OPEN DATA · ARTIFICIAL INTELLIGENCE · PUBLIC INFRASTRUCTURE

TECHNICAL REPORT · TD-TR-2026-01

agridata-mcp

A local MCP server for conversational access to Tunisia's agricultural open data.

AUTHOR

Tarek Gasmi

PUBLISHED

June 2026

STATUS

Independently produced and published

TYPE

Technical report

ABSTRACT

agridata-mcp is a Model Context Protocol server that makes Tunisia's agricultural open data portal (catalog.agridata.tn) directly queryable by LLM clients running locally. It exposes eight tools over a live CKAN API proxy, backed by a two-layer schema registry and a semantic layer that bridges how users phrase questions and how the underlying tables are actually structured. This report documents the system as it ships today — a source-distributed, local stdio server — and gives the rationale behind its principal design decisions and the limitations of the running system.

LICENSE

CC BY 4.0

tanitdata — autonomous studio · Tunis, Tunisia
tanitdata.org · contact@tanitdata.org

1. Introduction

Tunisia's Ministry of Agriculture publishes agricultural open data through the portal at `catalog.agridata.tn`, a CKAN instance covering climate, crops, fisheries, water resources, livestock, prices, trade, and related domains. The data is public and released under the Licence Nationale de Données Publiques Ouvertes. In principle it is open; in practice it is hard to use.

Three frictions dominate. First, the catalog is **fragmented**: datasets are split across dozens of producing organizations (CRDAs, directorates, agencies) with no uniform schema, and the same concept appears under different column names in different resources. Second, the data is **poorly typed**: every field in the CKAN DataStore is stored as text, so even a simple numeric average requires a cast and a regex guard against non-numeric values. Third, the metadata is **inconsistent and multilingual**: dataset titles, field names, and categorical values are French-dominant with significant Arabic content, and a non-trivial fraction of resources carry encoding corruption ("mojibake") in their Arabic field names. The combined effect is that the portal is reachable by a determined data engineer but effectively closed to the policy analyst, journalist, or researcher who would otherwise be its natural user.

agridata-mcp addresses this by making the catalog queryable through an AI assistant. It is a server implementing the **Model Context Protocol (MCP)**, the open standard by which LLM clients call external tools. A user runs agridata-mcp locally and connects it to an MCP-compatible client — Claude Desktop, Cursor, or any other — and then asks questions in natural language: the LLM selects and chains its tools to discover datasets, run SQL against them, and return cited results. (The server is distributed as the `tanitdata` Python package, from the `agridata-mcp` repository; this report refers to the software as agridata-mcp throughout, and to tanitdata as the publishing studio.)

Distribution model and audience. This report documents agridata-mcp as it actually ships today: as **source code**, distributed for local use. A user clones the repository, runs `uv sync`, and launches the server as a local stdio process that their MCP client spawns. There is no hosted endpoint and no managed service; the server runs on the user's own machine against their own client. This is a deliberately modest distribution model, and it bounds the audience honestly: agridata-mcp in its current form serves a **technically capable user** who is comfortable cloning a repository and editing a client configuration file. It lowers the barrier to *querying* the data dramatically — no SQL, no CKAN API knowledge — but it does not (yet) lower the barrier to *running the tool*. We return to this in Future Work.

The contribution of this work is not the protocol, which is a standard, nor the idea of wrapping an API, which is routine. It is the **domain adaptation**: the specific engineering that turns an awkward, inconsistent, text-typed CKAN catalog into something an LLM can query correctly on the first or second attempt. The most original element of that adaptation is the semantic layer described in Section 3.2.

2. Background

2.1 The Model Context Protocol

The Model Context Protocol is an open standard for connecting LLM applications to external tools and data sources. A client (the LLM application) launches or connects to one or more servers, each of which advertises a set of *tools* — named, typed, documented functions the model may call. The protocol is transport-agnostic; the two transports relevant here are **stdio**, where the client spawns the server as a subprocess and communicates over standard input/output, and **streamable-HTTP**, where the server runs as an HTTP service. agridata-mcp defaults to stdio for local use and retains an HTTP transport for self-hosters.

MCP is treated here as settled infrastructure, not as a research contribution. What matters for this report is that the protocol gives us a clean boundary: agridata-mcp's job is to present a small, well-documented set of tools whose descriptions and return values are good enough that an LLM can use them correctly without bespoke prompting on the client side.

2.2 The catalog.agridata.tn open data landscape

The portal runs CKAN, a widely used open-data platform. Its API exposes datasets (logical groupings), resources (individual files or tables within a dataset), and a **DataStore** — a queryable database layer that, for DataStore-active resources, accepts SQL through the `datastore_search_sql` action. Not all resources are DataStore-active; some are downloadable files (CSV, XLSX) only. agridata-mcp's tool design follows directly from this distinction: SQL-queryable resources are served one way, file-only resources another.

Several portal characteristics shape the system and recur throughout this report: the all-text typing of DataStore columns; a restricted SQL dialect (notably, `CASE WHEN` is not whitelisted and some resources reject SQL entirely); a faceted-search path for the portal's 55 producing organizations (adopted because the `organization_list` action historically returned an incomplete subset); and a French/Arabic metadata convention with intermittent encoding corruption.

3. System Architecture

agridata-mcp is a Python package (`tanitdata`) built on the official MCP SDK's `FastMCP` framework. At its center is a single asynchronous CKAN client (`CKANClient`) that proxies the live portal API, rate-limited to one request per 0.3 seconds. Around this sit eight tools, a two-layer schema registry, and a semantic layer. Figure 1 shows the overall structure.

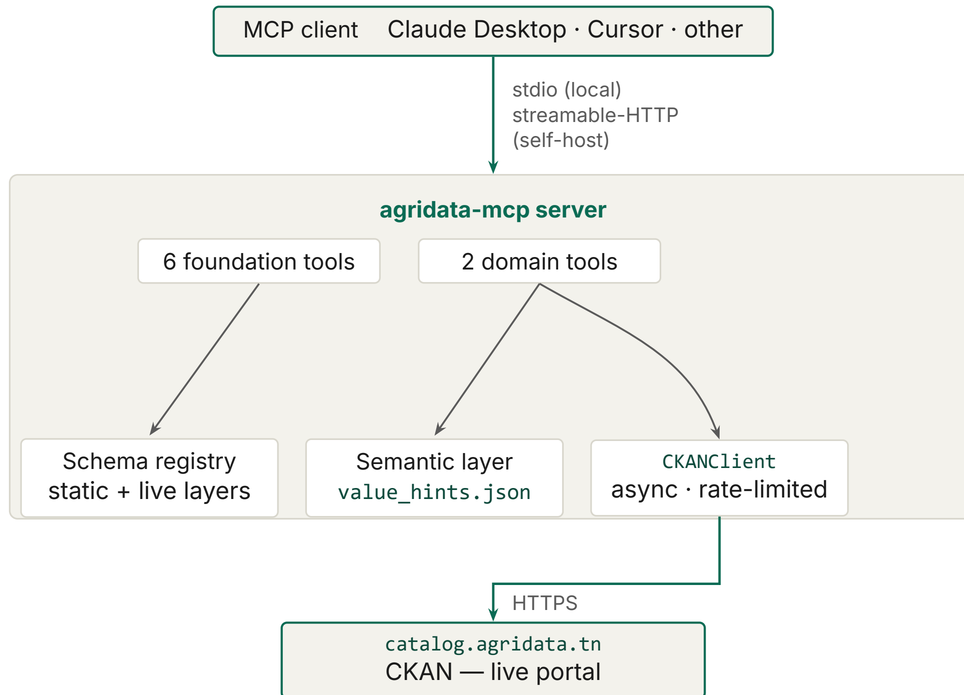

Figure 1. System architecture. An MCP client connects over stdio to the local agridata-mcp server, which hosts eight tools backed by a two-layer schema registry and a semantic layer. All tool data resolves through a single rate-limited `CKANClient` against the live portal; there is no local data store.

3.1 The tool suite

agridata-mcp exposes **eight tools**. The design is deliberately a **hybrid**: six general-purpose tools that let the LLM handle domain routing itself, plus two domain-specific tools retained where a bounded, single-schema problem justified specialized code. Table 1 summarizes them.

Tool	Kind	Purpose
search_datasets	foundation	Keyword / organization / group / format search over the catalog.
get_dataset_details	foundation	Full metadata and resource list for one dataset.
query_datastore	foundation	SQL (or simple browse) against any DataStore resource.
read_resource	foundation	Download and parse a non-DataStore CSV/XLSX file.
list_organizations	foundation	Producing organizations with dataset counts.
query_climate_stations	domain	Climate sensor and historical rainfall queries (EAV).
get_dashboard_link	foundation*	Map a topic to an interactive dashboard URL.
search_bibliography	domain	Search ONAGRI bibliographic catalogs.

Table 1. The eight tools. *get_dashboard_link is a pure local lookup over a fixed dashboard index; it makes no API call.

The foundation tools. search_datasets and get_dataset_details are the discovery path: find datasets by keyword or facet, then inspect a dataset's resources to see which are DataStore-active. query_datastore is the workhorse — it executes SQL against any DataStore resource and returns the schema plus records. It accepts either a resource UUID or a dataset slug; if given a slug, it **auto-resolves** to the first DataStore-active resource in that dataset, and if the caller's SQL referenced the slug as a table name, it rewrites the SQL with the resolved UUID. This auto-resolution exists to absorb the single most common LLM error — passing a human-readable dataset slug where the API expects an opaque resource UUID. read_resource handles the file-only resources: it downloads a CSV or XLSX on demand, parses it, and caches the result, returning data in the same shape as query_datastore. list_organizations reports producing organizations with dataset counts; it is implemented over faceted package_search, originally because the portal's organization_list action returned only an incomplete subset of the 55 organizations.

The two domain tools. query_climate_stations wraps Tunisia's weather monitoring network and historical rainfall records. It exists because the climate data is stored in an **entity-attribute-value (EAV)** layout across **three distinct schema variants** (differing in date, parameter, value, and unit column names), which the tool auto-detects; it also performs natural-language parameter aliasing (e.g. "température", "wind", "pluie" → canonical sensor groups), chooses the correct aggregation (precipitation sums, other sensors average), supports multi-station comparison, and caches per-station sensor lists for the session. None of this generalizes to other domains, and expecting the LLM to reconstruct it per query through the generic tool proved unreliable — hence a dedicated tool. search_bibliography searches ONAGRI's bibliographic catalogs (25,944 records across six catalogs) with a **tiered** strategy: the two large catalogs are queried by SQL, while four smaller thematic libraries that reject SQL (HTTP 409) are fetched via datastore_search

and filtered in Python, with results cached. It also reconstructs PDF download URLs from record fields.

The foundation-tool pattern. An early version of agridata-mcp moved toward proliferating one domain tool per data domain (a dedicated crops tool, etc.). That direction was reverted in favor of the foundation pattern, where generic tools — chiefly `query_datastore` — let the LLM perform domain routing and schema interpretation itself, guided by tool documentation and the semantic layer. The retained domain tools are the exceptions that earned their specialization. The evolution is recorded in the repository's tag history: tag `v1.2-crops` marks the domain-tool direction, and tag `v1.3-foundation-pivot` marks the reversal. Section 4.1 gives the rationale.

3.2 The semantic layer

The semantic layer is the most original part of the system and is treated here as a first-class architectural component, not a tuning detail.

The problem it solves. An LLM translating a user's question into SQL must guess two things it cannot see: the exact column names and the exact categorical values in the target resource. Left to guess, it writes plausible but wrong SQL. The canonical failure: a user asks about wheat, and the model writes `WHERE "culture" ILIKE '%blé%'` — but the resource has no `culture` column and no value `blé`; the actual data is in a **wide-format** column named `Ble_dur_qx` (durum wheat, in quintaux). The query returns nothing, and the model has no signal as to why.

The mechanism. agridata-mcp closes this gap at two points in the model's decision cycle (Figure 2):

- **Before the query — tool-description enrichment.** The `query_datastore` tool description carries compact, portal-specific guidance the model reads upfront: the unit-suffix dictionary (`_ha`=hectares, `_t`=tonnes, `_qx`=quintaux, `_mm`=millimètres, and so on), an explicit warning that many crop resources are wide-format (the crop name *is* the column — select it directly, do not filter on it), common crop-name prefixes, and a recovery strategy ("if unsure about exact columns or values, call with no SQL first to preview the schema and sample data").
- **After the query — response value hints.** A committed runtime artifact, `value_hints.json`, holds per-resource categorical values mined from the portal: **735 resources, 1,077 columns, 11,721 distinct values** (capped at 50 per column). When a query returns, agridata-mcp appends the exact known values for the result's categorical columns, so the model writes its follow-up `WHERE` clauses against real strings rather than guesses.

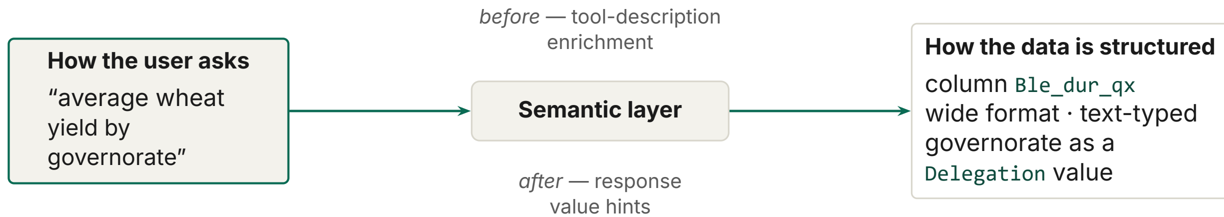

Figure 2. The semantic bridge. The layer sits between how a user phrases a question and how the data is actually structured: tool-description enrichment informs the model *before* it writes SQL; value hints correct it *after* the first query. It turns a guess into an exact-value query.

The layer is built offline by scripts in the public repository (`extract_vocabulary.py` → `extract_value_hints.py`) and loaded at startup by the schema registry. It requires **zero** additional API calls at query time — it is a local dictionary lookup. A small within-codebase ablation, holding the rest of the system fixed, found that adding the semantic-layer bundle raised the number of fully-correct answers on a ten-query set from four to six; we cite this only as evidence that the layer does what it was designed to do, and defer any broader evaluation to separate work (Section 7).

3.3 The schema registry

Tools need to know what resources exist, what columns they have, and how to attribute results — without paying a network round-trip on every call. The `SchemaRegistry` provides this through two layers.

The static layer is loaded once, synchronously, at startup from the committed `schemas.json` (a curated index of domains, schema clusters, Arabic field-decoding maps) and `value_hints.json`. It loads in roughly 30 milliseconds, is never mutated, and is sufficient to make every tool callable immediately.

The live layer is seeded from the static layer at startup and then kept current against the portal. On startup the server launches a **background refresh task** so that the first tool call is never blocked on the network. Thereafter, every tool invocation calls `maybe_refresh()`, which checks whether the refresh interval (**6 hours**) has elapsed; almost all calls return instantly after a timestamp comparison, and only the first call after the interval acquires a lock and refreshes (double-checked locking). A refresh fetches the live DataStore inventory via the `_table_metadata` pseudo-resource, skips aliases, and fetches field lists and record counts for any resources not already known, via paginated `datastore_search`. If the portal returns nothing, the registry keeps its existing inventory rather than discarding it.

The registry also computes governorate coverage and serves source attribution (resource → dataset → organization → portal URL) so that every tool response can carry provenance.

3.4 Data-quality handling as engineering features

Much of agridata-mcp's code exists to absorb the portal's data-quality problems so the LLM does not have to. These are shipped behaviors, not aspirations:

- **All-text typing.** Tools and tool documentation steer numeric and date work through explicit casts (`::numeric`, `::timestamp`) guarded by regex so non-numeric rows do not abort a query.
- **Arabic mojibake decoding.** A field-name decoding map repairs known encoding corruption (notably across a set of Bizerte price datasets) so the model and user see meaningful Arabic/French field names.
- **EAV detection.** The climate tool detects which of three entity-attribute-value schema variants a resource uses and adapts its SQL accordingly.
- **Excel-overflow exclusion.** Two known corrupted resources padded to the Excel row ceiling (1,048,575 rows) are recognized and excluded from query paths.
- **Format-mislabel handling.** `read_resource` dispatches its parser by sniffing file magic bytes rather than trusting the declared format, because a substantial number of resources are CSV/XLSX-mislabeled.

4. Design Decisions and Rationale

4.1 Why a foundation-tool pattern

The reverted domain-tool direction would have meant a growing roster of bespoke tools, each encoding one domain's schema assumptions. Two problems made this untenable. First, the portal has far more schema variation than domains — crop production alone spans many distinct schemas — so per-domain tools would still not have matched the real data, and per-schema tools would have numbered in the hundreds. Second, a large tool roster degrades LLM tool selection: the more near-synonymous tools the model must choose among, the more often it chooses wrong. The foundation pattern inverts the bet: keep the tool set small and generic, and invest instead in *information* — tool documentation and the semantic layer — that lets the model route itself. The two retained domain tools (`query_climate_stations`, `search_bibliography`) are kept because each wraps a single, bounded schema problem (EAV variants; SQL-blocked tiered catalogs) where the specialization pays for itself and does not multiply.

4.2 Why the semantic layer was necessary

The foundation pattern only works if the model can write correct SQL against resources it has never seen. Early use showed it could not: it guessed column names and values, and guessed wrong, in exactly the wheat/`Ble_dur_qx` shape described in Section 3.2. The semantic layer is the direct response — it is what makes a generic `query_datastore` tool usable on a catalog whose structure the model cannot infer from question text alone. This is why we treat it as the system's central contribution rather than a refinement.

4.3 The data layer: a reverted local snapshot

An earlier release embedded a **local snapshot data layer** — the catalog's DataStore tables converted to local columnar files, queried by an embedded SQL engine, with no network dependency — as resilience against upstream outages. This architecture is preserved on tag `v3.1.3`. It was **reverted** at `v3.2.0` in favor of always proxying the live portal. The trade was explicit: the snapshot bought outage resilience and reproducibility at the cost of data freshness and a great deal of build and maintenance complexity (a conversion pipeline, a second query engine, a parallel data path to keep correct). With the portal generally reachable again, freshness and simplicity won. The consequence — a hard dependency on the portal's availability — is stated plainly as the system's sharpest limitation in Section 5. The version history here is offered as local code-architecture evolution, anchored to the cited tags.

4.4 The authentication decision

agridata-mcp ships a Bearer-token authentication path — a Starlette middleware (`BearerAuthMiddleware`) backed by an API-key store — but it is **dormant by default**. It is wired only in the HTTP transport, and even there it is a no-op unless an `API_KEYS_SECRET` environment variable is configured; absent that, every request passes through. In the default stdio/local distribution the auth code is **not in the request path at all**. This is by design: the data agridata-mcp serves is entirely public and read-only, the local server is reachable only by the user who launched it, and authenticating a local stdio tool would add friction for no security gain. The auth path is retained, not removed, so that a self-hoster who chooses to expose the HTTP transport has a ready mechanism to lock it down. The corresponding self-hosting caveat — that exposing HTTP without configuring the secret runs the server open — is noted in Section 5.

4.5 The deferred document-RAG pipeline

Full-text analysis of the portal's downloadable documents — semantic search over the PDF corpus — was assessed and **deferred**, spun off to a separate project rather than abandoned, because the downloadable document corpus was largely unsuitable for automated text extraction. In its place, `search_bibliography` provides catalog discovery and document links — search across the bibliographic records with links to the underlying PDFs — rather than full-text or analytical search over the documents themselves. The work is deferred pending a tractable text source, not written off.

5. Limitations and Known Issues

This section is deliberately complete; for an archival document, candid limitations are a credibility feature.

Single hard dependency on the upstream portal (sharpest limitation). agridata-mcp in its current form has **no offline fallback**. Every tool that touches data proxies `catalog.agridata.tn` live. The local-snapshot layer that once provided resilience was reverted

off the main line (Section 4.3; it survives only on tag `v3.1.3`). The portal has demonstrably been unavailable at times — the snapshot work existed precisely because of such an outage. **When the portal is down, agridata-mcp is non-functional.** This is the direct, accepted cost of the freshness-over-resilience decision.

Pervasive data-quality issues from all-text typing. Because every DataStore field is text, numeric and date logic depends on casting with regex guards, and several concrete corruptions recur: Arabic **mojibake** in field names (on the order of 39 resources), fields literally named `"None"` (again on the order of 39 resources), Excel-overflow corruption (resources padded to 1,048,575 rows), **wide-format** resources that encode values in column names (around 29 use year-as-column layouts), and **CSV/XLSX format mislabeling** (on the order of 86 resources). agridata-mcp handles each where it can (Section 3.4), but handling is mitigation, not cure.

Restricted CKAN SQL. The DataStore SQL dialect is limited: `CASE WHEN` is not whitelisted (returns HTTP 409), some resources reject SQL outright (HTTP 403), and the four ONAGRI thematic-library resources block SQL entirely (409), which is why `search_bibliography` falls back to `datastore_search` plus Python-side filtering for them.

Coverage gaps. Not every domain covers all 24 governorates, and there are named holes — for example, Sousse has no climate-station resources, and some sensor stations lack particular sensor types. The data is as uneven as its many producers.

Limited automated testing. The repository's tests cover the CKAN client's base-URL handling, the schema registry's static load, and the `read_resource` download-failure message contract. There is **no end-to-end test asserting on a tool's rendered data output** — the `tests/test_tools/` package is an empty placeholder. Tool behavior has been validated chiefly through live exercise rather than committed regression tests.

Schema-registry staleness window. The live layer refreshes on a 6-hour interval, so resources added or removed on the portal within that window are not yet reflected. The committed static `schemas.json` is a point-in-time baseline that drifts from live reality as the portal's resource inventory changes between regenerations. The registry reconciles toward live on refresh, but the static file is a snapshot, not a mirror.

No full-text document search. The bibliographic layer offers catalog record discovery and links to the underlying PDFs, not full-text or analytical search over the documents themselves (Section 4.5).

Authentication is dormant by default (self-hosting caveat). A self-hoster who exposes the HTTP transport without setting `API_KEYS_SECRET` runs the server **open**. For the default local stdio use this is a non-issue, but it is a real caveat for anyone adapting the server to a networked deployment.

Partial semantic-layer coverage. Value hints cover **735 of roughly 1,248 DataStore resources (~59%)**. Resources without hints — those with no categorical columns, or only single-value columns — give the model less help when it constructs SQL against them.

French/Arabic-dominant metadata. Dataset titles, field names, and values are predominantly French with significant Arabic. Non-Francophone, non-Arabic users depend en-

tirely on the LLM client's own translation; the server does not localize.

5.1 Handoff to the planned Data Audit

The data-quality issues above are named here as **known and handled-where-possible**, not as a complete characterization. A systematic audit — quantifying mojibake, mislabeling, overflow, wide-format prevalence, SQL-blocked resources, and semantic-layer coverage gaps across the full catalog — is the proper instrument for that, and is planned as separate Data Audit work. This report draws the seam deliberately: it documents how the running system copes with these issues, and hands their exhaustive measurement to the audit rather than overclaiming completeness here.

6. Future Work

Concrete next steps, in rough priority order:

- **Lower the barrier to running the server.** The current source-only, local distribution serves a technical user. A packaged install (e.g. a published package or a one-command launcher) would extend reach to the non-technical users the data is meant for.
- **Optional offline resilience.** Reintroduce the local-snapshot path (preserved on tag `v3.1.3`) as an *opt-in* fallback rather than the default, recovering outage resilience without giving up live freshness.
- **Broaden semantic-layer coverage** beyond the current ~59% of DataStore resources, and refresh value hints on a schedule as the catalog changes.
- **Close the testing gap** with end-to-end tool-output regression tests.
- **Revisit document analysis** if and when a tractable text source becomes available.
- A systematic **Data Audit** (Section 5.1), which a separate evaluation effort will also build on.

7. Conclusion

agridata-mcp turns an awkward public CKAN catalog into something an LLM can query in natural language. Its value is not in the protocol it speaks or the API it wraps, but in the domain adaptation between them: a small hybrid tool suite, a schema registry that stays current without slowing tool calls, careful handling of the portal's many data-quality quirks, and — most of all — a semantic layer that bridges how users ask and how the data is actually structured. The system as documented here is a local, source-distributed server with a single live dependency on its upstream portal; its limitations are real and stated plainly. A separate evaluation of the architecture's effect on answer quality is left to future, separate work.

About the Author

Tarek Gasmi

FOUNDER, TANITDATA

Tarek Gasmi is Founder of tanitdata and Head of Data and AI at datadoit, with an academic affiliation at the University of Manouba. His work focuses on AI governance, digital infrastructure, sustainability, and technology policy, with particular attention to transitional economies and the geopolitical economy of AI systems.

Citation and License

Suggested citation. Gasmi, T. (2026). *agridata-mcp: A Local MCP Server for Conversational Access to Tunisia's Agricultural Open Data.* Technical report TD-TR-2026-01. tanitdata, Tunis.

License. This report is published under the Creative Commons Attribution 4.0 International (CC BY 4.0) license. The agridata-mcp software is distributed under its own repository license; the data it serves originates from catalog.agridata.tn under the Licence Nationale de Données Publiques Ouvertes (Décret 2021-3).

Version. v1.0, June 2026.

References

Model Context Protocol — open standard specification and SDKs. modelcontextprotocol.io.

CKAN — open-source data management system (DataStore and API). ckan.org.

Portail des Données Agricoles Ouvertes en Tunisie (catalog.agridata.tn), Ministère de l'Agriculture, des Ressources Hydrauliques et de la Pêche. Licensed under the Licence Nationale de Données Publiques Ouvertes (Décret 2021-3). catalog.agridata.tn.

agridata-mcp source repository, github.com/tanitdata/agridata-mcp (source tree, `schemas.json`, `value_hints.json`, `README`, `LICENSE`, `Dockerfile`). Version history referenced by tag (`v1.2-crops`, `v1.3-foundation-pivot`, `v3.1.3`, `v3.2.0`).