CELLULE-PRO DOCUMENTATION

API Reference

OpenAI-compatible endpoints · sk-cellule-* tokens

cellule.ai · pro.cellule.ai
2026-05-18

Contents

Contents **2**
0.1 Cellule-PRO — API reference . . . 3
Table of contents . . . 3
1. Base URL and versioning . . . 3
2. Authentication . . . 3
3. Conventions . . . 4
4. Endpoints by domain . . . 6
5. Error codes . . . 12
6. Integration examples . . . 13
7. Interactive Swagger / ReDoc . . . 14
Cross-references . . . 14

0.1 Cellule-PRO — API reference

Audience: application integrator, dev who connects a business client to Cellule-PRO (internal application, workflow automation, dedicated assistant).

Principle: Cellule-PRO is **API-first**. No user-facing chat UI. Everything goes through these endpoints. Cf. `feedback_enterprise_api_first_no_limits`.

Status: skeleton created session 2026-04-21 (RAG Phase 6 workstream). Sections to be completed along phases 1-5.

Table of contents

1. Base URL and versioning
2. Authentication
3. Conventions
4. Endpoints by domain
5. Error codes
6. Integration examples
7. Interactive Swagger / ReDoc

---

1. Base URL and versioning

- **Default base URL**: `https://<pool-host>/` where `<pool-host>` = the unified nginx entry of the cluster (`:18080` internally, behind client TLS / Cloudflare in prod).
- **Path pattern**: all public routes are prefixed `/v1/`.
- **Versioning**: prefix change `v1` → `v2` only on a breaking change. Additions (new optional field, new endpoint) stay under `v1` without increment.
- **Auto-generated Swagger**: `/docs` and `/redoc` (cf. §7).

---

2. Authentication

2.1 v1 — `X-Cellule-User` header

Current state: simplified auth, the pool trusts the user_id sent by the upstream gateway. Each request must carry:

```
X-Cellule-User: <user_id>
```

Without that header → `401`. This design assumes that real authentication (SSO / bearer token) is handled by a reverse proxy upstream of the pool (nginx entry, Cloudflare Access, VPN tunnel). The pool treats the user_id as ground truth.

Never expose the pool directly on the internet without a reverse proxy that authenticates upstream.

2.2 v1.5 — Internal bearer token (planned)

Planned: `Authorization: Bearer <token>` wired to the session module of the public pool. Rotation, grace period, revocation will be documented at implementation. Not shipped in v1.

2.3 Intra-project roles

Authentication identifies **who** (user_id). Authorization on a project resource depends on the user's role in `project_members`:

Role	Can
`viewer`	GET /projects, /documents, /search
`editor`	+ POST /documents, /memory
`owner`	+ PATCH/DELETE project, invite/remove members, GET /audit

Hierarchy: `owner > editor > viewer`. An endpoint that requires `min_role=editor` accepts `editor` and `owner`, refuses `viewer`.

No separate pool-admin scope in this API: infrastructure admin goes through `/v1/admin/*` (dashboard), not the project endpoints.

---

3. Conventions

3.1 Format

- Requests: UTF-8 JSON, `Content-Type: application/json`
- Responses: UTF-8 JSON
- Dates: ISO 8601 UTC (`2026-04-21T18:30:00Z`)

3.2 Pagination

- `GET /v1/projects`: returns all visible projects (no v1 pagination, list bounded by `DEFAULT_MAX_PROJECTS_PE`
- `GET /v1/projects/{id}/documents`: full list, sort `uploaded_at DESC`.
- `GET /v1/projects/{id}/memory`: list bounded to the 200 most recent rows.
- `GET /v1/projects/{id}/audit?limit=<1..500>`: only endpoint with explicit pagination via query param (default 100).

No cursor or `X-Total-Count` header in v1. Larger volumes → use `/v1/projects/{id}/search` with a semantic filter.

3.3 Rate limits

Application rate limit on sensitive mutating endpoints:

- `POST /v1/projects`: max `DEFAULT_MAX_PROJECTS_PER_USER=20` simultaneous active projects per user. Exceeding → `429`.

No per-user global quota nor `X-RateLimit-*` header in v1. If anti-abuse protection is needed → upstream nginx entry or gateway middleware.

3.4 Idempotency

- `project_documents` **table**: `UNIQUE (project_id, content_hash)` — two identical POSTs on the same document hash produce a single row. The second returns `409 Conflict`.
- `project_memories` **table**: `UNIQUE (project_id, fact_hash)` — same logic.
- **RAG ingest** (`_insert_chunks`): DELETE then INSERT in a transaction → re-ingest of the same document atomically replaces the existing chunks.

No `Idempotency-Key` header in v1; the schema-level UNIQUE constraints play that role.

3.5 SDK resilience — retry pattern (RAID-2 workstream)

The pool maintains a short-TTL local `pending_jobs` queue (5-minute TTL via `cleanup_expired_jobs`). This queue is **strictly local** by design — it is NOT replicated cross-pool, in line with the industry pattern (OpenAI, Anthropic, AWS Bedrock SDK).

Consequence: if a pool falls between accepting an inference request (`POST /v1/chat/completions`) and executing it, the request can be lost. The client must implement an idempotent retry.

Recommended pattern on the SDK / client side:

```
import time
import httpx
from httpx import HTTPStatusError, TimeoutException

def chat_completion_with_retry(client, payload, max_retries=3):
    """Exponential backoff on 503/504/timeout. OpenAI SDK pattern."""
    last_error = None
    for attempt in range(max_retries):
        try:
            r = client.post("/v1/chat/completions", json=payload, timeout=120)
            r.raise_for_status()
            return r.json()
        except HTTPStatusError as e:
            last_error = e
            if e.response.status_code in (503, 504, 502):
                # Pool unavailable / failover in progress / gateway timeout
                wait = 2 ** attempt  # 1s, 2s, 4s
                time.sleep(wait)
                continue
            raise  # 4xx = no retry (auth, format, etc.)
        except TimeoutException as e:
            last_error = e
            time.sleep(2 ** attempt)
    raise last_error
```

Why on the client side and not the pool side: - Ed25519 LAN gossip latency (~50-200ms) » fast inference latency: replicating the queue would be slower than the job itself - The client knows the context (natural idempotency of chat requests), can decide the business timeout (UI = 30s, overnight batch = 1h) - Distributed lock to share the queue = very heavy for a hypothetical gain (real failover ~seconds)

Natural idempotency: `chat/completions` requests are not "executed twice" in a problematic way — a retry after a 120s timeout simply generates a new response (the LLM is non-deterministic by construction). The client can display the response that arrives first.

For strictly idempotent workloads (overnight batch, document ingest), pass a client identifier in the payload `metadata` and deduplicate on the application side.

What IS replicated cross-pool in case of failover (RAID workstream shipped rc89→rc92): - Identities (`accounts`, `api_tokens`, `employees`) — Ed25519 propagation - User memory (`user_memories`, `agent_episodes`, `conversations`) — memory_replication gossip - RAG projects (`projects`, `project_documents`, `project_document_chunks`, `project_memories`, `project_conversations`, `project_members`) - Inference audit (`jobs_audit`) — cross-pool operational history - Model catalog (`pool_models`, `pool_model_assignments` - Configuration (`moe_local_config`, `pool_config`, `signup_config`)

What is NOT replicated (by design): - `pending_jobs` (local 5min TTL inference queue — see above) - `jobs` (operational history + smart routing embeddings — local per pool, but minimal audit propagated via `jobs_audit`) - `workers`, `worker_benchmarks`, `worker_tasks` (workers physically attached to a single pool) - `sessions`, `routing_feedback` (transient)

---

4. Endpoints by domain

4.1 Projects (workstream #10)

Method	Path	Min role	Description
`POST`	`/v1/projects`	auth only	Create a project (owner = author)
`GET`	`/v1/projects`	auth only	List the projects I am a member of
`GET`	`/v1/projects/{pid}`	viewer	Project details
`PATCH`	`/v1/projects/{pid}`	owner	Modify name/description/settings
`DELETE`	`/v1/projects/{pid}`	owner	Archive (soft delete, status='archived')
`POST`	`/v1/projects/{pid}/members`	editor+	Invite a user
`GET`	`/v1/projects/{pid}/members`	viewer	List members
`DELETE`	`/v1/projects/{pid}/members/{uid}`	owner	Remove a member
`PATCH`	`/v1/projects/{pid}/members/{uid}`	owner	Change a member's role

```
{
  "name": "Cabinet-Martin-Dossier-X",
  "description": "Collaborative space for client Martin SA dossier"
}
```

POST `/v1/projects` — **body**

- `name`: 1..256 chars, no newlines, uniqueness not globally guaranteed (two projects can share a name if owners differ).
- `description`: optional, free text.

Responses: - `201` → `{ project_id, name, owner_user_id, created }` - `400` → invalid name - `429` → user already has `DEFAULT_MAX_PROJECTS_PER_USER=20` active projects

```
{
  "user_id": "bob",
  "role": "editor"
}
```

POST `/v1/projects/{pid}/members` — **body**

- `role` `("owner", "editor", "viewer")`, default `editor` if omitted
- Ed25519-signed by the host pool before gossip — workstream #10 invariant

Responses: `201` / `400` (invalid role) / `403` (not editor+) / `404` (project not found) / `409` (already a member).

4.2 Documents

Method	Path	Description
`POST`	`/v1/projects/{id}/documents`	Upload document metadata (binary v1.5 multipart)
`GET`	`/v1/projects/{id}/documents`	List documents + indexing status
`GET`	`/v1/projects/{id}/documents/{doc_id}`	Detail + chunks_count
`DELETE`	`/v1/projects/{id}/documents/{doc_id}`	Delete (CASCADE chunks)

Minimum required role: `editor` for POST/DELETE, `viewer` for GET.

POST `/v1/projects/{id}/documents` — **attach metadata** JSON body:

```
{
  "filename": "contract-acme-client.pdf",
  "content_hash": "e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855",
  "size_bytes": 123456,
  "mime_type": "application/pdf"
}
```

Responses: - `201` → `{ project_id, document_id, filename, hosting_pool_id }` - `400` → missing or invalid field (`content_hash` must be SHA256 hex 64 chars, `size_bytes >= 0`) - `403` → role member < `editor` - `413` → `size_bytes > 50 MB` (cap locked in v1) - `415` → `mime_type` outside whitelist (TXT / MD / PDF / DOCX / ODT only)

No binary upload in v1 — this endpoint attaches metadata only. The binary will be uploaded by a separate v1.5 multipart endpoint. The RAG indexes as soon as the binary arrives.

`indexed_status` **lifecycle** Field on each document:

Value	Meaning
`NULL`	Legacy doc (uploaded before the RAG rollout) or binary not yet arrived
`pending`	Ingest dispatched, running in background
`done`	Chunks + embeddings stored in `project_document_chunks`, `chunks_count > 0`
`skipped`	Unsupported format / size exceeded / empty text / extract lib missing
`failed`	Pipeline error (embedder unavailable, DB error) — retry possible via re-POST

GET /documents response (example, indexing fields included):

```
{
  "project_id": "proj_abc123",
  "documents": [
    {
      "id": 42,
      "filename": "contract-acme-client.pdf",
      "size_bytes": 123456,
      "mime_type": "application/pdf",
      "indexed_status": "done",
      "chunks_count": 37,
      "indexed_at": "2026-04-21T18:45:12Z",
      "uploaded_at": "2026-04-21T18:44:58Z"
    }
  ]
}
```

Status polling The client can poll `GET /v1/projects/{id}/documents` after a binary upload to watch the `pending → done` transition. Typical time: 1-10s for a 5 MB PDF (384d MiniLM embedding on dormant CPU, cf. §1 ADMIN_GUIDE).

> *To be completed when the v1.5 binary multipart endpoint is implemented: form-data body, streaming upload, resume on interruption.*

4.3 RAG search (RAG workstream)

Method	Path	Description
POST	/v1/projects/{id}/search	KNN cosine over project_document_chunks

Minimum required role: viewer.

```
{
  "q": "retrieve judgments similar to the Martin SA case",
  "k": 10,
  "sources": ["doc"]
}
```

Body

Field	Type	Default	Constraint
q	string	— (required)	non-empty, truncated to 2000 chars
k	int	10	1 k 50
sources	list[string]	["doc"]	{"doc", "memory", "conv"}

v1 note: only the "doc" source is implemented. "memory" and "conv" are accepted but return empty with an explicit notes — pending the project_memories → vector(384) ALTER (v1.5).

```
{
  "project_id": "proj_abc123",
  "q": "retrieve judgments similar to the Martin SA case",
  "k": 10,
  "sources": ["doc"],
  "results": [
    {
      "source": "doc",
      "project_id": "proj_abc123",
      "document_id": 42,
      "chunk_id": 137,
      "chunk_idx": 4,
      "content_text": "… text of the semantically closest chunk …",
      "similarity": 0.87,
      "filename": "judgment-martin-2024.pdf",
      "mime_type": "application/pdf"
    }
  ],
  "notes": []
}
```

200 response

- similarity: cosine similarity [0, 1], 1 = identical

- Default min threshold: `min_similarity = 0.3` (silent if < threshold)
- Sort by `similarity` descending

Error codes

Code	Case
400	empty `q`, `k` outside [1,50], `sources` contains an unknown value
401	`X-Cellule-User` header missing
403	user is not a project member
404	`project_id` does not exist
410	archived project

Project isolation invariant (non-negotiable) The `WHERE project_id = $1` filter is **always** present in the SQL (verified by source-level test + molecule-guardian validation). **No cross-project leak possible by design** — attorney-client privilege + GDPR data segregation compliance.

```
curl -X POST https://pool.example.com/v1/projects/proj_abc123/search \
  -H "X-Cellule-User: alice" \
  -H "Content-Type: application/json" \
  -d '{"q": "non-compete clauses", "k": 5}'
```

curl example

4.4 Chat completions

`POST /v1/chat/completions` endpoint — 100% OpenAI-compatible schema. Any client that speaks the OpenAI API works without modification: opencode, Cursor, Continue.dev, Open-WebUI, openai-py, aichat, llm, curl…

Authentication

```
Authorization: Bearer sk-cellule-<token>
```

The token is generated on the user side (the "Use the API" tab) or on the admin side. It authenticates the call **and** serves as a crypto key to encrypt long-term memory (per-user RAG).

Minimal payload

```
{
  "model": "CELLULE",
  "messages": [{"role": "user", "content": "Hello"}]
}
```

The pool exposes a single virtual `CELLULE` model. The internal routing (tier-aware smart routing, MoE sharding, cross-pool forwarding) is **transparent** to the client: it picks the worker that fits the prompt best (Coder for code, long-context for large prompts, 9B for fast chat, etc.) and forwards to another LAN pool if needed.

SSE streaming: `"stream": true` in the payload. Standard OpenAI event-stream format. Also works when the request is forwarded to a peer (rc40+).

Tool-calling: `tools` field OpenAI-compat. Works transparently with opencode/Cursor/Continue — these clients send their tools in the payload, the LLM uses them (Read/Glob/Edit/Bash on the dev's local endpoint).

Automatic long-term memory (RAG) Each user (`sk-cellule-*` token) has an **encrypted memory** on the pool side, retrieved automatically on each call based on semantic similarity with the last user message.

Enable / disable: "Preferences" tab of the user space. Persistent mode = ON by default for api_gateway tokens. Private mode = no RAG, no conversation save.

`[MEMORIZE: fact]` pattern

To explicitly store a fact in long-term memory, ask the LLM to end its response with:

```
[MEMORIZE: your fact to memorize]
```

The pool intercepts that tag in the response, embeds it with the user token (per-user encryption), and strips it before returning to the client. The fact is then retrievable in a later session (another process, another machine, another conversation) as long as the token is the same.

Cross-session behavior

- Session N: normal conversation, or explicit `[MEMORIZE: ...]`. The conversation is embedded (auto-summarization) **and** the `[MEMORIZE: ...]` are persisted.
- Session N+1 (another process, no `--continue`): new user prompt. The pool does a RAG retrieval on the user's facts, injects the relevant facts, the LLM answers leaning on them.

Validated E2E with opencode (separate processes, no `--continue`) in tool-calling mode — see `feedback_killer_commercial_memoire_navigation_codebase.md` on the internal team side.

Example

```
# Session 1 (terminal A): plant a fact
curl $POOL/v1/chat/completions \
  -H "Authorization: Bearer sk-cellule-..." \
  -d '{"model":"CELLULE","messages":[{"role":"user",
        "content":"Note for tomorrow: the bug is in parse_log() line 42, current_section not init. End your

# Session 2 (terminal B, later, another laptop): retrieve
curl $POOL/v1/chat/completions \
  -H "Authorization: Bearer sk-cellule-..." \
  -d '{"model":"CELLULE","messages":[{"role":"user",
        "content":"Which bug was I supposed to fix?"}]}'

# → "UnboundLocalError line 42 in parse_log() when the log is
#    empty. Probable cause: current_section not initialized."
```

Tools & memory in agent mode (opencode/Cursor/Continue) When the payload contains `tools`, the pool injects the RAG context **into the last user message** (instead of the system prompt) to force saliency against the 5+ KB of tool-calling instructions sent by agent clients. Transparent behavior — the client's tools work normally, but the LLM consults user facts before exploratory tool-calling (Glob/Read/grep).

Without this targeted injection, the agent LLM tends to favor its tools over the RAG system prompt. With it, the killer feature "my LLM remembers me" also works in agent mode. Implemented pool.py rc46.

4.5 Admin

`/v1/admin/*` endpoints are reserved for the internal operator dashboard. Separate documentation `ADMIN_GUIDE.md`. Not intended for customer application integration.

---

5. Error codes

5.1 HTTP

Code	Case
400	Invalid body (format, missing fields)
401	Token missing or invalid
403	Valid token but insufficient scope
404	Resource not found (or invisible to this token)
409	Conflict (e.g. project name already taken)
413	Upload > 50 MB
415	Unsupported document format
422	Pydantic validation failed
429	Rate limit exceeded
500	Internal error (with trace_id in response body)

5.2 Error response structure

```
{
  "error": {
    "code": "project_not_found",
    "message": "Project 'xyz' does not exist or is not visible to you.",
    "trace_id": "abc123"
  }
}
```

To be completed: exhaustive list of business `codes` per domain.

---

6. Integration examples

6.1 curl — full project + RAG workflow

```
BASE=https://pool.example.com
USER=alice

# 1. Create a project
curl -X POST $BASE/v1/projects \
  -H "X-Cellule-User: $USER" \
  -H "Content-Type: application/json" \
  -d '{"name": "Cabinet-Martin-Dossier-X"}'
# → 201 {"project_id": "proj_abc123", ...}

PID=proj_abc123

# 2. Attach a document (v1 metadata)
curl -X POST $BASE/v1/projects/$PID/documents \
  -H "X-Cellule-User: $USER" \
  -H "Content-Type: application/json" \
  -d '{
    "filename": "contract-martin-2024.pdf",
    "content_hash": "e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855",
    "size_bytes": 123456,
    "mime_type": "application/pdf"
  }'
# → 201 {"document_id": 42, ...}

# 3. Poll indexing status
curl -s $BASE/v1/projects/$PID/documents \
  -H "X-Cellule-User: $USER" | jq '.documents[0].indexed_status'
# → "pending" then "done"

# 4. Semantic search inside the project
curl -X POST $BASE/v1/projects/$PID/search \
  -H "X-Cellule-User: $USER" \
  -H "Content-Type: application/json" \
  -d '{"q": "24-month non-compete clauses", "k": 5}'
# → 200 {"results": [{"source":"doc", "content_text": "...", "similarity": 0.87, ...}]}
```

6.2 Python (httpx)

```
import httpx

BASE = "https://pool.example.com"
HEADERS = {"X-Cellule-User": "alice"}

async def search_project(pid: str, query: str, k: int = 10) -> list[dict]:
    async with httpx.AsyncClient(base_url=BASE, headers=HEADERS) as client:
        r = await client.post(
            f"/v1/projects/{pid}/search",
            json={"q": query, "k": k},
        )
```

```
        r.raise_for_status()
        return r.json()["results"]
```

6.3 TypeScript (fetch)

```
const BASE = "https://pool.example.com";
const HEADERS = { "X-Cellule-User": "alice", "Content-Type": "application/json" };

async function searchProject(pid: string, q: string, k = 10) {
  const r = await fetch(`${BASE}/v1/projects/${pid}/search`, {
    method: "POST",
    headers: HEADERS,
    body: JSON.stringify({ q, k }),
  });
  if (!r.ok) throw new Error(`${r.status} ${await r.text()}`);
  return (await r.json()).results;
}
```

6.4 RAG framework integrations (LangChain / LlamaIndex)

Cellule-PRO exposes a `POST /v1/projects/{id}/search` endpoint compatible with a custom Retriever wrapper. Implementation of an official wrapper is tracked in `project_todo_client_framework_integratio` Not shipped in v1.

---

7. Interactive Swagger / ReDoc

FastAPI automatically exposes two auto-generated UIs:

- **Swagger UI**: `https://<pool-host>/docs`
- **ReDoc**: `https://<pool-host>/redoc`
- **OpenAPI JSON**: `https://<pool-host>/openapi.json`

For a "try it out": add the `X-Cellule-User: alice` header in the Swagger interface before triggering a request. Typed client generation: `openapi-generator-cli generate -i openapi.json -g <target>` (python, typescript-fetch, rust…).

Cross-references

- `ADMIN_GUIDE.md` — deployment, topology, operations
- `BOUNDARY.md` — public PRO contract (which endpoints are inherited from the public pool)
- `SECURITY_MODEL.md` — authn/authz model
- `CHANTIER_10_MODE_PROJET.md` — project scope genesis
- `CHANTIER_RAG_PROJECT_DOCUMENTS.md` — RAG search genesis