Agentscan Architecture Proposal

🌐 1. System goals

Mission:

Create an open, modular “agent scanner” for tracking **ERC-8004 agents (later other standards)** and **agent activity (x402 / MCP / A2A / etc.)** across multiple chains and off-chain protocols — similar in spirit to *Etherscan + Nansen + x402scan*, but open and extensible.

Core principles

1. **Multi-chain from day one** → modular chain adapters
2. **Open-source first** → clear contribution surfaces, strong CI/CD
3. **Stateless public explorer** → no logins, no accounts
4. **Read scalability** → API and UI optimized for cached, high-volume read queries
5. **Visitors:** can browse everything via the frontend.
6. **API consumers:** must go through the **frontend proxy** (not raw backend URLs).
7. **Indexers:** must be fully decoupled and scalable; backend reads from indexed DB/search only (never from live chains).

🧩 2. Architectural overview (multi-chain + modular)

Python
```

                      ┌──────────────────────────────┐
```

```
                         │        Next.js frontend                    │
                         │ Static pages + SWR + CDN                            │
                         └──────────────┬───────────────┘
                                        │
                        HTTPS / Same-Origin API Proxy
                                        │
                         ┌──────────────┴──────────────┐
                         │        FastAPI backend                          │
                         │  (read-only, rate-limited)                      │
                         └──────────────┬──────────────┘
                                         │
                      ┌───────────────────────────────────────────┐
                      │        High-availability data layer        │
                      │ Postgres (core) │ ClickHouse (events)     │
                      │ Opensearch (search) │ Redis (cache)        │
                      └──────────────┬──────────────┬──────────────┘
                            │                         │
        ┌─────────────────┘              └─────────────────┐
                       │                                                   │
      ┌──────────────┐                                  ┌──────────────┐
          │ Chain Indexers│                                    │ Off-chain
Indexers│
          │ (EVM, L2, etc.)│                                    │ (x402, MCP,
A2A)  │
      └──────────────┘                                  └──────────────┘
```

Indexers run as independent async services connected via a message queue and update the high-availability database and search clusters asynchronously from the main application.

3. API access model (public-for-frontend-only)

3.1 Goals

- Prevent API scraping and heavy third-party consumption.
- Preserve open browsing experience for human visitors.

3.2 Implementation

- **Next.js server acts as API proxy**:
    - Browser → frontend/api/* → proxied internally to FastAPI → response cached/CDN'ed.
    - API routes are accessible only through the Next.js proxy layer (e.g., /frontend/api/*) and are blocked from direct client access via firewall or routing rules.
- **Rate limits per IP & origin** via CDN / reverse proxy.
- **API keys** reserved for future partner integrations.
- **CORS locked**: only allowed origin is agentscan.org.

3.3 Result

✅ Visitors can use the full app.

❌ Random bots or external clients can't hammer the backend or bulk-extract data.

4. Indexing architecture

4.1 Overview

Indexers operate as independent async services with their own scheduling and fault-tolerant pipelines, ensuring the high-availability DB/search layer is continuously updated without impacting the main FastAPI or frontend services.

4.2 Indexer layer

A cluster of async workers, each a microservice:

Type	Purpose	Example
evm_indexer	Reads ERC-8004 & x402 events via web3, normalizes to unified schema	Ethereum, Base, Optimism
mcp_indexer	Polls MCP servers, collects context logs	OpenAI MCP, Anthropic MCP
a2a_indexer	Listens to agent-to-agent communications (public logs or events)	A2A messages
x402_indexer	Monitors x402 payment events / resource registries	x402scan-compatible

Each indexer writes to:

- **Raw queue** (events_raw)
- **Processing pipeline** (Celery → normalization)

- **Final database** (Postgres)

4.3 DB & search layer

- **Postgres (core)** → Agents, Identities, Reputations, Resources

- **Redis** → Hot cache for frequent queries
- Each store (Postgres) operates with read replicas for horizontal scalability and high availability; the FastAPI backend always connects in read-only mode.

Backfill:

Indexers can replay chain history from genesis or last checkpoint; supports reorg detection & resync.

Scaling:

- Each indexer runs independently per chain/protocol.

- Message queues (RabbitMQ / Redis Streams / Kafka) decouple ingestion from DB writes.

- Horizontal scaling by spawning more workers.

5. Backend design (FastAPI)

5.1 Data source

Backend **reads only** from the HA DB/search cluster — *never* directly from chain RPCs or external APIs.

5.2 API responsibility

- Provide **read endpoints** for the frontend (proxied).

- Expose **aggregation views** (e.g. /agents/:id/summary, /activities/feed).

- Serve **static JSON (SSR)** pre-rendered by the indexers for high throughput (fast path).

5.3 Caching hierarchy

1. Redis / CDN edge cache
2. DB read replica
3. Primary DB

This makes it scale to thousands of concurrent viewers easily.

6. Frontend (Next.js)

- Uses **ISR (Incremental Static Regeneration)** to pre-render pages from API data.
- **SWR hooks** for live updates (WebSocket → Redis pub/sub).
- No login or cookies.
- Queries go through the **proxy API layer**, not directly to FastAPI.

⚙️ 7. Multi-chain support design

Each chain adapter integrates into the indexing layer, not the FastAPI runtime, ensuring consistent ingestion across blockchains without affecting request latency.

7.1 Chain adapter interface (Python plugin pattern)

Each chain (Ethereum, Base, Solana, etc.) implements a lightweight **adapter class** registered via entry points.

Python
```python
# app/chains/base.py
class ChainAdapter:
    chain_id: str
    name: str
    async def fetch_blocks(self, start, end): ...
    async def parse_agent_events(self, block): ...
    async def normalize_event(self, raw_event) -> dict: ...
```

- Each adapter defines how to connect, fetch logs, and normalize ERC-8004-like events into a **common schema**.
- Register adapters dynamically:

```python
from importlib import import_module
from pkg_resources import iter_entry_points

def load_chain_adapters():
    adapters = {}
    for ep in iter_entry_points('agentscan.chains'):
        module = import_module(ep.module_name)
        adapters[ep.name] = module.Adapter()
    return adapters
```

Benefit:

New contributors can add a new blockchain by adding chains/xyz_adapter.py + tests, without touching core logic.

🧱 8. Codebase modularization for open-source contributions

/agentscan

/api ← FastAPI (read-only)

/frontend ← Next.js app (proxy API)

/indexers ← async workers (chain/offchain)

/schemas ← shared Pydantic models

```
/db           ← migration & seed scripts

/search        ← OpenSearch sync jobs

/docs

/tests

CONTRIBUTING.md
CODEOWNERS
LICENSE
```

Contribution workflow

- Each adapter / ingestor is in its own directory with a well-defined test harness and fixtures.
- Use **pytest** + **tox** + **pre-commit** (black, isort, mypy).
- PR template requires:
  - new adapter registered,
  - sample configuration (YAML),
  - unit test for one block or payload,
  - update to docs/chains.md.

Review simplicity

Core team reviews only interface compliance (Adapter + JSON schema validation). CI automatically runs normalization tests across test data.

🧮 9. Stateless design (no logins)

- **All frontend pages are public;** API endpoints are public only via the frontend proxy (not directly accessible to external clients).
- **User state** (favorites, settings, etc.) deferred to localStorage later.

- **Write operations**: none. It’s a pure *scanner*, so API is read-only.
- **Request flow** is cacheable end-to-end:
    - CDN (Cloudflare / Vercel Edge) caches GET responses for minutes.
    - API servers are stateless → scale horizontally.
- No user DB → drastically simpler ops and fewer privacy concerns.

🚀 10. Scaling strategy for many views

10.1 Fast read paths

- **Pre-compute & cache** agent summaries (agent_profile_view) in Postgres materialized views or Redis keys.
- Use **asyncpg** for high-performance Postgres IO.
- Add **pagination + infinite scroll** in UI, **cursor-based API** in backend.
- Heavy queries offloaded to read replicas.

10.2 Frontend optimization

- Use **Next.js static generation (ISR)**:
    - /agent/[address] → statically generated + revalidated every N minutes.
    - /agents and /resources → SSR + SWR caching.
- Edge CDN caching (Vercel, Cloudflare) to serve hot pages instantly.

10.3 Horizontal scaling

- FastAPI behind load balancer, stateless.
- Workers (indexers) can scale independently by chain or protocol.

- Event ingestion batched; real-time updates via Redis pub/sub or WebSocket gateway.

🔌 11. Data model (multi-chain aware)

Add chain_id to all relevant tables:

Table	Key fields	Notes	Write Process
ecosystem	id, namespace, identifier, [endpoint], name, description	Different ecosystems. Examples for namespace: google, eip155; Examples for identifier: 1 (for Ethereum if namespace is eip155); Endpoint examples: etherscan.io for Ethereum. Name, description for display.	A config file defines all fields. Added via PR. Migration job adds the new line in the DB.
registries	id, ecosystem_id, endpoint, name, description	Different registries of agents in an ecosystem. This would be something like the ERC8004 registry on Ethereum or the Olas registry on Ethereum	Config file (defining all fields of on-chain or off-chain anchors). Added via PR. Same PR contains on-chain/off-chain indexing logic in Python. Once merged, migration job updates DB and indexers keep it updates.
agent_identities	id, registry_id, identifier, publisher, endpoint, type, metaData	Example of an identifier could be a token_id. Example of a publisher could be an email or an address. Example of endpoint could be an address. Type is enum. MetaData would be including the creation transaction for instance and other optional fields	See above

interactions	id,source_agent_identity_id, [target_agent_identity_id], [ecosystem_id], activity_type, payload	unify x402, MCP, A2A Must have either a target_agent_id OR ecosystem_id (including (maybe both but) not neither).	Someone defines a type of interaction (e.g. x402) via config in PR. They must make sure that before opening the PR for the interaction all the agent_identities/registries/ecosystem are backfilled. PR for interactions includes indexing logic for that particular interaction that henceforth runs in an indexer for the relevant ecosystem.
events_raw	ecosystem_id, source_type, raw_json	raw logs per chain	
agents	id, name, description, confidence_score, metadata	Name, description: comes out of ML model, Confidence_score: A score (from ML model) representing how confidently identities are grouped under this agent.	Gets purely created based off indexed data. So core contributors define and maintain core processing logic that runs in background tasks that defines and updates these properties.
agent_identity_links	agent_id, agent_identity_id, confidence, method	Agent_id is foreign key of agents.id, agent_identity_id is foreign key of agent_identities.id, Confidence that this identity belongs to the agent, ML or heuristic method used for linkage	See above

Chain ID (EIP-155 numeric) ensures no collision between networks.

Sample data for x402 [draft]:

Ecosystem: id 1, namespace EIP155, Identifier 100, endpoint https://gnosisscan.io/, name Gnosis mainnet, description Gnosis mainnet

Identity: Id 1, ecosystem_id 1, endpoint 0x901A2cBb3F44E73B03609351c27Cbc82d4Cc1036, name trader eoa on gnosis, description null, createdAt nov 6 1:45 UTC, metadata {}

Agent: Id 1, identity_id 1, identifier 0x4c190e3C582e877942007Db3009feACc1Da6C481(jenslee trader multisig), publisher jenslee.dsouza@valory.xyz, endpoint null, type Olas trader, createdAt nov 6 1:45 UTC, metadata {}

Agent: Id 2, identity_id 1, identifier 0x12A9a43b97985F160B1ca4F28B4bb8fe359Aa21b(prod x402 proxy server), publisher valory.xyz, endpoint https://x402.olas.autonolas.tech, type resource server, createdAt nov 6 1:45 UTC, metadata {}

Settlement: Id 1, namespace EIP155, Identifier facilitator_address, endpoint facilitator-url, name valory prod facilitator, description x402 facilitator

Interactions: id 1, source_agent_id 1, target_agent_id 2, activity_type x402-payment, payload 0.001 USDC

Interations: Id 2, source_agent_id 1, target_agent_id 2, activity_type x402-payment, payload 0.001 USDC

Interactions: id 3, source_agent_id 1, target_agent_id null, activity_type unknown, payload 5 xDAI

Events_raw: ecosystem_id 1, source_type onchain-logs,

🪶 12. Developer ergonomics

- **Config-driven:** config.yaml lists active chains, endpoints, API keys.
- **Local development:** docker-compose up brings Postgres, Redis, FastAPI, Next.js dev server.
- **Docs site:** docs/ built with Docusaurus or MkDocs, auto-published from main branch.
- **Public API playground:** auto-generated Swagger + GraphQL schema.

🧠 13. Security / governance

- No user auth = smaller attack surface.
- Limit any webhook sources (x402 etc.) by IP or signature verification.
- Use read-only API keys for external consumers later.
- Core maintainers control merges to /core and /models only; everything else modular.
- **Frontend proxy** hides backend endpoints.
- **CORS** limited to agentscan.org.
- **Rate limiting** at CDN + app layer.
- **DB isolation:** indexers write → read replicas serve → backend readonly connection pool.
- **Auditable ingestion:** raw event store + replay pipeline.

🪜 14. Implementation roadmap

Phase	Goal	Key deliverables
M1	Core scanner (EVM only)	ERC-8004 event indexer, FastAPI read API, static Next.js UI
M2	Multi-chain plug-in framework	Chain adapter base, test harness, 2nd chain demo

M3	Off-chain connectors	x402, MCP ingestion pipelines
M4	Scaling + caching	CDN + Redis + read replicas
M5	Community contributions	docs, PR templates, example new adapter

🧭 15. Example contribution flow

1. Fork repo
2. Create chains/polygon_adapter.py implementing ChainAdapter.
3. Add test file tests/test_polygon_adapter.py.
4. Run make test → must pass normalization schema tests.
5. Submit PR → CI runs chain validation.
6. Core team merges after review of adapter outputs only.

This keeps the review burden light and the core team focused on stability, not code specifics.