieving over **OpenSearch** (hybrid vector + lexical — no separate vector DB).
- **Layer 2 — Foundations.** Turning raw data and knowledge into governed products: **OPA** (policy at the tool boundary), **Docling** (parsing), **Haystack** (RAG), **Dagster** (orchestration), **dbt** (transforms), **Cube** (the metrics layer), **OpenMetadata** (catalog + lineage).
- **Layer 3 — Self-service.** Query, visualize, ship: the **Iceberg** lakehouse (**Polaris** catalog, **MinIO** object storage) with **central Trino** as the *one* governed query engine, **dashboards rendered natively in the OS** (Apache ECharts on the Cube semantic layer; **Superset** remains an optional stand-alone console), and in-cluster **Forgejo + Argo CD** for software delivery (git → CI → GitOps).
- **Layer 4 — Science / ML.** Classic ML — **JupyterHub**, **MLflow**, **Featureform**, **KServe** — *opt-in and off by default* (heavier, GPU-oriented).
- **Security baseline** spans every layer: default-deny egress through a single proxy chokepoint, a governed `web_fetch`, OPA tool authorization, externalized secrets, hardened pods.

7.2 The lakehouse & semantic layer

An upload becomes a real **Iceberg** table in your per-user schema; Silver and Gold builds run one compiled CTAS each; everything is queried through **central Trino** under your identity, so there is exactly one governance boundary for data. **Polaris** holds the catalog metadata in a durable relational-JDBC metastore (so the warehouse registration survives restarts), and **MinIO** keeps the data files on a PVC. Above the lakehouse, **Cube** is the semantic layer: a promoted Gold dataset auto-registers as a queryable Cube model, and a `define_metric` call adds named measures — so "revenue" has one definition that BI, agents and the explorer all resolve identically. Cube picks up new and changed models **deterministically**: its `schemaVersion()` hashes every model file's name and bytes, so any add/edit/remove flips the hash and triggers a lazy, per-context recompile on the next query — replacing dev-mode's file-watcher, which never saw the model-sync sidecar's cross-container writes. A freshly promoted metric therefore resolves reliably, without a Cube restart.

7.3 Models & the gateway

Every model call goes through **LiteLLM** — the one gateway that enforces the allowlist, per-key spend caps, tracing and graceful back-pressure. Inference runs on **STACKIT AI Model Serving**, an EU-sovereign, pay-per-token, three-tier set that an Administrator configures in **Admin → Models & Providers** (a single live-sourced store; the three below are the helm defaults — the OS is admin-configurable, not hardcoded to any provider). Each model also carries **per-token prices (EUR per 1M input / output tokens)**, editable in the same Models & Providers screen; these drive the cost figures in Monitoring — a model with no price set shows **"—"** honestly rather than a fake €0:

ROLE	HELM-DEFAULT MODEL	LITELLM NAME
Reasoning / planning (the PLAN phase)	`Qwen3-VL-235B-A22B-Instruct-FP8`	`sovereign-reasoning`
Standard / worker (tool-calling, coding, chat)	`gpt-oss-20b`	`sovereign-default`
Embeddings (4096-dim)	`Qwen3-VL-Embedding-8B`	`sovereign-embed`

STACKIT usage draws on one shared **€250/week** budget; once exhausted the gateway returns a graceful HTTP 429 rather than failing hard, and resets weekly.

7.4 The Context Assembler

Agents that read real data hit real context limits. The **Context Assembler** (`lib/infra/context/`) is a budget-aware prompt builder: a per-model window registry (with reserved output, admin/env-overridable), tool-result **compaction** (row-sets → header + sample + "…N more"; long text → head/tail), and a greedy pinned-first pack that **guarantees the prompt never exceeds the model window**. It's wired into the single-agent harness, the multi-node graph handoff (each node hands on an assembled summary, not the full transcript), *and* Ask the OS — which is what lets an agent discover and query real tables without blowing a 200K window.

7.5 The MCP front door

The platform exposes itself as **governed MCP servers**, live end-to-end at `https://agentic.datamasterclass.com/api/mcp` — one cross-tab server plus per-tab servers at `/api/mcp/<tab>` (`software` , `data` , `knowledge` , `agents` , `files` , `metrics` , `dashboards` , `bigbets` , `science`), each shipping a token-minimal `CONTEXT.md` . Around **55 governed tools** ship — reads, writes, and read-back parity — so an external client can build the entire Data → Metrics → Agents flow above. Every tool delegates to the **same library function the UI calls**; promotion-class actions stay Builder/Admin-gated; and every failure returns a typed, model-readable `{ code, reason, hint }` so a client can self-correct.

7.6 Durability

Every user-facing in-process store mirrors to **OpenSearch** through one shared core (`lib/infra/os-mirror.ts`): write-through on change plus hydration on boot, so artifacts survive redeploys and node-rolls. On STACKIT a three-tier backup system (nightly Postgres dumps, nightly off-cluster Velero volume backups, and a pre-upgrade backup gate) protects the stores themselves — practiced with a restore-drill runbook, with honest documentation of what is *not* protected.

8 Quickstart — run it

8.1 Locally, in one command

```
# prereqs: docker (running), kind, helm, kubectl  ·  ~14 GB RAM / 6 CPU free
./install.sh            # press Enter through every prompt
```

Pressing **Enter** through every prompt gives the **fully self-contained** install: every backend runs inside the chart and a tiny local model answers model calls — nothing external, no API key. `install.sh` creates the `kind` cluster if needed, builds and loads the images, installs the chart, seeds the demos, and prints the front door and demo logins.

```
./install.sh --defaults     # non-interactive, all bundled (CI / quick)
./install.sh --uninstall    # remove the release (keeps the cluster)
```

8.2 Open the front door

```
kubectl -n agentic-os port-forward svc/os-ui 8080:3000   # → http://localhost:8080
```

Every surface calls the in-cluster backends through **server-side API routes**, so credentials never reach the browser. Locally there's no login; on a real deployment you sign in with your Ory identity. The stack's operational console is embedded at **Platform → Components** — there's no separate admin service to run.

8.3 Try the seeded demos

Four end-to-end demos ship seeded, so the system proves itself the moment it's up: **ask the RAG agent** (retrieve → generate → trace), **query the lakehouse** (the governed `query` tool over central Trino), **build a dashboard** (native ECharts panels on governed Cube metrics), and **ship software** (push → Forgejo CI builds an image → Argo CD redeploys). Each has a one-card launcher on **Home**.

8.4 Deploy to your cloud (STACKIT)

The same chart runs the full Layer 1–4 stack on a sovereign STACKIT cluster; switching a backend to a managed service is only a values choice. Provision an SKE cluster + Object Storage + a load balancer + DNS, bootstrap the in-cluster prerequisites, point the OS at managed backends in `values.stackit-managed.yaml`, then:

```
helm install agentic-os charts/sovereign-agentic-os -n agentic-os --create-namespace \
  -f values.stackit-managed.yaml -f values.generated.yaml
```

Full, verified steps live in the **Reference** below and in `docs/stackit-deployment-guide.md`. Rough cost: **€450–670/mo** for L1+L2 at typical sizing; scale the node pool to zero between sessions (storage + IP persist at ~€16–20/mo).

8.5 Connect from Claude or ChatGPT (MCP)

Point any Streamable-HTTP MCP client at `https://agentic.datamasterclass.com/api/mcp` (or open any tab's **"Connect your AI Tool via MCP"** button for a one-click import link). The server uses managed OAuth (client-id-metadata-document pattern): your client fetches the metadata, you approve once at the OS consent screen, and it receives a 180-day access token held in the client — never in the OS. From then on **every tool call runs as you** — role floor re-checked from the live session, OPA-authorized, DLS-scoped. First stop, always: `whoami` and `list_capabilities`.

9 How to contribute

The OS UI is where most contribution happens, and it's built to be joinable. One rule governs the whole layout: **everything is either a tab, infrastructure, or core.** Learn one tab and you can work on any tab, because every tab is shaped the same way.

9.1 The three layers

Dependency direction is strict and one-way: `<tab>` → `infra` → `core`.

- `lib/core/` — cross-cutting primitives (session, config, auth, scopes, lifecycle, versioning, the artifact model, nav). No tab logic, no external IO.
- `lib/infra/` — the governed spine + every external-service client. The *only* layer that talks to OPA, Trino, OpenSearch, LiteLLM, MinIO, Forgejo, k8s. `governed.ts` (authorize → queryRun → trace) is the spine every tab write goes through; `mcp/` is the MCP transport.
- `lib/<tab>/` — one module per OS tab, all with the same internal shape. A tab imports *down* into core + infra, and **never sideways** into another tab's internals — only through that tab's `index.ts`. (A tab reaching into another tab's internals is the one thing code review rejects.)

9.2 The tab-module contract

Every `lib/<tab>/` has the same files:

FILE	RESPONSIBILITY
`index.ts`	The tab's **public API** — the only thing other tabs / routes import.
`schema.ts`	The tab's types (artifact shape, tiers, visibility). Pure.
`store.ts`	The **governed adapter** — CRUD/list/promote/lifecycle, each through `infra/governed`.
`<feature>.ts`	Pure, unit-tested domain logic. IO is injected so it stays testable.
`*.test.ts`	Co-located with the file it tests.
`README.md`	One screen: what the tab does, its golden path, its public API, its invariants.

Where to start. Read `os-ui/ARCHITECTURE.md` (the full contract) and `lib/connections/` (the reference tab-module: index / schema / store / README). Then pick a tab, copy the contract, and keep the invariant sacred: **all authz + trace live in `infra/governed` and each tab's `store.ts` — never scattered.** Consistency here *is* robustness: it's what makes the single governed path auditable.

See `CONTRIBUTING.md`, `GOVERNANCE.md`, and `CLA.md` at the repo root for the project process.

10 Reference

10.1 Deploying to STACKIT — the verified path

Locally everything is self-contained. On **STACKIT** (or any cloud) the platform runs the full Layer 1–4 stack from the **same chart**; switching a backend to a managed service is a values choice (`values.stackit-managed.yaml`), and heavier layers (Science, Terminal) ship pinned and provisioned but off by default.

1. **Prerequisites.** A STACKIT organization + project in **EU01 / Deutschland Süd**, and a service-account key with provisioning roles (SKE + Object Storage + DNS), saved as `stackit/sa-key.json` (gitignored). **This key gates any live deploy** — you can build and validate the entire chart on local `kind` with no key.
2. **Provision managed resources** (Terraform preferred): an **SKE cluster** (CNI = Cilium), a node pool sized for the full stack, **Object Storage** buckets + S3 credentials, a **load balancer + public IP**, a **DNS zone**, and **Secrets Manager / KMS**.
3. **Bootstrap the in-cluster platform** before the OS chart: ingress-nginx + cert-manager, the SKE storage class, Cilium default-deny egress, the External Secrets Operator, CloudNativePG, Velero, and Argo CD.
4. **Point the OS at managed backends** in `values.stackit-managed.yaml`: object storage and Postgres to STACKIT, the LLM to STACKIT AI Model Serving (`llm.mode: external`), Trino → the Polaris REST catalog, plus ingress hostnames, the egress allowlist and per-domain quotas. You can mix freely — managed Postgres but bundled OpenSearch, for example.
5. **Deploy and verify:**

   ```
   helm install agentic-os charts/sovereign-agentic-os -n agentic-os --create-
           namespace \
     -f values.stackit-managed.yaml -f values.generated.yaml
   ```

   Point DNS at the load balancer, confirm the consoles, confirm the default-deny egress baseline is active, then create your first domains.

> **Recommended: single node.** The primary, verified STACKIT path is one node, single AZ, all backends self-contained (`docs/stackit-deployment-guide.md`). Managed-services mode and multi-node HA are currently known-blocked on SKE cross-node networking; single node sidesteps it.

10.2 First-run bootstrap

On a real deployment the platform starts **closed**. The secure first-run path:

1. **Claim the first administrator.** Identity is backed by **Ory**; the bootstrap creates exactly one tenant Administrator that **auto-verifies** — no email server required to start.
2. **(Optional) wire up email** — Microsoft Graph `sendMail` (recommended for M365) or an SMTP fallback; sender `support@datamasterclass.com`. With neither, the platform runs without email and later accounts are verified out of band.
3. **Create domains** (one per team/business area; toggle optional layers like Science).
4. **Invite real users** — by email (the email is the username), with a role per domain. The platform generates a one-time temporary password shared out of band; the invitee sets their own on first login.
5. **Set the guardrails** — model allowlist, egress allowlist, cost envelope — all compiling through the same OPA the tabs enforce.

> **Honest status on the live STACKIT tenant:** outbound mail is currently **not delivering** (provider port-25 block; relay + sender-domain DNS pending), so on that deployment accounts are verified out of band. Graph and SMTP transports work wherever those services are reachable.

10.3 Sizing & capacity

Three different resources get confused under one word, "size":

RESOURCE	THIS DEPLOY	HOLDS	HOW IT SCALES
Node RAM	128 GB (`m3i.16`)	Running pods (Trino heap, OpenSearch; inference is on STACKIT, off-node)	With **concurrency/workload**, not data. Ran ~2–4% here.
Node disk	**200 GB**	**Container images** (all L1–4 images ~40–60 GB + churn headroom)	**FIXED.** Does not grow with your dataset.
Data storage	Object storage + PVCs	The Iceberg lakehouse on object storage (→ TBs) + PVCs for OpenSearch, Postgres, ClickHouse, MLflow	**Independently**, with the dataset.

The one gotcha: **don't confuse node RAM (128 GB) with the node disk** (the small volume that fills). Real data never touches the node disk — it lives on separate, independently-scalable storage.

10.4 Security model — four guarantees

- **Default-deny egress.** NetworkPolicies deny outbound except DNS, intra-namespace, and the API server; only the allowlist-only, logging **egress proxy** may reach the internet. (On `kind` the app-layer chain OPA → proxy → `web_fetch` provides the guarantee; on STACKIT, Cilium enforces it with FQDN-aware allowlists.)
- **OPA tool authorization, least privilege.** A principal may invoke a tool only if granted; agents use a scoped virtual key with a spend cap, never the master key.

- **The web is data, not instructions.** The only path out is the governed `web_fetch` — OPA-authorized, routed through the egress proxy, returned as sanitized data, never auto-written into the knowledge base.
- **No real secret in git.** On STACKIT every secret lives in Secrets Manager / KMS, synced by the External Secrets Operator; the chart references secrets by name only. The local dev passwords below exist only under `profile: local`.

10.5 Components at a glance

LAYER	COMPONENTS
L1 — Agent core	LiteLLM (gateway → STACKIT three-tier set) • OpenSearch (retrieval) • Langfuse (tracing) • query-tool (Trino MCP) • system agents (Domain RAG • ML pipeline • Hermes runtime)
L2 — Foundations	OPA • Docling • Haystack • Dagster • dbt • Cube • OpenMetadata
Infra	Postgres (CloudNativePG) • ClickHouse • Valkey • MinIO (PVC-backed) • Polaris (durable JDBC metastore)
L3 — Self-service	central Trino • Superset (optional console) • Forgejo (sovereign git) • Argo CD • CI runner • OpenSearch Dashboards • Terminal
L4 — Science	JupyterHub • MLflow • Featureform • KServe (opt-in)
Security & platform	egress-proxy • web_fetch • WireGuard tunnel (optional) • OS UI (embedded Components console • same-origin tool proxy + Level-1 SSO • MCP servers)

10.6 Demo logins (profile `local` — throwaway, never reused on STACKIT)

CONSOLE	PORT-FORWARD (`KUBECTL -N AGENTIC-OS …`)	URL	LOGIN
OS UI	`port-forward svc/os-ui 8080:3000`	`http://localhost:8080`	—
Langfuse	`port-forward svc/agentic-os-langfuse-web 3000:3000`	`http://localhost:3000`	`admin@datamasterclass.com` / `langfuse-local-dev-admin`
Superset	`port-forward svc/agentic-os-superset 8088:8088`	`http://localhost:8088`	`admin` / `superset-admin-local-dev`
Forgejo	`port-forward svc/forgejo-http 3001:3000`	`http://localhost:3001`	`gitea_admin` / `forgejo-admin-local-dev`
Argo CD	`port-forward svc/argocd-server 8082:80`	`http://localhost:8082`	`admin` / secret `argocd-initial-admin-secret`
MinIO	`port-forward svc/minio 9001:9001`	`http://localhost:9001`	`agentic-os-local` / `agentic-os-local-secret`

(The full console table is in `docs/getting-started.md`.)

10.7 The live teaching cohort

The live STACKIT deployment doubles as the classroom for the **Agentic Leader Program**, and its setup is a worked example of the whole operating model. A cohort domain hosts the instructor as **Builder** plus the participants as **Creators** (each signs in with email as username), with a separate `test` domain for dry-runs. The **Northpeak Commerce campaign-optimization exercise** — the running example throughout this guide — is seeded at **Domain scope** through the platform's *own governed endpoints*: campaign datasets, knowledge documents, sample files, a ready-made Campaign Evaluation Agent, and a Campaign App. Because the materials are Domain-scoped, every participant can *use and run* them — but as Creators they cannot edit them or promote their own work without a Builder. The exercise teaches the promotion ladder by living inside it.

10.8 Troubleshooting

- `ImagePullBackOff` **on** `demo-app` **right after install** — expected; clears once the first CI run builds and bumps the image tag.
- **Out of memory / pods pending** — the local slice is RAM-bound; keep heavy components off or give the VM more RAM.
- **Agent answers look canned** — STACKIT AI Model Serving isn't wired (no `STACKIT_API_KEY`), so chat has no live model. Configure the key or point LiteLLM at any model — no agent change.
- `web_fetch` **returns 403 / 502** — 403 means OPA hasn't granted `web_fetch`; 502 means the domain isn't on the egress allowlist. Both are by design.
- **Do I have to use STACKIT?** No — any Kubernetes works; the chart is portable. STACKIT is the sovereign EU default. You can build and validate everything on `kind` with no cloud key.

10.9 Status — what's live, what's next

The governance spine — OPA, approvals, RLS, promote ladders, roles, audit, MCP (live end-to-end at `/api/mcp`), auth, Knowledge, and the physical Data pipeline (Ingest → Define → Harmonize → Validate → Publish, i.e. upload → Bronze → Silver → Gold, with real data-quality checks + freshness/volume/schema monitors at Validate, then publish-on-approval → Cube → Talk to your data) — is **fully live**. Every build tab now shares **one staged Builder Framework** (five numbered stages, honest gating, a per-stage assistant, Simple/Developer views, lifecycle in the header). Layers 1–3 are in place; **Science (Layer 4)** is an integrated model-as-a-service tab (Define → Train → Deploy → Predict → Monitor) wrapping a live KServe `predict` model, with the raw MLflow/Featureform/JupyterHub/KServe consoles as a Developer escape hatch. **Software** is now a *governed frontend over the OS API*: new apps scaffold from the **Sovereign standard app template** — a Vite/React SPA that boots in the OS design (`@sovereign-os/ui`), signs in through the OS session only (no local passwords), scopes records My / Domain, and calls back through the OS-client SDK (`@sovereign-os/app-sdk`) under the signed-in user's own security — with a **live streaming Build** (the plan first, then one honest line per action, warnings and retries visible, behind a persistent Build ▸ Preview ▸ Deploy status rail) driven from the **Epics & stories tree** (per-story status chips, one-click build targets, *"N of M stories built"*) — every built story **auto-wired into the app's navigation** (the section registry regenerates from the committed pages, so a story can never build-but-not-show) and a **failed build surfaced honestly** in Test and Publish (never a fake "complete" over a stale release). It shows real per-run file diffs, live preview, and a Builder-reviewed deploy that scans the live repo tree; apps build a real image in-cluster (Forgejo CI) or publish static, and deploy to a live per-app URL. **Dashboards** render **natively in the OS** — Apache ECharts on the governed Cube layer, every panel queried **as the viewer** under per-user row-level security — with Power BI / Tableau / Superset-console bridges over the Cube SQL API. A developer `sos` **CLI** (Phase 0) brings the same governed door to your own terminal. The OS UI is v1.0: every sidebar tab is a real, brand-themed surface with light/dark theming.

Connections federate the outside world through one governed door: the tab lists connections (All/My/Domain/Company, with app-generated MCP connections folded in), a **vendor-stack-**

grouped, searchable Supported Connectors gallery (Microsoft • Google • AWS • Databricks • Snowflake • Salesforce • Kajabi • Atlassian • Open source • Other), a **Custom Connector** for your own REST/GraphQL API or MCP server (one action creates the connection *and* files the egress request), and **Talk to Connectors**. The catalogue spans **operational databases** (PostgreSQL • MySQL • SQL Server • MongoDB via Trino), **code & DevOps** (GitHub), **docs & knowledge** (Notion • Atlassian), **Supabase**, **messaging & calendar** (Slack • Gmail • Google Calendar • Outlook • Teams — reads auto, sending approval-gated and never automatic), **cloud governance / ML** (Microsoft Entra • Purview • Azure AI Foundry • AWS SageMaker, read-only), data-ingest (Google Drive / OneDrive), the medallion **layer choice** on agent data grants, an admin-enabled **external-warehouse connector** (federate AWS Glue/Athena • Snowflake • BigQuery • Databricks/Delta, plus experimental Fabric/OneLake, through Trino — discover → register → import → **scheduled incremental sync** (full-refresh / append / merge as the dataset's owner, cursor + lookback, presets from every 15 minutes to weekly, auto-pause after repeated failures), no YAML — with PostgreSQL • MySQL • SQL Server surfaced as **sync-capable operational databases**, **Apache Kafka** as a streaming source (configured topics federate as governed tables; an append-only **per-partition offset cursor** lands messages in the lakehouse — no one-time import of an unbounded stream), and **Salesforce** as an API-based operational source (no Trino connector exists, so the sync pulls `SystemModstamp` slices over the REST API page-by-page and streams them into the lakehouse; deletes are not detected in v1 and every pull consumes API quota), and **Kajabi** as its SaaS peer over the public API (OAuth client-credentials from Settings → Public API; JSON:API pages stream into the lakehouse with honest per-resource cursors — purchases by `updated_at`, contacts/customers/orders by `created_at` new-records-only, the rest full-refresh only; deletes never detected)), **one-click Power BI** (a `.pbids` file into Cube's Postgres-wire SQL API, DirectQuery, as the per-domain `bi_<domain>` principal so per-domain RLS re-runs on every query — no embedded password), an **Apache Airflow** connector (governed `trigger_dag`/monitor), and **OpenMetadata** (read/discover of a customer's existing catalog as a Connection, DLS-clamped and **active-only** — archiving an OS dataset soft-deletes its catalog entity, unarchiving restores it). Secrets are write-only; setting up each connector's OAuth app / tokens is the operator's step (every connector ships an install guide). External connectors are off by default and validated against a live source with your own credentials.

Shipped as explicitly-labeled Phase-1 slices (their next phases need new infra or your cloud credentials): Science's guided-train + real training runtime, OpenMetadata scoped write-back, and true per-viewer Power-BI RLS (today's Power BI path enforces per-*domain* RLS via the `bi_<domain>` principal). The developer **`sos` CLI** is Phase 0 (login • whoami • datasets • query); its roadmap (typed REST, device-code auth, signed-binary distribution, git-through-policy) is in `cli/sos/ROADMAP.md`. The full, versioned history is in `CHANGELOG.md`.

Sovereign Agentic OS — built from permissively-licensed open source for EU data residency. The core is Apache-2.0; bundled components keep their own licenses. This guide is generated from the repository; to update it, edit `docs/Sovereign-Agentic-OS-Guide.md` and run `scripts/build-docs.sh`.