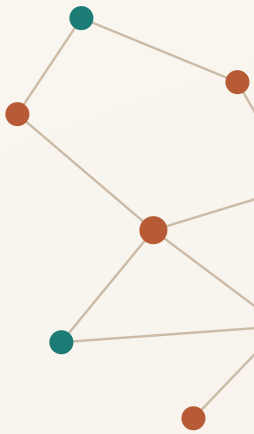
LEDGYX PLATFORM



Ledgyx

A serverless, event-driven platform for data, APIs, AI agents and pages.

Technical overview for integrators and technology providers — from the platform model down to the individual building blocks: the SQL engine, AI agents, file storage, HTML generation and on-chain subscriptions.

TECHNICAL OVERVIEW · NOVEMBER 2026

ABOUT THIS DOCUMENT

How to read it

The document goes **top-down**: first what the platform is and how it behaves as a whole, then each building block in more detail, then the cross-cutting concerns an integrator has to plan for.

1	Ledgyx in one page — serverless, event-driven	the model
2	Platform map — layers and runtimes	the architecture
3	Core concepts — the vocabulary	glossary
4	Ineron SQL engine — query in, finished JSON out	block 1
5	Entities — the data model without migrations	block 2
6	Events and the API Builder — a working REST API	block 3
7	AI agents and the AI Builder	block 4
8	File storage	block 5
9	HTML generation — pages rendered together with their data	block 6
10	Subscriptions — data, timers, files, chat, blockchain	block 7
11	Putting it together — a worked example	end to end
12	Languages — Arabic and English, natively	cross-cutting
13	Tenancy, security, versioning and metering	cross-cutting
14	Integration surface — for integrators and for providers	how to plug in
15	Design boundaries — what to plan around	honest limits
A	Quick reference	appendix

What this document claims. Everything described here exists in the running platform. Where the platform has a firm limit, it is stated in the text and collected in section 15, so you can design around it rather than discover it. Examples marked *illustrative* show the shape of a result, not a recorded response.

1 · THE MODEL

Ledgyx in one page

Ledgyx is a **serverless, event-driven platform**. You do not deploy servers or write a backend service. You describe a product — its data, its logic, its pages, its automations — and the platform runs it, scoped to your own workspace (a *configuration*).

Serverless

Business logic is not code you host. It is stored inside the platform as SQL handlers, templates and agent graphs, and executed on demand. There is no build, container or deploy step for your logic: saving an object makes it live, and the platform handles routing, execution, isolation and scaling.

Event-driven

The unit of work is an **event**: a named handler that runs when something happens. A REST call, a change to a record, a timer, a file landing in storage, a Telegram or WhatsApp message, a smart-contract log on Ethereum — each one is turned into an event. A handler reads and writes data, calls other services, starts an AI agent or renders a page. Its outcome changes shared state, and that change can trigger the next event.



Figure 1 — Every action updates shared state, and shared state can raise the next event.

Seven building blocks, one runtime

1 · SQL engine

Queries that return finished JSON, whatever their complexity, plus writes and entity definitions.

2 · Entities

Typed, linked data objects. Change the model live — there is no migration step.

3 · Events & API

SQL handlers published as REST endpoints with generated docs and a test console.

4 · AI agents

Visual agent graphs with skills, triggers, sandboxed code execution and access to every module.

5 · File storage

A per-workspace disk that is also the site root, the agents' workspace and an event source.

6 · HTML generation

Templates rendered together with the JSON the event returns.

7 · Subscriptions

React to data changes, schedules, file operations, chat messages and smart-contract events on Ethereum — and run any event, agent or webhook in response.

Because all seven share one data model, one event bus and one permission system, a finished product is mostly *configuration*: an API that works the moment you save it, pages that read the same data, agents that act on it, and automations that connect them.

2 · THE ARCHITECTURE

Platform map

From the outside there is one product with one API. Inside, a service layer sits on one transactional data core, and separate workers handle the work that should not run inside a request.

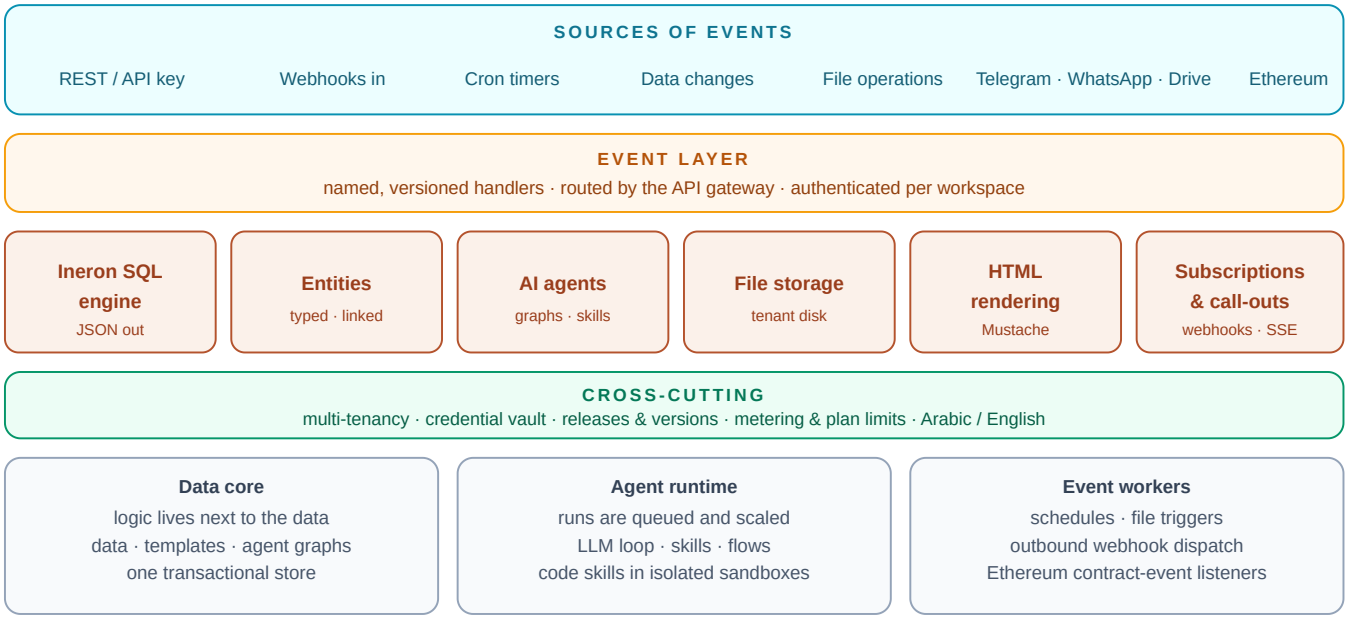


Figure 2 — Layers of the platform (top: what starts work; middle: what does it; bottom: where it runs).

Why this shape matters to an integrator

- **One database is the source of truth.** Data, templates, endpoint definitions and agent graphs live in one transactional store. A page, an API call and an agent all read the same state.
- **Request work and background work are separate.** A REST call is answered by the platform directly. Agent runs, scheduled jobs and blockchain listeners are executed by their own workers, so a slow LLM call or a long-running job never ties up your API.
- **Code you do not trust runs isolated.** Agent skills that execute Python or shell run inside disposable isolated sandboxes with no network by default (section 7).
- **You only meet the surface.** Integration is through REST, webhooks, the end-user API, an MCP endpoint, SSE and the chat/file/chain connectors listed in section 14.

3 · THE VOCABULARY

Core concepts

Term	What it is
Configuration	A tenant workspace. Every object — entities, events, endpoints, templates, agents, files — belongs to exactly one configuration. Typical use: one per client or per environment. API keys, disks, credentials and quotas are all scoped to it.
Entity	A typed data object (*Dictionary*, *Enum*, *Const*, *Operation*, *Contract* …) with user-defined fields. A field is a scalar or a reference to another entity.
Event	A named handler holding one Ineron SQL query per HTTP method (GET, POST, PUT, DELETE). The query result becomes the response. The same event can be started by a REST call, a subscription, a chat message or an agent.
Endpoint	A public URL `/{group}/{name}` bound to an event, with generated documentation and a test console.
Subscription	A listener that runs an event when something happens: a record changes, a timer fires, a file is added, a contract emits a log.
Agent	An LLM-driven worker defined as a node in a graph, with a system prompt, a model and a set of skills.
Skill	A tool an agent may call: run code, query data, read a file, send a message, call an HTTP API, start another agent.
Trigger	The entry point that starts an agent automatically — cron, webhook, Telegram, WhatsApp, Google Drive, file operation, manual input.
Template / Sitemap	A Mustache HTML template, and the route table that maps a URL to a template plus the event supplying its data.
Release	An immutable, named snapshot of a configuration's logic that can be made the default and rolled back instantly.
Vault	The credential store. Secrets are written by a human, applied by the platform and never shown to an LLM or an API client.

4 · BLOCK 1

The Ineron SQL engine

The foundation of the platform is a query engine that turns a SQL-style statement into **finished JSON**. You write the shape of the answer; you do not post-process rows.

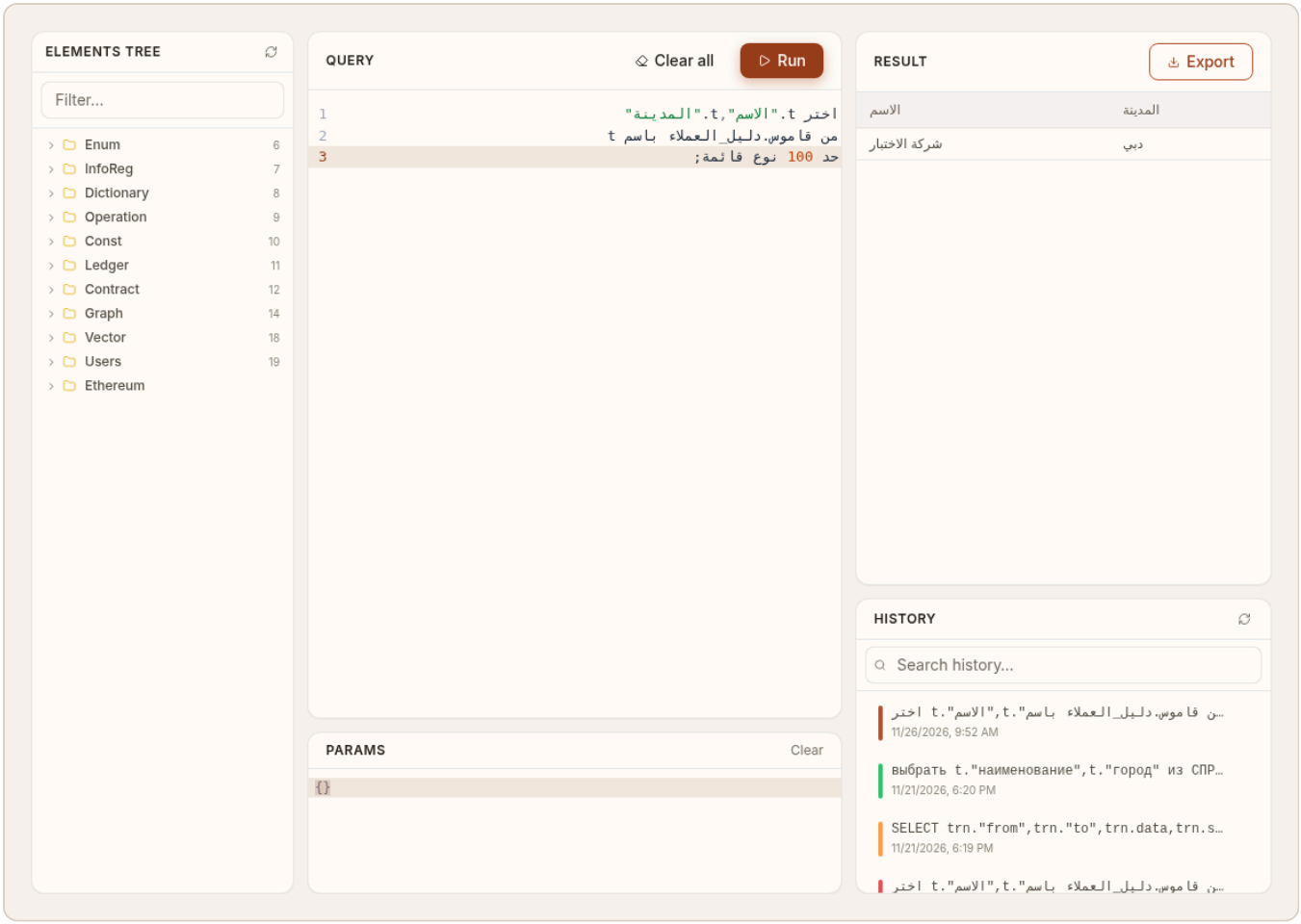
What it gives you

- **JSON is the native result.** A query ends with `TYPE OBJECT` (one JSON object) or `TYPE LIST` (an array). The HTTP response, the page data and the agent's tool result are exactly that JSON.
- **Nested results without a second request.** A sub-select inside the select list becomes a nested JSON array or object, so a parent and its children arrive in one response, shaped as the client needs them.
- **Linked data without a schema generation step.** Entities link to each other through reference fields, and a query follows them with an explicit join or a nested sub-select. It covers much of what you would reach for GraphQL to do, with ordinary SQL, and nothing to generate or keep in sync when the model changes.
- **One language for read, write and define.** `SELECT`, `INSERT`, `UPDATE`, `DELETE` and entity definition (`CREATE`, `ALTER`, `DROP`) are all Ineron SQL.
- **Parameters straight from the request.** `&name` reads a field of the incoming JSON; a typed cast (`&id::uuid`) is available where needed.
- **Safe by default.** Archived (soft-deleted) records are filtered out automatically; a `select allowed` variant applies access rules.

A query and its result

```
select h.name, h.city,
  (select r.number, r.price
     from Dictionary.room as r
    where r.hotel = &hotel::uuid
    type list) as rooms
from Dictionary.hotel as h
where h.ref() = &hotel::uuid
type object;
```

```
// illustrative result — one query, one nested JSON document
{ "name": "Palm Residence", "city": "Dubai",
  "rooms": [ { "number": "101", "price": 420.00 },
             { "number": "102", "price": 455.00 } ] }
```



The Playground: a query written in Arabic runs against the `main` configuration and returns a result table with Arabic column names. Query history is on the right of the full screen.

What SELECT supports

Feature	Available
Joins	`JOIN` / `INNER JOIN`, up to three tables and self-joins; the condition may compare fields of two tables, or a field with a literal or a parameter.
Sub-queries	In the select list (becomes nested JSON), in `FROM` (with its own `type list`), correlated `EXISTS` / `NOT EXISTS`, and scalar sub-queries in comparisons.
Aggregation	`count`, `sum`, `avg`, `min`, `max`; `GROUP BY` with several keys; `HAVING`.
Shaping	`ORDER BY` (several keys, `ASC` / `DESC`), `LIMIT`, `OFFSET` (number or parameter), `CASE` in the select list and as a comparison operand, `IN` / `NOT IN` over a list, `LIKE`, `IS [NOT] NULL`, arithmetic and `||`.
Functions	`substr`, `trim`, `concat`, `uuid`, `nullref`, `now`, `isnull` / `coalesce`, and `render(template, json)` — the Mustache engine from section 9, callable inside a query.
Input	`&name` parameters (also `&body.a.b`), and a JSON array as a recordset: `from &data as d(id int, name text)`.
Output	`TYPE LIST` / `TYPE OBJECT`; references to an Enum come back as `{"id","name"}`, references to a Dictionary as a UUID.

Boundaries of each feature are listed in section 15.

Writing data

```
-- POST event: the request body is the root, so &number reads {"number": ...}
insert into Dictionary.room(hotel, number, price)
values (&hotel::uuid, &number, &price);

-- Idempotent write by natural key: a repeat updates the same record
insert into Dictionary.goods(ref(), title, price)
values (uuid(&title), &title, &price);
```

Deterministic references (`uuid(...)` built from a natural key) make integrations safe to retry — a replayed webhook updates a record instead of duplicating it. Bulk loads take a JSON array as a recordset (`insert … select … from &data as d(id int, name text)`), with each element key named like the target column. Deletes are soft by default; `DELETE PERMANENTLY FROM …` removes rows for good. Writes are supported on Dictionary, Contract, Ledger and Operation entities.

> **Two cautions on writes.** `UPDATE` and `DELETE` without a `WHERE` clause apply to the *whole table* — there is no guard, so always supply the condition. And `ON CONFLICT … DO UPDATE` works only with `INSERT … SELECT` over a recordset; written after `VALUES` it is accepted but writes nothing — use the `ref()` / `uuid()` form above for idempotent single-row writes.

Defining the data model in the same language

```
CREATE DICTIONARY IF NOT EXISTS room (
    number text  field NOT NULL,
    price  numeric(12,2) field NOT NULL,
    hotel  DICTIONARY.hotel field NOT NULL   -- reference to another entity
);
CREATE ENUM IF NOT EXISTS room_status AS (free, booked, closed);

-- change the model later, in the same language
ALTER DICTIONARY.room ADD COLUMN note TYPE text(50);
ALTER DICTIONARY.room RENAME COLUMN number TO room_no;
DROP Dictionary.room;
```

Entities can be created, altered and dropped in SQL, from the Entities screen, or by the AI Builder, and they are usable in queries immediately; there is no migration cycle. Column types include `text` , `text(n)` , `numeric(p,s)` , `int` , `bigint` , `bool` , `date` / `timestamp` , `uuid` , `json` , `address` and references to other entities; attributes include `NOT NULL` , `DEFAULT` , `CHECK (… IN (…))` , `UNIQUE` and `PRIMARY KEY` . `ALTER` covers `ADD / DROP COLUMN` , `ALTER COLUMN` (type, null, default), `RENAME COLUMN` and, for Dictionaries, `RENAME TO` . `CREATE VECTOR TABLE` defines a vector store (section 5).

Calling the rest of the platform from SQL

A handler can reach other modules with `CALL` , so an event is also a workflow step:

Provider	What it does
`AI_AGENT`	Starts an AI agent asynchronously with the request as its input; `NEXT` names the event that receives the result.
`AI_CHAT_COMPLETION`	Calls an LLM directly and returns a response conforming to a named JSON schema.

`TG_SEND_MESSAGE` (`TG`), `TG_SEND_DOCUMENT`, `TG_GET_FILE`	Sends a Telegram message or document; downloads a Telegram file and optionally runs AI extraction on it.
`AI_TRANSCRIBE_AUDIO`	Transcribes an audio file.
`WEBHOOK`	Calls an external HTTP service.
`SSE_SEND`	Pushes a real-time update to browsers listening on a token.

```
select call(AI_AGENT "support_agent" FROM &body NEXT "support/reply" POST) type object;
```

`CALL` is an expression in the select list and ends with a `TYPE`; the quoted name is the configured agent, bot, credential or webhook. Calls are **asynchronous**: the statement returns `{"success":true}` once the job is queued, and the provider's answer reaches the event named by `NEXT`. `AI_AGENT` needs a `task` field in its input; `TG_SEND_MESSAGE` reads `message`, `chat_id` and `parse_mode`. Further providers (image generation, embeddings, moderation) are not available yet.

Language note. The query keywords exist in Arabic as well as English — see section 12. Boundaries of the dialect (for example, joins are inner joins) are collected in section 15.

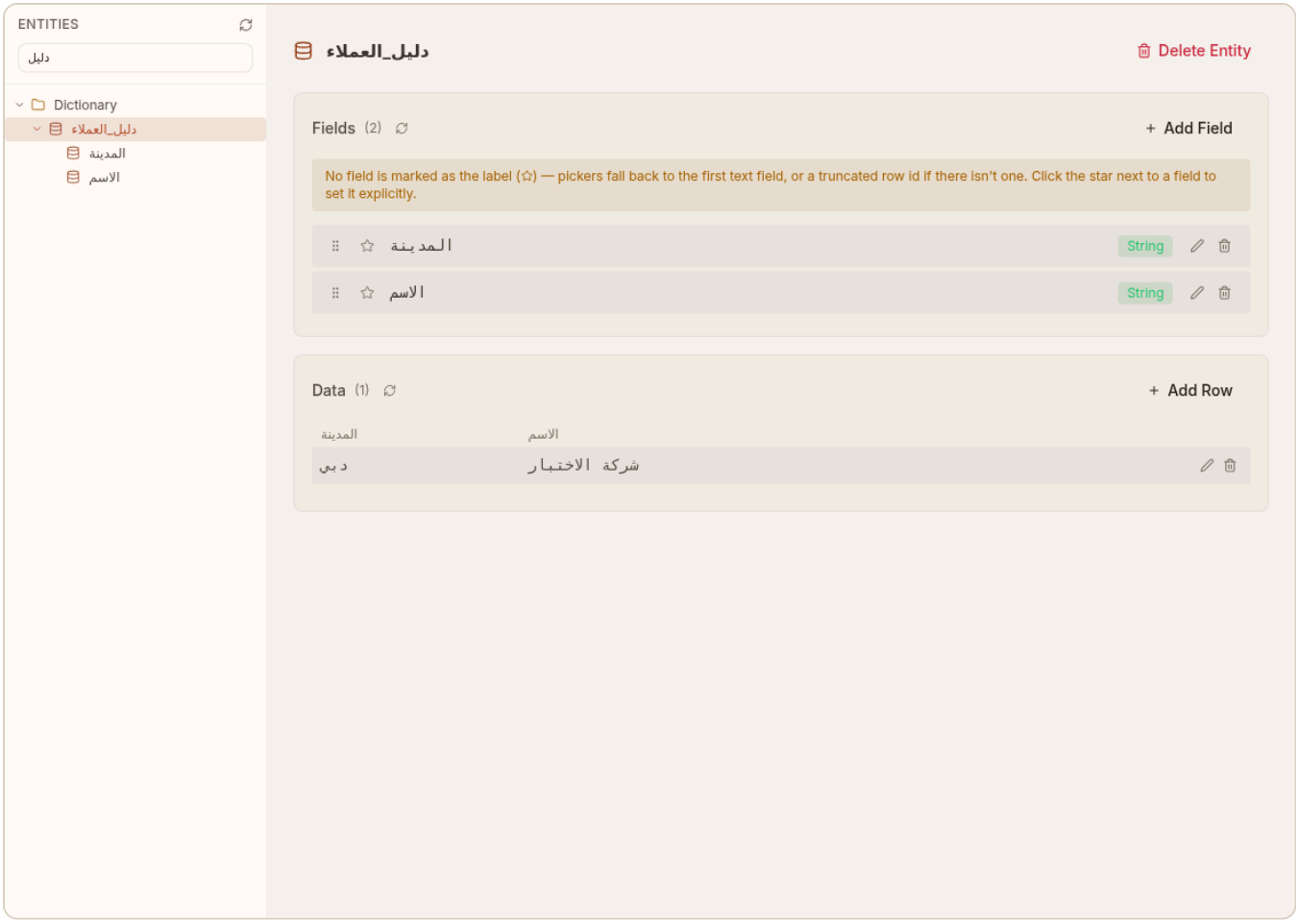
5 · BLOCK 2

Entities — the data model without migrations

Entities are the objects your product stores. They behave like typed tables, but the schema is data you edit live rather than a migration you ship.

Categories

Category	Use it for
Dictionary	Regular records and reference data — customers, products, rooms, tickets. The most common type.
Enum	A fixed list of named values — statuses, categories.
Const	A single named constant — a base URL, a rate, a flag.
Operation	Transactional records — orders, payments, bookings (`DOCUMENT` is a synonym).
Contract	A smart contract: ABI and address, whose events can be subscribed to (section 10).
Ledger, InfoReg	Register-style records — balances and information registers.
Graph	Relationship data, queried with graph patterns: `GRAPH.nodes MATCH (a:User)-[:follows]->(b:User) RETURN a.name LIMIT 3 TYPE LIST;` A graph can also be used as a sub-query. Graphs are created through the platform rather than by SQL `CREATE`.
Vector	Vector (semantic-search) storage: `CREATE VECTOR TABLE "items" (DIM 1536, METRIC COSINE)` and nearest-neighbour search: `select v.node_id from VECTOR "items" NEAREST &query_embedding LIMIT 5 as v type list;`



The Entities screen: an entity with Arabic name and Arabic field names, its typed fields (add, edit, delete from the toolbar) and its rows, which can be edited in place.

Fields

Scalar types

String · Number · Boolean · Date · JSON · Address (a blockchain or wallet address) · Data (binary).

References

Any entity can be the type of a field in another entity. That is how relationships are modelled; queries follow them with joins and sub-selects.

Money fields need a scale. A *Number* field stores whole numbers unless you give it a scale (0–3 decimal places). Without one a price of 19.99 is stored as 20. Declare the scale when the field is created — for currency use 2 — because the rounding happens at write time and cannot be undone afterwards.

Properties worth knowing

- **Every record has a stable identity**, read with `ref()` — an immutable UUID. Parent and owner references are available as `parent_ref()` and `owner_ref()`.
- **Deletes are soft by default.** A deleted record is archived and filtered out of queries automatically; `delete PERMANENTLY` removes it for good.
- **The model is live.** A new entity or field is available to queries, endpoints and agents as soon as it is saved. Dictionary rows can also be edited directly in the interface.
- **Package-installed entities are locked** against accidental edits (section 14).

- **Names can be in Arabic.** Entity and field names are not limited to Latin script; see section 12.

How this differs from a conventional backend

	Conventional stack	Ledgyx
Change a field	Write a migration, deploy, update the API layer and the client types	Edit the field; queries and endpoints see it at once
Shape the response	Resolvers or serializers per endpoint (or a generated GraphQL schema)	Write the query; the response is its result
Add a relationship	Foreign key, ORM mapping, API change	Add a reference field; join or sub-select it

6 · BLOCK 3

Events and the API Builder

An event is where logic lives. The API Builder turns events into a documented REST API. Together they replace the service layer you would otherwise write.

Events

An event has a stable identifier such as `rooms/list` and up to four handlers — GET, POST, PUT and DELETE — each one Ineron SQL query. For a GET the query string is the parameter root; for POST, PUT and DELETE the JSON body is. Saving a changed event creates a new version and archives the previous one, so history is kept.

The API Builder

- **Tree of groups and endpoints.** A group is a URL prefix; an endpoint binds `/{group}/{name}` to an event.
- **Generated reference.** Every group and endpoint is listed with its methods, URL and description in a Swagger-style view; curl examples written in the description appear verbatim.
- **Built-in test console.** Send real requests with headers, query parameters or a JSON body; see the status, the timing and the response. Requests are proxied through the platform, so there is no CORS friction and no mock.

Calling the API

```
# key in the path
curl https://app.ledgyx.com/rest/v2/{api-key}/rooms/list

# key in a header — keeps secrets out of URLs, logs and browser history
curl https://app.ledgyx.com/api/rooms/list  -H "X-API-Key: {api-key}"

# write
curl -X POST https://app.ledgyx.com/api/rooms/create \
     -H "X-API-Key: {api-key}" -H "Content-Type: application/json" \
     -d '{"hotel":"<uuid>","number":"103","price":470.00}'
```

Both URL forms reach the same endpoint and return identical responses. A missing or invalid key on the header form returns HTTP 401 with `{"success": false, "message": "…"}`.

Pinning a version from the client

A client can choose which version of the logic it talks to, by header:

Header	Effect
`Ledgyx-Version: <release>`	Resolve every event against that named, immutable release.
`Ledgyx-Snapshot: <branch>`	Resolve against a draft branch for preview. `latest` selects the working version.
(no header)	The configuration's default release; if there is none, the latest saved versions.

This lets an integrator test a change against production data with a header, and lets a client stay on a known release while the next one is prepared.

Inbound webhooks

Any external system can start a flow with `POST /webhooks/{token}`. The token is unguessable and scoped to one event or agent trigger, the same pattern used by GitHub and Stripe webhooks. The JSON body becomes the event's input.

7 · BLOCK 4

AI agents and the AI Builder

Agents are first-class objects of the platform. They are defined as graphs, run on a managed runtime, and can reach every other module through skills — so an agent works on the same data, files and events as the rest of your product.

The agent graph

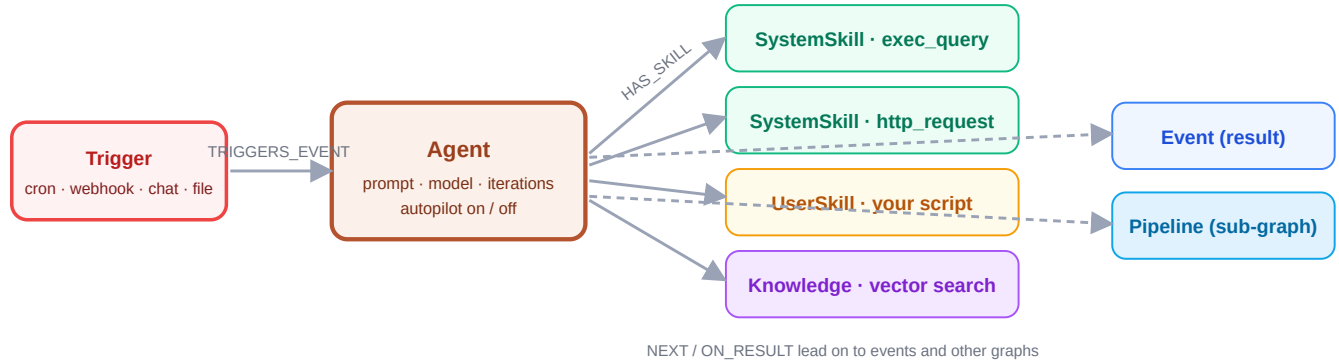
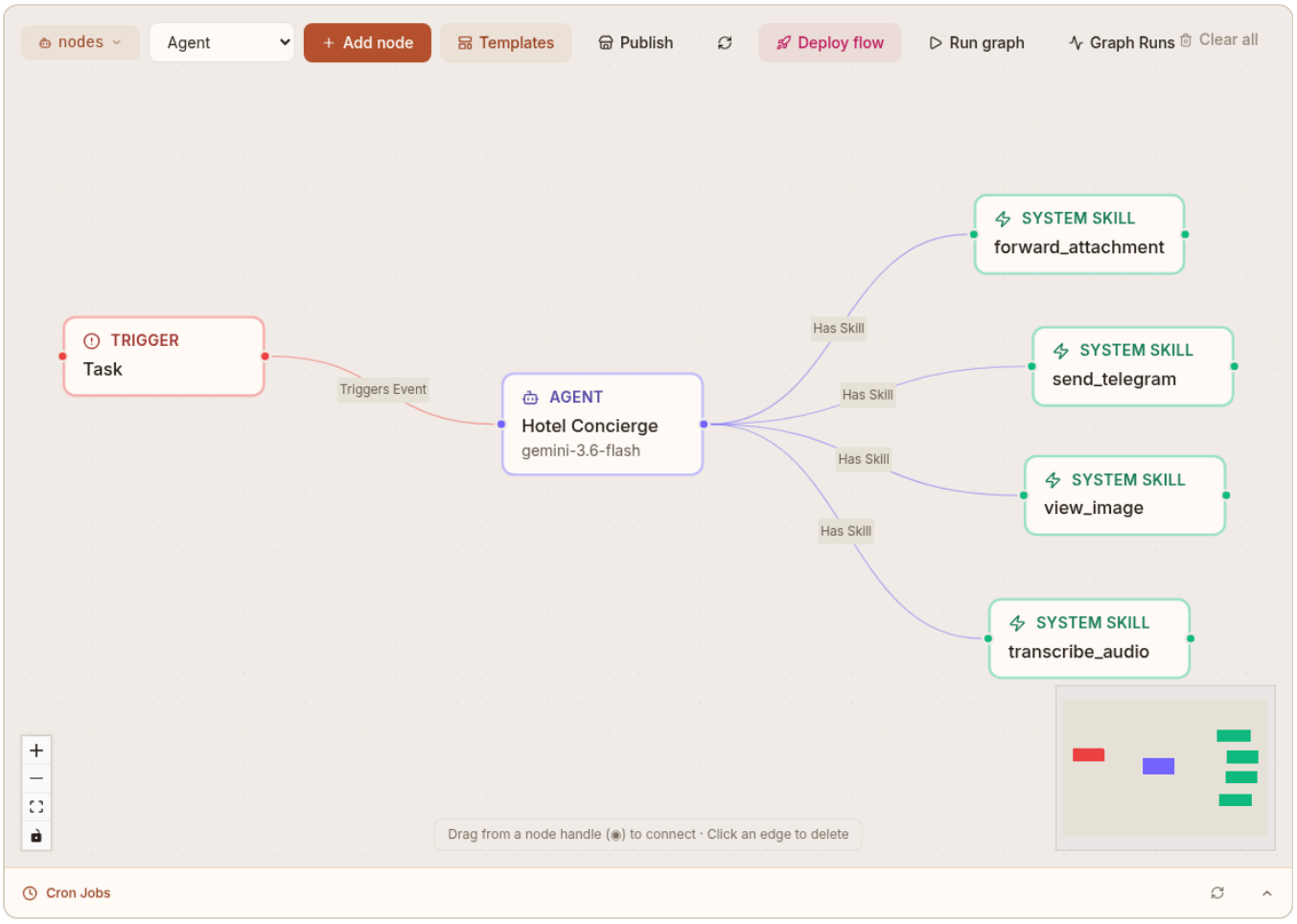


Figure 3 — Agents, skills, triggers and routing are nodes and edges on a visual canvas.



A real agent graph: a trigger starts the "Hotel Concierge" agent, which holds four skills (forward attachment, send Telegram, view image, transcribe audio). The toolbar deploys the graph as a flow, runs it and lists its runs.

You build agents by dragging nodes onto a canvas and connecting them, or by asking the AI Builder to do it. One configuration can hold several independently named graphs. An agent has a **name**, a **system prompt**, a **model**

(Anthropic Claude natively, plus OpenAI and any OpenAI-compatible endpoint — for example Google Gemini or a local Ollama — through the tenant's own credential and base URL), a **temperature**, a **maximum number of tool iterations** and an **autopilot** switch for runs that need no human confirmation.

What an agent can do — the skills

Area	Skills
Data	`exec_query` runs Ineron SQL against the workspace; `build_structured_event` turns the agent's conclusion into a structured event that your SQL handler stores; `ledger` for ledger-style records.
Files	`file_manager_get_text` (text, JSON, XML, YAML and PDFs with a text layer), `file_manager_view_image` (the agent sees the image), `file_manager_transcribe_audio`, `file_manager_save_text`.
Google Drive	`gdrive_list`, `gdrive_get_text`, `gdrive_create_text`, `gdrive_download`, `gdrive_upload`.
Messaging	`telegram_reply`, `whatsapp_reply`, `send_email`, `read_email` and `read_email_attachment`, `send_sse` for live browser updates, `webhook_send`.
External APIs	`http_request` — an authenticated HTTPS call with the secret attached by the platform (see "Credentials" below).
Code	`execute_python` and `execute_shell` in an isolated microVM with `/workspace` mapped to the tenant disk.
Knowledge	`knowledge_search` (semantic routing over Knowledge nodes) and `knowledge_add`.
Scheduling	`schedule_cron_job`, `list_cron_jobs`, `cancel_cron_job` — an agent can schedule its own follow-ups.
Orchestration	`agent_run` (call another agent), `agent_spawn`, `call_pipeline` (run an agent graph kept as a library), `agent_list`.

Beyond the built-ins, a **UserSkill** is a custom tool you define: a name, a description the model uses to choose it, a JSON-Schema input, and a script on your disk run by the Python or shell executor. A skill can also be described in a `SKILL.md` file.

Triggers — how agents start

Trigger	Starts the agent when…
`cron`	a schedule fires.
`webhook`	an external system POSTs to the trigger's URL.
`telegram`, `whatsapp`	a message arrives at the connected bot or number, including attachments.
`gdrive`	a file appears or changes in a watched Drive folder.
file operation	a file is added to a folder of the workspace disk.
`text_input`	you type a test message in the canvas.

Activating a trigger creates the platform event and registers whatever the outside world needs (a webhook URL, a Telegram webhook, a Drive watch). It is idempotent, so activating twice is safe.

Beyond one agent: flows

A whole graph can be deployed as a **flow** and run as one process, with deterministic control around the probabilistic model:

- **Condition** — branch on an expression evaluated against run state (a small expression language, no `eval`).
- **Loop** — repeat while a condition holds, bounded by a maximum iteration count.
- **Smart Switch** — choose one of several declared branches by *rules*, by an *LLM*, or *hybrid* (rules first, LLM only when none matches), with a confidence threshold and a fallback branch. The chosen branch is always checked against the declared ones.
- **Pipeline** — call an agent graph kept as a reusable library. Calls are depth-limited.

Flow deployment is opt-in: until a graph is deployed as a flow, triggers run single agents exactly as before.

Credentials: calling third-party APIs safely

Secrets never pass through a model. An administrator stores them once in the **vault**; an agent is given a credential by *name only*. When the agent calls `http_request`, the platform attaches the secret after the request is assembled and scrubs it from the response, the error text and the stored run log.

- Credential kinds: `http_bearer`, `http_basic`, `http_header`, `query_param` and `oauth2_service_account` (the platform signs the JWT, exchanges it for a token, caches it and re-issues it before expiry).
- Each agent lists the **exact hosts** a credential may be sent to; with a credential attached an empty list refuses every call.
- HTTPS only, redirects not followed, non-public addresses refused, DNS resolved once and the connection pinned, 30-second timeout.
- Responses are truncated by default at 32 KB (configurable up to 256 KB); the agent can ask for specific JSON paths so only those values enter its context.

Isolation and limits

- **Code runs in isolated micro-virtual machines.** Each execution gets its own VM. `/workspace` is the only persistent path and is the same disk shown in the File Manager.
- **No internet by default.** A VM starts with no outbound network. An administrator can allow specific domains per agent; traffic then leaves through an allowlist proxy and every connection is logged.
- **Quotas are enforced per plan:** agent call depth, maximum execution time, parallel VMs and monthly VM minutes. Agent steps and VM runtime are metered.
- **Every run is recorded** step by step — each model call, tool call and result, with token usage and duration — and visible on the Runs tab.
- **Agents are versioned**: versions can be pinned and rolled back, and agents are part of releases.

The AI Builder

The AI Builder is a chat assistant in the admin panel that builds *projects*, not just snippets. It reads the current configuration, then creates entities, events, endpoints, templates, sitemap routes, disk files, agents and triggers using the same tools a person would. For multi-object builds it follows a fixed protocol: inspect, design, **show a plan**, wait for confirmation, execute, then read back and verify. It never receives a secret — it can prepare an

empty vault entry and tell you where to fill it in. It answers in the language you write in and keeps per-workspace chat history.

From Telegram. The same chat can be driven from Telegram. Send `/start` to the Ledgyx bot to receive a one-time linking code (valid for five minutes) and enter it in Settings → Telegram Bot; from then on every text message in that chat reaches the AI Builder exactly as if it had been typed in the panel, in the configuration you have selected (`/config` switches it with buttons and offers only your company's configurations). The reply comes back as plain text, split into several messages when it is long. The plan-and-confirm protocol is unchanged: you confirm by answering in the chat. `/approve` lists the agent plans and MCP calls that are waiting for a decision as Approve / Reject buttons (a rejection takes your explanation as the next message), and notifications such as agent failures and plans ready for review are pushed to the chat. The bot is Ledgyx's own: you link a chat to your account rather than registering a bot of your own. Limits: text only (photos and files are ignored) and 20 messages per minute per chat; as in the panel, secrets are never typed into the chat. The bot's own service messages and buttons are in English, while the Builder's replies follow the language you write in.

Typical use for an integrator. Describe the customer's process in a paragraph; review the plan; let the builder create the entities, handlers, API and agent; then refine each object by hand in the same interface.

8 · BLOCK 5

File storage

Each configuration has its own disk. The same disk is the site's document root, the agents' workspace, a source of events and a place to exchange documents with people and other systems.

For people

The File Manager: folders, multi-file upload, drag-and-drop moves, image thumbnails, recent files, a recovery area and a trash with restore, and a storage meter against the plan quota.

For sites

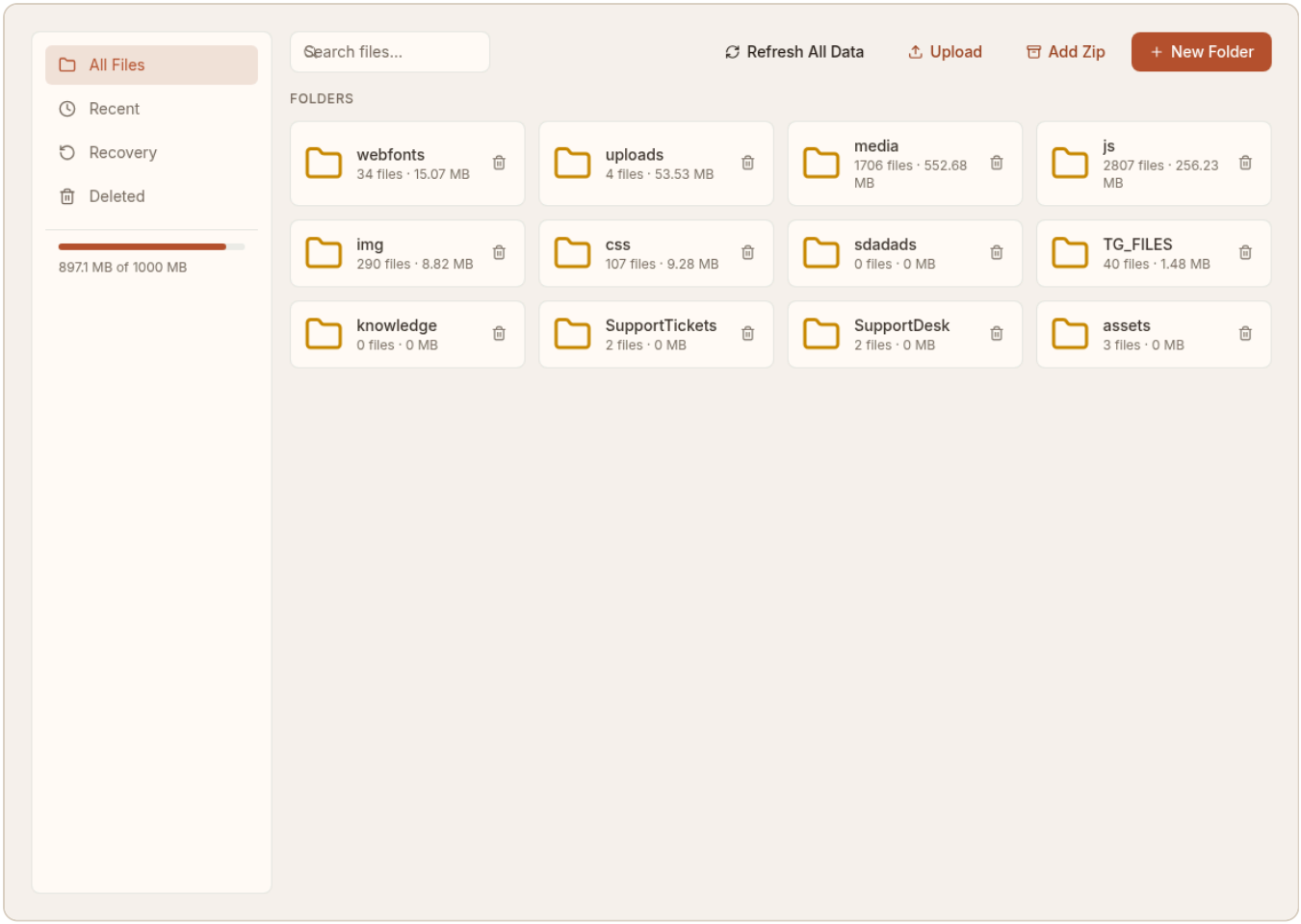
Static assets — CSS, JavaScript, images, JSON — are served from the disk, with the disk root as the document root: `css/style.css` is reachable at `/css/style.css`.

For agents

In code skills the disk is mounted at `/workspace`. File skills read text, PDFs, images and audio directly, and can save results back.

For events

A file operation can be a subscription source: a document dropped into a folder starts an agent or a SQL handler.



The File Manager of a configuration: folders with file counts and sizes, upload, zip import, search, recent, recovery and deleted views, and the storage meter.

Access control

File access is governed by rules: access rules can be created, listed and removed per folder, and a web policy decides what is publicly served. Deleted files go to a trash with a retention schedule rather than disappearing immediately.

Typical patterns

Pattern	How it works
Document intake	A file lands in `input/` (upload, Telegram attachment, Drive sync) → a file subscription fires → an agent reads the text or image → `build_structured_event` hands structured fields to a SQL handler → rows are stored; the original is moved to `processed/`.
Report generation	An agent runs Python in the VM, writes `output/report.csv` to `/workspace` → the file is immediately visible in the File Manager and downloadable.
Static site	CSS and JS are written to the disk; pages are rendered by templates (section 9) and reference them.
Drive bridge	A Google Drive folder trigger starts an agent for each new file; the agent can write its output back to Drive.

Paths inside agent code. Users speak of folders as they see them in the File Manager (`input` , `output/report.csv`); the runtime translates them to `/workspace/input` for code skills. Only `/workspace` is persistent — anything written elsewhere in the VM is discarded when it stops.

9 · BLOCK 6

HTML generation

A page on Ledgyx is not a separate front-end application that calls an API. The page is produced in the same step as its data: the event's JSON is fed straight into a template, and the HTML comes out of the platform.



Route, event, data and render happen in one step — no separate web application tier.

Figure 4 — The request path for a server-rendered page.

How it works

- **Templates** are written in Mustache — logic-less, so a designer can edit them safely. Variables, loops over lists, conditional and inverted sections, and partials (`{{> folder/name}}`) cover typical pages.
- **Layouts** are the outer shell (`<html>` , navigation, footer). One layout is the configuration's default; any route can choose another.
- **The sitemap** maps a URL path to a layout and an **event**. The route can be nested and can match dynamic segments with a regular expression.
- **The event's JSON is the template's data.** The same SQL that serves `/api/hotels/get` as JSON can serve `/hotels/palm` as HTML.
- **Rendering is done by the platform next to the data**, so a page needs no separate web application tier or rendering service.
- **The same engine is available inside any query** as `render(template, data)` — useful for composing an e-mail body, a message or an HTML fragment next to the data it uses: `select render('Hello {{name}}!', &data) as t type object;` returns `{"t":"Hello World!"}` for `{"data":{"name":"World"}}` . The data must be JSON (`&data` , `&body` , `&params` or a JSON literal); variables, escaped and raw output, lists and inverted sections work, while `{{> partial}}` includes work in the page pipeline but not through this function.

```
<!-- pages/hotel.html — receives the JSON from the event in section 4 -->
<h1>{{name}}, {{city}}</h1>
{{#rooms}}
  <div class="room">Room {{number}} — AED {{price}}</div>
{{/rooms}}
{{^rooms}}<p>No rooms available.</p>{{/rooms}}
```

Beyond hand-written templates

- **Page Builder.** A visual block editor that saves and publishes pages on the same pipeline, including ready-made blocks such as an end-user login form.

- **Live pages.** A page can hold a server-sent-events channel; an event that calls `SSE_SEND` (or an agent using `send_sse`) updates open pages in real time.
- **Your own domain.** Each configuration can be served on a custom domain with automatic SSL.
- **AI-generated sites.** The AI Builder can create layout, partials, CSS, JavaScript and sitemap routes from a brief.

One easy mistake. Partial references must be folder-qualified: `{{> pages/home}}` resolves, `{{> home}}` does not — and renders as the literal text `NULL` without raising an error. Check every include against the folders you created.

10 · BLOCK 7

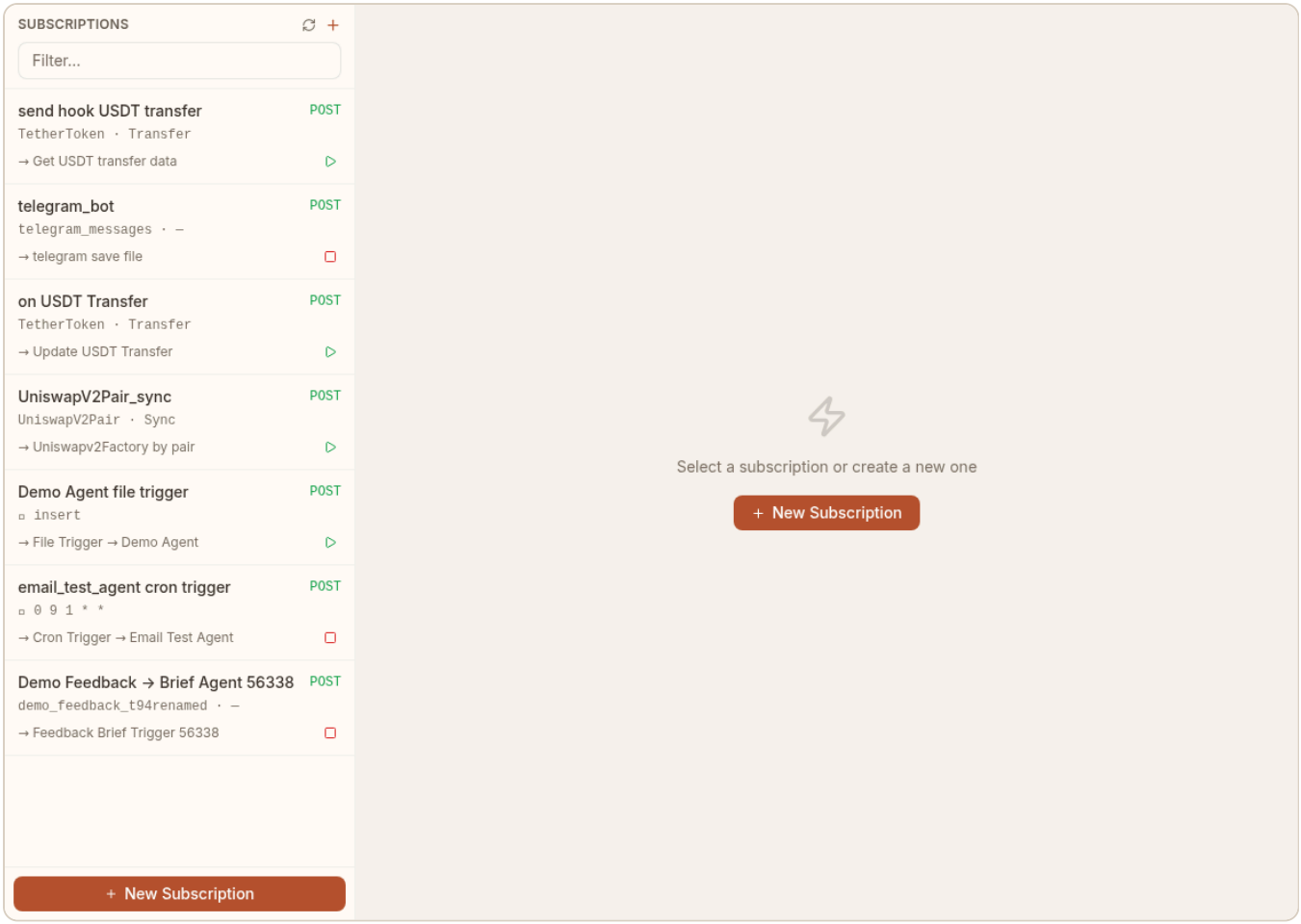
Subscriptions — reacting to the world

A subscription listens to an *emitter* and runs an event when it fires. This is what turns Ledgyx from an API host into an automation platform — including automation driven by what happens on a blockchain.

Source	Fires when…	Runs
Data change	a record of a Dictionary, Enum or Const entity is created, updated or deleted; timing is *before* or *after* the change.	an event
Schedule	a cron expression ticks. Agents can also own cron jobs.	an event or agent
File operation	a file is added to a watched folder of the workspace disk.	an event or agent
Chat	a Telegram or WhatsApp message arrives.	an agent
Google Drive	a file appears or changes in a watched folder.	an agent
Smart contract	a registered Ethereum contract emits a chosen event.	an event
External data	an external source posts a notification.	an event

Strictly, chat and Drive messages reach the platform as *agent triggers* (section 7) rather than subscriptions; they are listed because they answer the same need.

Each subscription can be started and stopped without deleting it, shows its status, and is independent per configuration. One emitter can have several subscriptions, each running a different event.



The Subscriptions screen of the `main` configuration: contract-event subscriptions (USDT *Transfer*, Uniswap V2 *Sync*), a file trigger that starts an agent, a cron trigger and a Telegram feed. Each row shows its emitter, the event it runs, and a start / stop control.

Ethereum events

The platform can watch a smart contract and turn its logs into events. You register the contract once — its **ABI**, its **address** and a node **endpoint** — then subscribe to a specific event from the ABI. A listener opens a live connection to the node; every matching log is delivered to your event as JSON, where SQL can store it, branch on it, notify someone or start an agent.

What you can build

- An indexer: store each `Transfer` in an entity and query it with ordinary SQL.
- Alerts: message a Telegram channel when a large transfer occurs.
- Reconciliation: match on-chain payments to orders and mark them paid.
- Agent workflows: let an agent investigate an unusual event and write a report to the disk.

What to know

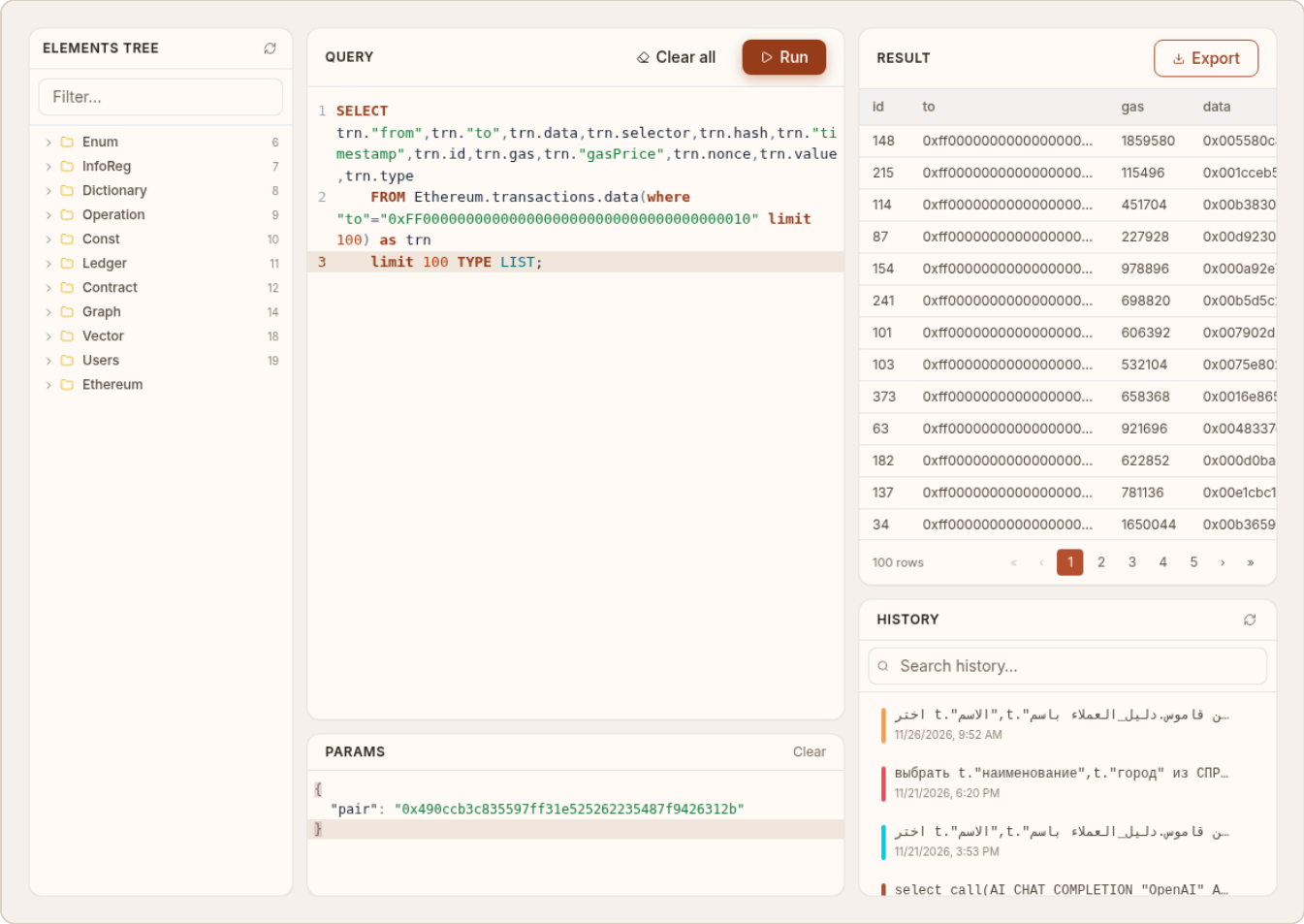
- LIVE Ethereum is the supported network today.
- Subscriptions are to **contract events** you declare through an ABI.
- Wallet addresses have their own field type, *Address*.
- The node endpoint is part of the contract definition, so you choose the provider.

Reading the chain with SQL

Subscriptions push events to you as they happen. The platform also lets you **query Ethereum data directly in Ineron SQL**: the *Ethereum* category sits in the entity tree next to your own entities, and its transactions can be

filtered and paged like any table. Chain data and your business data are then one query away from each other — for example, you can match transfers to the orders stored in a Dictionary.

```
SELECT trn.hash, trn."from", trn."to", trn.value, trn.gas, trn."timestamp"
FROM Ethereum.transactions.data(where "to"="0xFF0000000000000000000000000000000000000010" limit 100) as
trn
limit 100 TYPE LIST;
```



The same kind of query in the Playground: 100 transactions returned from `Ethereum.transactions.data`, paged in the result panel, with the query kept in history.

Besides `transactions.data(…)` there is `transactions.log(…)` for event logs. Inside the parentheses the clauses come in the order `WHERE`, `ORDER BY`, `LIMIT`, `OFFSET`; the limit defaults to 1000, and only the first `ORDER BY` field reaches the node query. `parse(address, topics, data)` turns a raw log into decoded JSON fields, which a sub-query can read. These sources can be joined with your own entities and used as the source of an `INSERT … SELECT`; they are read-only, and `select *` is not available on them.

Contracts you register are entities of the *Contract* category (`CREATE CONTRACT`), queried and written like a Dictionary, so the history of a contract — for instance a Uniswap V2 factory's pairs — can be stored and joined with ordinary SQL.

Cascades

The handler a subscription runs is an ordinary event, so it can write data (raising further data-change subscriptions), call an agent, send a message, or push to a browser. A chain such as *contract log → record → notification → agent review → report file* is configuration, not code.

```
-- exec event of a Post-timing subscription on Dictionary.order: alert the team
select call(TG_SEND_MESSAGE "ops_bot" FROM (select 'New order received' as message, &chat_id as
chat_id)) type object;
```

11 · END TO END

Putting it together — a worked example

A hotel-booking back end with a public site and a Telegram concierge. Each step names the block it uses; every step is a saved object, and none involves deploying code.

#	Step	What you create	Block
1	**Model the data**	`Dictionary.hotel` (name, city) and `Dictionary.room` (number, price, *hotel* reference). One statement each.	SQL · Entities
2	**Write the handlers**	Events `rooms/list` (GET), `rooms/create` (POST), `hotel/get` (GET with a nested room list).	SQL · Events
3	**Publish the API**	Group `rooms` with two endpoints; the reference and test console appear automatically.	API Builder
4	**Add the public site**	A layout, a `hotel.html` partial, a CSS file on the disk and a sitemap route `/hotels` bound to `hotel/get`.	HTML · Files
5	**Add the concierge**	An agent with `exec_query` and `telegram_reply`, a Telegram trigger. A guest asks "what rooms are free?" and the agent queries the same entities.	Agents
6	**Automate the follow-ups**	A data-change subscription on bookings sends a confirmation message; a nightly cron subscription produces an occupancy report into the disk.	Subscriptions · Files
7	**Ship it**	Make a **release**; clients keep a stable version while you prepare the next.	Versions

What the integrator sees

```
# the same data, three surfaces — all backed by the same events
GET  /rest/v2/{key}/rooms/list          # JSON for your own app
GET  https://hotel.example.com/hotels   # HTML page rendered from the same data
Telegram: "what rooms are free?"         # an agent answering from the same entities
```

Or ask the AI Builder

```
Build a hotel room booking system with:
- Entities: Hotel (name, city, stars), Room (number, type, price, hotel_ref)
- CRUD events for rooms, REST API with /hotels and /rooms groups
- A public listing page at /rooms
- A Telegram agent that looks up a guest's booking
```

The builder replies with a plan listing every entity, event, endpoint, template, disk file, sitemap route and agent it intends to create. On your confirmation it creates them in dependency order and reads the result back to confirm the graph is what you intended.

Why this is fast to iterate. A change — a new field, another filter, a different page layout — touches one object and is live on save. If it is wrong, the previous version is one click back, and a release lets you decide when clients see it.

12 · CROSS-CUTTING

Languages — Arabic and English, natively

Language support in Ledgyx goes below the interface. The query language itself can be written in Arabic, and Arabic text is first-class in the data.

The platform

The full admin interface is available in English and Arabic, with right-to-left layout. Even the agent canvas mirrors for RTL.

The queries

Ineron SQL keywords exist in Arabic as well as English, so a query can be written entirely in Arabic.

The data

Values are stored as Unicode, and entity categories, entity names and field names can all be in Arabic — a query can address Arabic-named tables and fields.

The same query in two languages

```
select t."name", t."city"
from Dictionary.customer_directory as t
limit 100
type list;
```

```
اختر t."الاسم",t."المدينة"
من قاموس.دليل_العملاء باسم t
حد 100 نوع قائمة;
```

The Arabic query is taken from the platform's own Arabic console and runs as written. Even the entity category (قاموس = Dictionary), the entity name (دليل_العملاء = customer directory) and the field names (الاسم = name, المدينة = city) are Arabic. The English version is its direct equivalent.

English	العربية	English	العربية	English	العربية
SELECT	اختر	WHERE	حيث	INSERT	إدراج
FROM	من	ORDER	ترتيب	UPDATE	تحديث
AS	باسم	BY	حسب	DELETE	حذف
LIMIT	حد	GROUP	تجميع	CREATE	إنشاء
TYPE	نوع	JOIN	ضم	CALL	استدعاء
LIST	قائمة	AND / OR	و / أو	LIKE	مثل
OBJECT	كائن	IN	ضمن	NOT	ليس

A selection of the keyword set.

The rules, precisely

- **Arabic alongside English.** The query language has 127 keywords, each with an English and an Arabic spelling.
- **Mixing is allowed.** One parser reads both, so English and Arabic keywords may be mixed in a single query.
- **Case.** Latin keywords and names are case-insensitive; Arabic has no case.
- **Entity categories** have Arabic names: قاموس (Dictionary), تعداد (Enum), عملية (Operation), سجل (Ledger), عقد (Contract), متجه (Vector).
- **Arabic names are valid** for entities and fields, unquoted or in backticks, as in the example above. Such names are case-sensitive.
- **What stays in English:** functions (`count` , `sum` …), `CALL` provider names, HTTP verbs, network names, and the type names `INT` , `NUMERIC` , `UUID` , `BOOL` . The types `TEXT` , `DATE` , `DATETIME` , `TIME` , `TIMESTAMP` and `BINARY` are localised.

What this means for an integration team

- **Regional teams work in their own language** — administrators, analysts and implementers can read and write queries without translating them.
- **Customer-facing content is not an afterthought.** Pages, API responses and agent conversations carry Arabic text without special handling, and templates can render right-to-left layouts.
- **The AI Builder follows the user.** It answers in the language it is written to, and agents can converse in any language their model supports.
- **Script generators are localised.** The Playground generates query scripts in the interface language.

13 · CROSS-CUTTING

Tenancy, security, versioning and metering

Tenancy

A configuration is a hard boundary. Objects, files, credentials, API keys and quotas belong to one configuration, and a request is always resolved inside the one it targets. Typical integrator layouts: one configuration per client, or one per environment (development, staging, production) of the same client.

Security

Area	What the platform does
Authentication	Email and password, Google and GitHub sign-in for administrators; API keys (path or `X-API-Key` header) for the REST API; unguessable tokens for inbound webhooks; bearer tokens for the MCP endpoint and the end-user API.
Secrets	A vault holds third-party credentials, **encrypted at rest** (see below). Values are write-only for humans, never shown to the AI Builder or to agents, applied by the platform at call time and scrubbed from responses, errors and run logs. A credential can be bound to a single configuration.
Outbound calls	Host allowlists per credential, HTTPS only, redirects not followed, non-public addresses refused (SSRF protection), pinned DNS, timeouts.

Code execution	Isolated micro-virtual machines, no network by default, per-agent domain allowlist with logging, per-plan resource limits.
Data access	Archived records excluded automatically; a permission-aware query form; file access rules per folder; end-user roles and grants (below).
Audit	API call log, per-step agent run history, VM egress log, MCP call log, usage audit.

The credential vault

Everything an integration needs to authenticate — API keys, bearer tokens, basic-auth passwords, Google service-account keys, bot tokens, mailbox passwords, LLM provider keys — lives in one place, the **vault** (the *Security → Credentials* screen), instead of being scattered through configuration, prompts or code.

Encrypted at rest

Each secret is stored only as ciphertext (strong symmetric encryption). The database never holds the secret in clear text.

The key is not in the database

The master key is kept outside the database, so a database dump or backup contains ciphertext only. Key rotation, which re-encrypts every credential, is supported.

Write-only for people

A value is typed once into the vault. Screens and APIs report only whether a value is present — never the value. Non-secret details (header name, username, scopes) are kept separately and stay visible.

Decrypted only at the moment of use

The runtime decrypts a secret just long enough to attach it to an outgoing request; it never reaches an LLM, a prompt, a response, an error message or the stored run log. Deleting a credential also wipes its ciphertext.

- **Scoped.** A credential belongs to an account and can additionally be bound to a single configuration, in which case it works nowhere else.
- **Typed.** The kind says how the secret is applied — bearer, basic, header, query parameter, service-account OAuth, and kinds for e-mail (IMAP/SMTP), Telegram, WhatsApp, Google Drive and LLM providers.
- **Traceable.** Each credential records when it was created, last rotated and last used.
- **Referenced, not copied.** Agents, triggers and integrations point to a credential by reference, so rotating a key in the vault updates every place that uses it.
- **Guarded.** An agent can only be deployed against a credential that exists, has a value and is visible to its configuration; every authenticated outbound call is restricted to the hosts listed for it.

Flexible by design — you decide how the secret is sent

The vault does not assume one way of authenticating. Each credential has a *kind* and a small set of non-secret settings that you choose, so the platform can speak to almost any HTTP API exactly the way that API expects:

Kind	You configure	The platform sends
`http_header`	the **header name of your choice** — `X-API-Key`, `api-key`, `X-Auth-Token` … — and the secret value	`<your header>: <secret>`
`query_param`	the **parameter name of your choice** — `key`, `api_key` … — and the secret value	`?<your parameter>=<secret>`
`http_bearer`	the token	`Authorization: Bearer …`

`http_basic`	the user name; the password is the secret	`Authorization: Basic …`
`oauth2_service_account`	the full list of scopes, optionally a user to impersonate and a token URL (HTTPS)	a short-lived access token that the platform requests, caches and renews on its own

Because the choice is made per credential rather than per product, the same agent skill can call a payment gateway that wants a custom header, a mapping API that wants a query parameter and a Google API behind a service account — with each secret kept in the vault, and each one allowed to reach only its own hosts. Providers do not need to adapt to Ledgyx: describe how your API takes its key, and a customer configures the credential to match.

The end-user tier

Besides the admin and API-key surfaces, a configuration can expose an **end-user API** under `/app/…` for the people who use *your* product. It has its own sign-up and sign-in, tokens, roles and grants, and is deliberately separate from the administrator tier: an end-user token cannot reach admin functions. What an end user can read, write or run (entity rows, agent runs) is whatever the administrator has enabled and granted. Pages published by the Page Builder can carry a login form that talks to this tier from your own domain.

An MCP gateway for agents you do not control

Ledgyx exposes a JSON-RPC **MCP endpoint** that works as a firewall in front of tools. Each client has a bearer token and a default decision; each tool call is evaluated against policy rules and ends in *allow*, *deny* or *require approval*. Calls awaiting approval are queued until a human approves them, and every call is logged. External AI clients can therefore use platform tools under policy.

Versioning and releases

Latest

The working version. Every edit writes here, and each event keeps a numbered version history.

Branches (snapshots)

A named draft with staged changes. Review the diff, then *promote* everything to production at once, or delete it.

Releases

An immutable, named capture of the configuration's logic — events, endpoints, templates, sitemap and agents. One release is the default; switching the default is the rollback, and it is instant.

Clients can select a release or branch with a request header (section 6). A **test mode** isolates changes so you can exercise a new release without touching live data.

Packages and reuse

A configuration can be exported as a **package** — entities, events, endpoints, templates and agent graphs — and installed into other configurations. Installed elements are locked against casual edits, and a package can be updated to a newer version (for example 1.0.1 → 1.0.2). An install option skips elements whose names collide with ones the target already has. Packages can be listed in a catalog; listings go through a moderation queue. For an integrator this is the route from "one client's solution" to "a repeatable offering".

Moving and backing up a configuration

Besides versioned packages, a whole configuration can be exported to a file and imported into another configuration — a lightweight route for hand-over, cloning between environments and backup. Both formats are produced from the same canonical document; they differ in what they keep.

Format	Contains	Use it for
ZIP archive `config.zip`	The complete configuration *as it is* — **nothing is removed**: structure and settings, **table data** (rows of Dictionary, Contract and Enum entities), agents with their graphs and skills, cron jobs, and every account-specific reference and ID exactly as stored.	**Backup** and restore; copying a working system together with its data.
YAML `config.yaml`	Structure and settings only, **no table data**. Prepared for handing to someone else, so everything tied to your account is **deliberately removed**: references to your own credentials, Drive folder and channel IDs, Telegram chat IDs and trigger event IDs are removed, and e-mail addresses and folder IDs inside agent prompts become fill-in placeholders. An agent's LLM credential travels as a name and provider, which the importer resolves by name.	**Transfer** to another account or client; storing a readable definition in version control.

- **Import accepts ZIP, JSON or YAML** (up to 10 MB) into a chosen configuration and **merges by name** rather than overwriting blindly.
- **Preview first.** A dry run never writes anything: it reports what would be created or merged, which objects **collide** with existing ones (for example an agent graph that already has nodes is left alone), which credentials could not be resolved and must be reconnected in the vault, and what is skipped. You commit only after reading it.
- **Honest results.** The import reports exactly what was written and what failed; it does not claim success it cannot back up.

Secrets and exports. The export never reads the vault: **no stored secret value is ever written into a ZIP or a YAML**, and an agent's LLM credential appears only as a name and a provider. The platform itself never places keys into configuration objects — they live only in the vault and are referenced. A key would appear in an export only if its author wrote it by hand into their own content, such as a custom script, which is the author's responsibility; the way to avoid it is the one the platform is built around: keep keys in the vault and reference them. Because a **ZIP** is a verbatim backup (it also holds table data and trigger URLs), store it as you would a database backup.

Separately, the File Manager's *Add Zip* uploads a ZIP archive and extracts it onto the configuration's disk — the quick way to publish a set of site assets.

Metering and plans

Plans bound the resources a configuration can use: API requests per month, subscriptions, webhooks, disk size, agent call depth, maximum agent execution time, parallel VMs and monthly VM minutes. Usage is metered — API calls, agent steps, VM runtime and model tokens — and token packages can be purchased on top of a plan. Enterprise plans are negotiated individually. Check current limits on the pricing page; they are configuration, not code.

14 · HOW TO PLUG IN

Integration surface — for integrators and for providers

Two kinds of partner meet Ledgyx from different sides. **Integrators** build solutions for end customers on top of the platform. **Providers** own a service — an API, a data source, a payment or messaging system — that those solutions need to reach.

Every way in and out

Surface	Direction	Use it to…
REST API `/rest/v2/{key}/… · /api/…`	in	Read and write data, run any event, from your own app or service.
Inbound webhooks `/webhooks/{token}`	in	Let an external system start an event or agent.
Outbound webhooks	out	Push an event to your endpoint: the `WEBHOOK` call, the `webhook_send` skill, or webhooks bound to subscriptions.
HTTP skill	out	Let an agent call your API with a vault credential.
End-user API `/app/…`	in	Give your customers sign-in, roles and access to their own data and agent runs.
MCP endpoint `POST /mcp`	in	Let external AI clients use platform tools under allow / deny / approve policy.
Server-sent events	out	Push live updates to open pages and clients.
Chat channels	both	Telegram and WhatsApp as conversational front ends for agents.
Files & Drive	both	Exchange documents through the workspace disk or a Google Drive folder.
Email	both	Agents can read mail and attachments and send mail.
Ethereum	in	Turn contract events into platform events.

For integrators

- **Headless back end.** Model entities, write events, publish endpoints; your web or mobile app consumes the REST API. Nothing to host.
- **Full-stack in one place.** Add pages (templates or Page Builder), end-user sign-in and agents, all over the same data.
- **Multi-client delivery.** One configuration per client, with a package as your reusable base, releases for controlled roll-out and per-client quotas.
- **Process automation.** Intake documents, classify them with an agent, write structured results, notify people — the pattern in section 8.
- **AI Builder as an accelerator.** Use it to draft the first version of a customer's solution, then refine by hand.
- **Regional delivery.** Deliver administration and customer content in Arabic or English without separate localisation tooling for the platform itself.

For providers

- **Make your API available to agents.** Describe how your API wants its secret — bearer, basic, *any custom header name*, *any query-parameter name* or service-account OAuth — and a customer configures a matching credential and stores their key once in the vault. Agents call you with `http_request`; the key never reaches a model.
- **Design for an agent's context.** Responses over 256 KB cannot be narrowed after the fact, so support server-side field selection or paging. Smaller responses can be trimmed by the agent asking for specific JSON paths.
- **Push, don't only wait to be polled.** Send webhooks into `/webhooks/{token}` to start flows the moment something happens on your side.
- **Ship a skill.** A custom tool is a name, a description, a JSON-Schema input and a script; you can package it with entities and events as an installable unit.
- **Offer a package.** A complete vertical solution — data model, handlers, pages, agents — can be published to the catalog for others to install.
- **On-chain providers.** A contract becomes a platform event source once its ABI, address and node endpoint are registered.

Constraints a provider should know. Calls from agents are HTTPS-only, do not follow redirects, time out at 30 seconds and refuse non-public addresses. Each call is limited to the hosts the customer listed for that credential. Code-execution agents have no internet unless the customer allows your domain.

15 · PLAN AROUND THESE

Design boundaries

These are the places where the platform is deliberately or currently limited. They are measured on the running engine, not assumed. Knowing them up front is what keeps a project predictable.

Ineron SQL dialect

Area	Boundary	What to do
Joins	Inner joins only; `LEFT`, `RIGHT`, `OUTER` and `FULL` joins are refused with an error. Join conditions are simple field comparisons. A join of two stored tables combined with `ORDER BY` is not available.	Use a nested sub-select to attach children.
Related-field paths	A dotted path through a reference field (`r.hotel.name`) is *not* resolved: the tail is silently dropped and you receive the reference itself. Paths into a JSON recordset work to two levels.	Join explicitly: `inner join Dictionary.hotel as h on h.ref() = r.hotel`.
Sub-queries	`IN (select …)` and `NOT IN (select …)` do not work.	Use `EXISTS` or a join.
DISTINCT	Accepted and silently ignored. `count(distinct x)` is a syntax error.	De-duplicate in the application, or use `GROUP BY`.

UPDATE / DELETE	Without `WHERE` they apply to the whole table — there is no guard.	Always give a condition.
ON CONFLICT	Works only with `INSERT … SELECT` over a recordset; after `VALUES` it writes nothing.	Use the `ref()` / `uuid()` idempotent form.
DDL	Entity definition covers Dictionary, Contract, Enum, Const, Ledger and Operation (altering: Dictionary, Contract, Const). There are no separate index, view or trigger objects, and a vector table cannot be dropped in SQL.	Drop columns one statement at a time.
CALL	An expression in the select list, asynchronous; the providers are those listed in section 4.	See section 4.
Aggregates	`count`, `sum`, `avg`, `min`, `max` work over one table. `GROUP BY` is `TYPE LIST` only, on plain table fields; no `ORDER BY` beside it; no expressions around an aggregate.	Sort or post-process groups in your application.
Dates	No date arithmetic (`now() - interval`).	Compute the bound outside and pass it as a parameter.
Ranges	`BETWEEN` is not supported.	`x >= &from and x <= &to`.
Casts	Seven spellings: `::TEXT ::NUMBER ::BOOL ::BOOLEAN ::UUID ::HEX ::DATE`.	Prefer a bare parameter when the JSON type already matches.
Ordering	`ORDER BY` takes a table field (optionally cast, `substr` or `trim`), not an expression.	Sort expressions in the application.
Writes	No `RETURNING`. `INSERT` returns `{"success":true}`. `CASE` is refused in `VALUES` and `UPDATE SET`.	Decide the value first, pass it as a parameter.
Text functions	`substr`, `trim`, `concat`, `isnull` / `coalesce` exist; `upper`, `lower`, `length`, `replace` do not.	Normalise text on write or in the application.
Empty results	No match returns `{}`, not `[]`.	Treat `{}` as "nothing found". `count(*)` returns a real 0.
Sparse fields in WHERE	A filter on a stored field silently excludes rows that never had a value for it (`where x is null` will not find them). A change to ordinary left-join semantics is planned but not yet released.	Filter on always-filled fields where completeness matters.

Where the engine says *not supported yet*, that form is final for now; change the approach rather than the spelling.

Platform

Area	Boundary
Blockchain	Ethereum is the supported network. The platform subscribes to contract events declared through an ABI and can query Ethereum transactions directly in SQL.
Agent code execution	No outbound internet by default; per-agent domain allowlist set by a human. Only `/workspace` persists.
Agent HTTP skill	30 s timeout, HTTPS only for credentialed calls, no redirects, response cap 256 KB (default 32 KB).
Models	Agents use the tenant's own LLM credentials: Anthropic Claude natively, OpenAI and OpenAI-compatible endpoints (Gemini, Ollama and similar) by base URL. Model availability follows the provider.
Interface languages	English, Arabic.
Quotas	Set by plan: requests, subscriptions, webhooks, disk, agent depth, execution time, VMs, VM minutes.
Entity names	A Dictionary can be renamed with `ALTER … RENAME TO`; other categories keep their name for life. Names containing non-Latin characters are case-sensitive.

Next step. The fastest way to understand Ledgyx is to build something small in it: create a configuration, ask the AI Builder for a three-entity model with an API, and open the generated reference. For partnership and technical discussions, contact Evgenii Iliushin, Founder & CTO — Ineron Information Technology Co. LLC (eugene@ineron.ae).

APPENDIX A

Quick reference

Endpoints

`/rest/v2/{key}/{group}/{endpoint}`	
`/api/{group}/{endpoint}` + `X-API-Key`	
`POST /webhooks/{token}`	
`/app/…`	end-user API (bearer)
`POST /mcp`	JSON-RPC (bearer)

Request headers

`X-API-Key`	
`Ledgyx-Version: <release>`	
`Ledgyx-Snapshot: <branch|latest>`	

Result types

`TYPE OBJECT`	one JSON object
`TYPE LIST`	JSON array

Parameters

`&field`	request JSON field
`&a.b`	nested field
`&id::uuid`	cast
`&data`	array → recordset

CALL providers

`AI_AGENT`	run an agent
`AI_CHAT_COMPLETION`	LLM with JSON schema
`TG_SEND_MESSAGE`	Telegram message
`TG_GET_FILE`	Telegram file + AI
`WEBHOOK`	external HTTP
`SSE_SEND`	live update

Graph node types

Agent · SystemSkill · UserSkill
Trigger · Event · Knowledge
Condition · Loop · Smart Switch · Pipeline

Graph edges

`HAS_SKILL`	agent → skill
`TRIGGERS_EVENT`	trigger → agent
`NEXT` · `ON_RESULT`	flow
`TRUE` · `FALSE` · `CASE`	branching

Credential kinds

`http_bearer` · `http_basic` · `http_header`
`query_param` · `oauth2_service_account`

Field types

String · Number (with scale 0–3) · Boolean · Date · JSON · Address · Data · reference to any entity.

Entity categories

Dictionary · Enum · Const · Operation · Document · Ledger · InfoReg · Contract · Graph · Vector.

Mustache

`{{var}}` escaped · `{{{var}}}` raw · `{{#list}}…{{/list}}` loop or if · `{{^list}}…{{/list}}` if empty · `{{> folder/name}}` partial.