Platform Architecture Whitepaper

Elis AI Technical Documentation

Generated: 2026-05-19 12:44 UTC

Repository: README

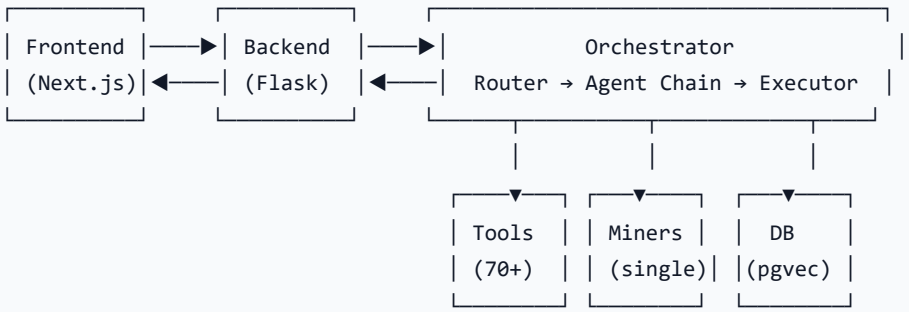
Elis AI

A dynamic AI orchestration platform that routes user prompts through adaptive agent chains, executes them across distributed model miners, and continuously improves routing quality through feedback-driven evaluation.

Current Listen Mode

- Listen currently uses browser-native speech by default.
- Server-side TTS via Piper is optional and only active when Piper and a voice model are installed in the backend container.

Architecture Overview



Two-Layer Chain System

- **Router chains** — select the best agent for a given prompt using similarity scoring, prior quality (EMA), and live miner feasibility.
- **Agent chains** — reusable execution blueprints containing rules, knowledge-base references, tool bundles, and multi-step workflow definitions.

Miner System

Model execution is dispatched to **miners** — containerized inference workers running OpenAI-compatible or Ollama-based models. Miners register over WebSocket and are scheduled by tier (nano / small / frontier / self-hosted).

Feedback & Evaluation

Every run produces a full trace (router selection, agent chain steps, tool calls, model I/O). A grading pipeline scores outputs and propagates EMA quality signals back to agents, templates, tool bundles, and model profiles — driving future routing decisions.

Project Structure

```
elis-ai/
├── backend/                # Python backend service
│   ├── agents/             # Core orchestration engine
│   │   ├── orchestration.py #   Chain executor, skill chains, execution modes
│   │   ├── router.py        #   Dynamic router-chain selection & ranking
│   │   ├── db.py            #   PostgreSQL + pgvector data layer
│   │   ├── vectorstore.py   #   Embedding search (agents, templates, bundles)
│   │   ├── tools/           #   70+ tool implementations
│   │   │   ├── bundle_resolver.py    # Tool bundle resolution + universal tools
│   │   │   ├── tool_calling_service.py # LLM function-calling dispatch
│   │   │   ├── web_search_service.py   # Web search integration
│   │   │   ├── sql_agent_tool.py       # LangChain SQL agent
│   │   │   └── ...
│   │   ├── seeds/           #   Seed data & agent catalog
│   │   ├── templates/       #   Agent templates
│   │   └── ...
│   ├── api.py              # REST API (auth, conversations, messages)
│   ├── company_api.py      # Multi-tenant company endpoints
│   ├── miner_api.py        # Miner registration & management
│   ├── miner_ws.py         # Miner WebSocket handler
│   ├── server.py           # Flask app setup & CORS
│   ├── main.py             # Orchestration entry point
│   ├── auth.py             # JWT + bcrypt authentication
│   └── tests/              # Unit & integration tests
├── frontend/               # Next.js 14 React application
│   ├── app/                #   App router (pages, layouts, API routes)
│   ├── components/         #   React components (chat, blocks, trace, admin)
│   ├── hooks/              #   Custom React hooks
│   ├── lib/                #   Utility functions & API client
│   └── styles/             #   Tailwind CSS
├── miner/                  # Model inference workers
│   ├── app.py              #   Flask miner service
│   ├── Dockerfile          #   OpenAI-based miner
│   ├── Dockerfile.ollama   #   Ollama self-hosted miner
│   └── openai_miners.json  #   Model definitions
├── browser_service/        # Selenium browser automation
├── services/               # Shared services (KBS)
├── config/                 # Configuration files
├── docs/                   # Documentation
├── docker-compose.yml      # Full stack compose (all services)
└── docker-compose.miners.yml # Azure miner deployment config
```

Tech Stack

Layer	Technology
Frontend	Next.js 14, React 18, Tailwind CSS, React Query
Backend	Python, Flask, Gunicorn + Uvicorn (ASGI)
Database	PostgreSQL 16 with pgvector extension
Cache	Redis 7
AI Models	OpenAI (GPT-5 family), Ollama (Qwen, Phi, Gemma)
Tools	LangChain, Presidio (PII redaction), spaCy
Deployment	Docker Compose, Azure Container Apps

Prerequisites

- **Docker** and **Docker Compose** (v2+)
- **Node.js** 18+ (for local frontend development)
- **Python** 3.11+ (for local backend development)
- **OpenAI API key** (for OpenAI-based miners)

Quick Start

1. Clone and configure

```
git clone <repo-url> elis-ai
cd elis-ai
```

Copy and edit environment files for each service:

```
# Backend
cp backend/.env.example backend/.env    # Set DB, Redis, JWT secrets

# Miner
cp miner/.env.template miner/.env      # Set OPENAI_API_KEY, enrollment secret

# Frontend
cp frontend/.env.example frontend/.env  # Set NEXT_PUBLIC_API_URL
```

2. Start with Docker Compose

```
docker-compose up -d --build
```

This starts all services:

Service	Port	Description
frontend	3000	Next.js web UI
backend	8000	Flask API + orchestrator
postgres	5432	PostgreSQL + pgvector
redis	6379	Cache & message broker
pgadmin	5050	Database admin panel
miner-openai-*	—	OpenAI model miners
miner-qwen-*	—	Ollama Qwen miners
miner-phi-*	—	Ollama Phi miners
miner-gemma-*	—	Ollama Gemma miners
browser	9020	Selenium automation

3. Access the application

- **Web UI**: http://localhost:3000
- **API**: http://localhost:8000
- **PgAdmin**: http://localhost:5050

Configuration

Key Environment Variables

Variable	Service	Description
`DATABASE_URL`	Backend	PostgreSQL connection string
`REDIS_URL`	Backend	Redis connection string
`JWT_SECRET`	Backend	Secret for JWT token signing
`DB_ENCRYPTION_KEY`	Backend	Fernet key for encrypted DB connections
`OPENAI_API_KEY`	Miner	OpenAI API key for model inference
`MINER_ENROLLMENT_SECRET`	Both	Shared secret for miner registration
`SQL_AGENT_MODEL`	Backend	Model for SQL agent (default: gpt-4.1-mini)
`REDACT_ENABLED`	Backend	Enable PII redaction (true/false)
`AZURE_COMMUNICATION_SERVICES_CONNECTION_STRING`	Backend	Azure Communication Services Email connection string
`AZURE_COMMUNICATION_EMAIL_SENDER`	Backend	Verified Azure sender address used for auth emails
`AUTH_REQUIRE_EMAIL_VERIFICATION`	Backend	Require signup email confirmation before login
`AUTH_REQUIRE_EMAIL_2FA`	Backend	Require an emailed login code after password authentication; off by default unless a deployment explicitly enables it

Orchestration Flow

Each user message follows this execution pipeline:

```
RECEIVE → PRE_RESPONSE_GATE → GENERATE → DELIVER → PERSIST
```

1. **Receive** — Parse user message, resolve tenant context.
2. **Pre-Response Gate** — Evaluate context size, token budget, memory tiers.

3. **Generate** — Router selects agent chain → executor dispatches to miners with tool access.
4. **Deliver** — Stream response blocks (text, evidence, artifacts) to the frontend via SSE.
5. **Persist** — Store adaptive records, update embeddings, trigger template candidate generation.

Execution Modes

The orchestrator dynamically selects one of four modes per request:

- `direct_answer` — Simple factual responses.
- `retrieval_augmented` — Augmented with KB/document retrieval.
- `skill_chain` — Multi-step workflow (search → expertise → context → answer).
- `multi_step_workflow` — Complex task decomposition and execution.

Multi-Tenancy & Roles

The platform supports multi-tenant company isolation with role-based access:

Role	Level	Capabilities
Owner	5	Full control, billing
Admin	4	Manage members, settings
Dev	3	Audit conversations, SQL access
Member	2	Chat, view own conversations
Viewer	1	Read-only access

Automations Product (Widgets)

- Automations is an organization-scoped product surface for recurring research widgets.
- The UI unlocks only when the user belongs to at least one organization.
- Widgets support mobile-friendly layout behavior, responsive controls, and drag/resize cards.
- Output modes include chart-first and non-chart render styles: `bar`, `line`, `pie`, `doughnut`, `radar`, `table`, and `text`.
- Notification dispatch supports threshold/text criteria plus a `Test Alert Now` action for immediate email verification.

Development

Backend (local)

```
cd backend
python -m venv .venv
source .venv/bin/activate  # or .venv\Scripts\Activate.ps1 on Windows
pip install -r ../requirements.txt
python -m flask --app server:app run --port 8000
```

Frontend (local)

```
cd frontend
npm install
npm run dev
```

Running Tests

```
cd backend
python -m pytest tests/ -v
```

Deployment

The platform deploys to **Azure Container Apps** using Docker images pushed to GitLab Container Registry. See deploy.md for detailed deployment instructions.

```
# Build and push images
docker build -t [private container registry]/backend:v1.0.5 -f backend/Dockerfile .
docker build -t [private container registry]/frontend:v1.0.5 -f frontend/Dockerfile .

# Deploy via Azure CLI
az containerapp update --name backend --resource-group <rg> --image <backend-image>
az containerapp update --name frontend --resource-group <rg> --image <frontend-image>
```

License

Docs: Architecture Performance Audit

Architecture & Performance Audit Report

Date: 2026-03-16
Baseline: 43.7s avg/turn (10-turn ML conversation), single-miner dispatch, 2 active OpenAI miners
Scope: Full backend orchestration pipeline — router → context → executor → persistence → tools

Part 1: Full Findings by Severity

CRITICAL

#	File	Function	Issue	Impact	Recommendation
C1	`agents/adaptive_persistence.py`	`_persist_adaptive_after_run()`	**Entire post-run persistence runs synchronously** before orchestrator returns. Includes LLM calls, N+M+8 DB writes, 3+2M embedding calls.	Blocks response delivery by 2-8s per turn depending on artifact count.	Move to background thread/task. Fire-and-forget after response yield.
C2	`agents/tools/__init__.py:1659`	`plan_tool_invocation()`	**LLM call on every tool planning request** — `gateway.generate(planner_prompt)` is mandatory, not fallback. Heuristic path only fires on LLM failure.	Adds 1-3s synchronous blocking per tool-planning step. On multi-tool chains, compounds.	Invert: make heuristic primary, LLM only when heuristic confidence < threshold. Cache plans by canonical prompt hash.
C3	`agents/orchestration.py:2938`	`WorkingContextAssembler.build()` → `ContextPacker.pack()`	**Conditional LLM calls via `_orch_transform()`** (3-5 calls per pack). Each calls `summarizer.summarize()` if text exceeds limit. No caching of summaries between calls.	0-5 LLM calls per context assembly. On long conversations, almost always triggers.	Cache episode summaries in DB keyed by (conversation_id, message_range_hash). Only regenerate on new messages.
C4	`agents/persistence.py:630`	`_filter_run_log()`	**json.loads on every history row** — unbounded payload size, deserialization happens even for fields never used.	O(max_items) parsing with large payloads per context load. With max_items=24, could be 24 large JSON parses.	Lazy deserialization: parse only on access. Or use SQL projection to return only needed fields.
C5	`agents/adaptive_persistence.py:711-717`	`persist_adaptive_records()`	**Duplicate artifact embedding** — same artifact indexed to generic namespace AND domain namespace = 2M embedding API calls instead of M.	Doubles OpenAI embedding API cost and latency for artifacts. 4 artifacts = 8 embedding calls.	Embed once → store vector → index into both namespaces with single embedding.

HIGH

#	File	Function	Issue	Impact	Recommendation
H1	`agents/adaptive_persistence.py`	`persist_adaptive_records()`	**N+M+8 individual DB writes per request** with no batching. Each INSERT is a separate round-trip.	With 5 steps + 3 artifacts = 16 DB round-trips. At ~2-5ms each = 30-80ms, but compounds under load.	Batch into single multi-row INSERT using `executemany()` or VALUES-list.
H2	`agents/embeddings.py:63-78`	`_embed_openai()`	**No caching on OpenAI embedding calls**. Every `embed()` hits the API. Same text embedded multiple times across calls.	100-500ms per API call. Same prompt embedded in context load AND artifact persistence.	Add Redis cache layer: key = `sha256(text[:16000])`, TTL 24h.
H3	`agents/tools/__init__.py:811-819`	`critic_agent()`	**5 tool calls executed sequentially**: gap_analysis, missing_constraints, assumption, contradiction, logic. Each may call web_search.	If each tool takes 1-3s, full critic chain = 5-15s.	Parallelize with `concurrent.futures.ThreadPoolExecutor`. All 5 are independent.
H4	`agents/persistence.py:700-725`	`_build_episode_summaries()`	**EpisodeSummaryGenerator has no caching**. Regenerates summaries from scratch on every `load_conversation_context()` call.	Conditional LLM via `_compress_text()` for messages older than recent_window. Repeated work across turns.	Persist episode summaries in `conversation_episode_summaries` table. Only regenerate for new message chunks.
H5	`agents/expert_router.py:140`	`_find_best_expert()`	**2 DB queries per facet** for expert selection. 3 facets = 6 DB queries. No batch domain→expert lookup.	6+ DB queries just for expert selection in multi-facet routing.	Batch: `SELECT * FROM agents WHERE domain IN (...)` single query for all facets.
H6	`agents/tools/__init__.py:1871`	`execute_tool()` → `get_tool_registry()`	**Tool registry rebuilt per call** — re-reads generated_tools.py and external catalog every invocation.	File I/O + dict construction per tool execution. On 5-tool chains, 5× overhead.	Memoize with signature-based invalidation (file mtime or content hash).

#	File	Function	Issue	Impact	Recommendation
H7	`agents/adaptive_persistence.py`	`_cap_payload()`	**Double JSON serialization** — `json.dumps()` to check byte size, then data is serialized again on storage. Happens 4× per step.	8× json.dumps instead of 4× needed. On large payloads (context, response), measurable.	Serialize once, check `len(serialized)`, store the already-serialized bytes.

MEDIUM

#	File	Function	Issue	Impact	Recommendation
M1	`agents/db.py`	Multiple functions	**50 json.dumps/json.loads occurrences** — pervasive manual serialization throughout DB layer.	Cumulative overhead per request. Every row read/write involves JSON parsing.	Consider JSONB columns where supported, or centralized serde helpers that avoid redundant work.
M2	`agents/model_clients.py:259`	`ModelGateway.generate()`	**No retry logic or backoff**. Single attempt per dispatch path, then cascades to next fallback.	Transient OpenAI failures immediately cascade to Ollama (slower). No recovery from brief network blips.	Add 1 retry with 2s backoff for OpenAI path before cascading.
M3	`agents/orchestration.py`	Repair/reroute loops	**Fully sequential repair/reroute** — when output fails evaluation, retries happen sequentially with no parallel exploration.	N sequential retries × full LLM call time. 3 retries = 3× latency.	For reroute, try 2 alternate miners in parallel, take first success.
M4	`agents/db.py`	Schema	**Missing composite indexes** — `conversation_facts(conversation_id, status, fact_type)` filters on all three but only indexed on first two.	Suboptimal query plan on fact_type filtering.	Add composite index.
M5	`agents/response_postprocess.py:105`	`enforce_source_integrity()`	**Source claim regex compiled fresh each call** — `re.IGNORECASE` pattern not precompiled.	Micro-optimization. Regex compilation ~0.1ms per call.	Precompile as module-level `_SOURCE_CLAIM_RE = re.compile(...)`.
M6	`agents/narrow_router.py:182`	`_llm_classify()`	**Optional LLM classification** — default disabled but no audit trail. If someone enables `ORCH_NARROW_ROUTING_USE_LLM=1`, adds 1-2s per request.	Risk of accidental production cost.	Add startup warning log if LLM narrow routing is enabled.

#	File	Function	Issue	Impact	Recommendation
M7	`agents/tools/__init__.py:158-211`	`_get_cached_payload()` / web cache	**Redis JSON serialize/deserialize on every web cache hit**.	Acceptable with TTL, but high-frequency paths (deep_search) multiply cost.	Consider msgpack or pickle for Redis values to reduce serde overhead.

LOW / INFO

#	File	Function	Issue	Impact	Recommendation
L1	`agents/orchestration.py`	`EvaluationService.grade()`	Pure heuristic grading — no LLM.	None (good design).	Keep as-is.
L2	`agents/facet_decomposer.py`	`DomainClassifier.classify()`	Uses embedding similarity, not LLM.	Efficient.	Keep as-is.
L3	`agents/response_postprocess.py`	`apply_postprocess_pipeline()`	No LLM, pure string transforms. Personality/style registries cached.	Negligible overhead.	Keep as-is.
L4	`agents/tools/self_tool_builder.py`	`build_and_register()`	Up to 2 LLM calls but only on new tool creation (rare). Hash dedup prevents rebuilds.	Amortized near-zero.	Keep as-is.

Part 2: Three-Frame Classification

Frame 1: Latency-Critical Hot Path (blocks user response)

These operations execute **before** the user sees any response:

Operation	LLM Calls	DB Queries	Embedding Calls	Location
`load_conversation_context()`	0-5 (episode summaries)	4-6	0-1 (if semantic rerank)	persistence.py:410
`WorkingContextAssembler.build()`	0-5 (_orch_transform)	0	0	orchestration.py:1449
`narrow_route()`	0-1 (if LLM enabled)	0	0	narrow_router.py:122
`FacetDecomposer.decompose()`	0	1-2	1	facet_decomposer.py:259
`_find_best_expert()`	0	2× per facet	0	expert_router.py:140
`plan_tool_invocation()`	**1 mandatory**	0	0	tools/**init**.py:1659
`executor.execute()` → `ModelGateway.generate()`	**1 (primary)**	0	0	model_clients.py:259
`apply_postprocess_pipeline()`	0	0	0	response_postprocess.py:50

Total hot-path LLM calls per request: 2-12 (1 mandatory executor + 1 mandatory tool planner + 0-10 conditional)

Frame 2: Can Be Deferred Until After Answer

These run synchronously but could be moved to background:

Operation	LLM Calls	DB Writes	Embedding Calls	Location
`_persist_adaptive_after_run()`	1 (domain classify)	N+M+8	3+2M	adaptive_persistence.py
`save_router_trace()`	0	1	0	db.py
`save_step_traces()`	0	N	0	db.py
`embed_message()`	0	1	1	persistence.py:537

Frame 3: Can Run in Parallel Safely

These are currently sequential but have no data dependencies between them:

Parallel Group	Operations	Savings
Context load	`_filter_run_log()` ‖ `_load_persistent_snapshot()` ‖ `_load_persistent_facts()` ‖ `_load_persistent_tasks()`	4 sequential queries → 1 round-trip
Critic agent	gap_analysis ‖ missing_constraints ‖ assumption ‖ contradiction ‖ logic	5× latency → 1×
Expert selection	`_find_best_expert(facet1)` ‖ `_find_best_expert(facet2)` ‖ ...	N× → 1× (or batch SQL)
Artifact embedding	`index_embedding(generic)` ‖ `index_embedding(domain)`	2× → 1× (embed once, index twice)
Post-run persistence	All DB writes in `persist_adaptive_records()`	Sequential → batch INSERT

Part 3: Prioritized Action Plan

A. Top 10 Highest-Impact Fixes

Rank	Finding	Estimated Savings	Effort	Risk
1	**Move `_persist_adaptive_after_run()` to background** (C1)	2-8s/turn	Medium	Low — fire-and-forget, no data dependency
2	**Cache episode summaries in DB** (C3 + H4)	1-5s/turn (avoids repeated LLM summarization)	Medium	Low — additive cache, fallback = regenerate
3	**Invert tool planner: heuristic first, LLM fallback** (C2)	1-3s/turn for simple prompts	Small	Medium — verify heuristic quality
4	**Deduplicate artifact embeddings** (C5)	50% embedding API cost + 200-1000ms/turn	Small	Low — single embed, dual index
5	**Add Redis cache to EmbeddingService** (H2)	100-500ms per repeated embed	Small	Low — additive cache layer
6	**Batch DB writes in persistence** (H1)	30-80ms/turn, much more under load	Small	Low — same data, fewer round-trips
7	**Parallelize context load queries** (Frame 3)	10-30ms (4 queries → 1 wall-clock)	Small	Low — read-only, no ordering deps
8	**Lazy JSON deserialization in `_filter_run_log()`** (C4)	Proportional to payload size × max_items	Medium	Low — access-on-demand wrapper
9	**Parallelize critic_agent 5 tools** (H3)	5-12s on critic-heavy paths	Small	Low — independent tools
10	**Batch expert selection queries** (H5)	4-10ms per extra facet	Small	Low — single SQL with IN clause

B. Quick Wins (Under 1 Hour Each)

#	Fix	File	Change
Q1	**Deduplicate artifact embedding** — embed once, index to both namespaces	`adaptive_persistence.py:711-717`	Call `embed()` once, pass vector to both `index_embedding()` calls
Q2	**Fix `_cap_payload()` double-serialize** — serialize once, check len	`adaptive_persistence.py`	Store result of first `json.dumps()`, reuse for size check and storage
Q3	**Precompile source-claim regex**	`response_postprocess.py:105`	Move `re.compile(...)` to module level
Q4	**Add Redis embedding cache**	`embeddings.py:53-78`	Wrap `_embed_openai()` with Redis get/set, key= `sha256(text)`
Q5	**Batch expert selection SQL**	`expert_router.py:140`	Replace per-facet queries with `WHERE domain IN (...)`
Q6	**Memoize `get_tool_registry()`**	`tools/__init__.py:1871`	Add `@lru_cache` or signature-based memoization
Q7	**Add composite index**	`db.py` / migration	`CREATE INDEX idx_conv_facts_full ON conversation_facts(conversation_id, status, fact_type)`

C. Refactors (Under 1 Day Each)

#	Refactor	Files	Description
R1	**Background persistence** — move `_persist_adaptive_after_run()` to a background thread	`orchestration.py`, `adaptive_persistence.py`	Wrap full post-run persistence in `threading.Thread(daemon=True).start()`. Ensure DB connections are thread-safe (new connection per thread or connection pool).
R2	**Episode summary caching** — persist summaries in DB table	`persistence.py`, `db.py`	Create `conversation_episode_summaries` table with `(conversation_id, message_range_hash, summary_json)`. Check cache before calling `EpisodeSummaryGenerator`. Invalidate when new messages arrive in range.
R3	**Invert tool planner priority** — heuristic-first architecture	`tools/__init__.py:1647-1680`	Run `_heuristic_tool_plan()` first. If confidence >= threshold (e.g., 0.7), return immediately. Only call `gateway.generate()` on low-confidence or missing-tool scenarios.
R4	**Lazy context deserialization** — proxy object for history rows	`persistence.py:625-640`	Return lightweight row objects that defer `json.loads()` until `.payload` is accessed. Avoids parsing unused rows.
R5	**Batch all persistence DB writes**	`db.py`, `adaptive_persistence.py`	Collect all INSERTs in a list, execute single `executemany()` at end of persistence cycle.
R6	**Parallelize context load** — concurrent DB queries	`persistence.py:410-440`	Use `concurrent.futures.ThreadPoolExecutor` to run `_filter_run_log`, `_load_persistent_snapshot`, `_load_persistent_facts`, `_load_persistent_tasks` in parallel.

D. Larger Architectural Improvements

#	Improvement	Scope	Description	Expected Impact
A1	**Async persistence pipeline**	Backend-wide	Replace synchronous `_persist_adaptive_after_run()` with Redis-backed task queue (e.g., `rq` or Celery lite). Decouple persistence from request lifecycle entirely. Enables retry on failures without blocking users.	Eliminate 2-8s of post-run latency. Enable persistence scaling independent of request handling.
A2	**Tiered context assembly**	Orchestration	Implement 3-tier context: (1) Always-hot: last 6 messages raw, (2) Warm: episode summaries from DB cache, (3) Cold: full history on-demand. Only promote tiers as needed by context budget. Eliminates redundant summarization.	Reduce context assembly from 0-5 LLM calls to 0-1. Faster turn times on long conversations.
A3	**Embedding service with batch API**	Embeddings	Replace individual `httpx.post` calls with OpenAI batch embedding endpoint (`input: [text1, text2, ...]`). Process all artifact embeddings in single API call instead of 2M individual calls.	Reduce embedding latency from O(M) API calls to O(1). ~80% reduction in embedding wall time.
A4	**Tool plan cache with semantic similarity**	Tools	Build a plan cache keyed by canonical prompt embedding similarity. When a new prompt is >0.95 similar to a cached plan, return cached plan directly. Falls back to heuristic → LLM chain.	Eliminate tool planner LLM call for repeat/similar queries. Faster on steady-state workloads.
A5	**Connection pooling for DB**	DB layer	Replace per-query connection with `psycopg2.pool.ThreadedConnectionPool` or `asyncpg` . Currently each DB operation may create/close connections.	Reduce connection overhead. Enable true parallel DB queries. Foundation for async migration.
A6	**ModelGateway retry with jittered backoff**	Model clients	Add configurable `max_retries=2` with exponential backoff + jitter for OpenAI path before cascading to Ollama. Prevents thundering herd on transient failures.	Recover from brief API blips without 120s Ollama fallback penalty.

E. Benchmark Plan

Baseline Measurements Needed

Metric	How to Measure	Current Value
End-to-end turn latency	10-turn ML conversation test	43.7s avg
Context assembly time	`[TIMING] load_conversation_context` log	Unknown
Persistence time	`[TIMING] _persist_adaptive_after_run` or PERSIST phase	0.03s (main.py), but orchestrator-internal unknown
Embedding API calls/turn	Count `httpx.post` to OpenAI embeddings endpoint	3+2M (estimated)
DB queries/turn	Count SQL executions in db.py per request	~20-30 estimated
LLM calls/turn	Count `gateway.generate()` and `openai.create` per request	2-12 estimated
Tool planner time	Time `plan_tool_invocation()` isolated	Unknown
Episode summary regen time	Time `_build_episode_summaries()` isolated	Unknown

Instrumentation Steps

1. **Add structured timing to orchestration.py**: Wrap each phase in `perf_counter()` pairs:
   - `CONTEXT_LOAD`, `CONTEXT_PACK`, `NARROW_ROUTE`, `FACET_DECOMPOSE`, `EXPERT_SELECT`, `TOOL_PLAN`, `EXECUTOR`, `POSTPROCESS`, `PERSIST_ADAPTIVE`
2. **Count external API calls**: Add counter in `EmbeddingService.embed()` and `ModelGateway.generate()` — log per-request totals.
3. **Count DB queries**: Add query counter in `db.py` cursor wrapper — log per-request total.

Test Protocol

Test	Purpose	Procedure
T1: 10-turn ML conversation	End-to-end regression	Same prompts as existing test. Measure total time, per-turn avg, per-turn variance.
T2: Single-turn simple question	Minimum latency floor	"What is 2+2?" — no tools, no retrieval. Target: <5s.
T3: Single-turn with tools	Tool planner overhead	"Search for recent Python security CVEs" — forces tool planning. Measure tool_plan + execute.
T4: Long conversation (20 turns)	Context scaling	20 turns of domain-specific Q&A. Track per-turn latency curve — should be flat, not increasing.
T5: Concurrent 3 users	Load behavior	3 parallel conversations. Track per-turn latency degradation.

A/B Comparison Framework

For each fix from the Top 10:
1. Run T1 (10-turn) as baseline
2. Apply fix
3. Run T1 again
4. Compare: `Δ = (baseline_avg - fix_avg) / baseline_avg × 100%`
5. Accept if: `Δ > 5%` AND no quality regression (all turns pass)

Part 4: Orchestration Pipeline Deep Dive

Request Lifecycle with LLM/DB/Embed Call Counts

```
HTTP POST /chat  →  main.py (6 phases)
│
├─ RECEIVE (0 LLM, 0 DB, 0 embed)
│  └─ Parse request, validate auth
│
├─ PRE_RESPONSE_GATE (0-1 LLM, 2-4 DB, 0-1 embed)
│  └─ recall_artifacts_for_prompt()  →  cosine classifier (no LLM)
│  └─ load_conversation_context()
│     ├─ _filter_run_log()           →  1 DB query, N×json.loads
│     ├─ _load_persistent_snapshot() →  1 DB query
│     ├─ _load_persistent_facts()    →  1 DB query
│     └─ _load_persistent_tasks()    →  1 DB query
│
├─ GENERATE  →  orchestration.py execute()
│  │
```

```
│  ├─ WorkingContextAssembler.build()  (0-5 LLM, 0 DB, 0 embed)
│  │  └─ ContextPacker.pack()
│  │     ├─ _rank_semantic_facts()     →  no LLM
│  │     ├─ _orch_transform() × 3-5   →  summarizer.summarize() IF text > limit
│  │     └─ _build_compacted_pack()    →  no LLM
│  │
│  ├─ Routing Decision
│  │  ├─ narrow_route()                →  0-1 LLM (default: 0)
│  │  ├─ FacetDecomposer.decompose()   →  0 LLM, 1 embed, 1-2 DB
│  │  └─ _find_best_expert() × facets  →  0 LLM, 2×F DB queries
│  │
│  ├─ Tool Planning (if tools needed)
│  │  └─ plan_tool_invocation()        →  1 LLM (mandatory)
│  │     └─ execute_tool() × planned   →  0 LLM (tools make own calls)
│  │
│  ├─ Primary Executor                 →  1 LLM (ModelGateway.generate)
│  │  └─ Cascade: miner_proxy → openai_direct → ollama (60s→120s timeouts)
│  │
│  ├─ Evaluation + Repair Loop (if output fails grade)
│  │  └─ Per retry: 1 LLM call         →  up to N retries
│  │
│  ├─ apply_postprocess_pipeline()     →  0 LLM (pure transforms)
│  │
│  └─ _persist_adaptive_after_run()    ← SYNCHRONOUS, BLOCKS RETURN
│     ├─ _llm_classify_domain()        →  1 LLM
│     ├─ persist_adaptive_records()
│     │  ├─ N step writes              →  N DB inserts
│     │  ├─ M artifact writes          →  M DB inserts
│     │  ├─ 2M artifact embeddings     →  2M embed API calls
│     │  ├─ 1 agent index embed        →  1 embed API call
│     │  ├─ 1 domain label embed       →  1 embed API call
│     │  └─ 1 chain variant embed      →  1 embed API call
│     └─ save_router_trace()           →  1 DB insert
│
├─ DELIVER (0 LLM, 1 DB, 0 embed)
│  └─ SSE stream to frontend
│
├─ PERSIST (0 LLM, 1-2 DB, 0-1 embed)
│  └─ embed_message() + save_message()
│
└─ READY
```

Per-Turn Cost Summary (typical case)

Resource	Minimum	Typical	Worst Case	Notes
LLM calls	2	4-5	12+	executor(1) + tool_planner(1) + summarizer(0-5) + domain(1) + retries(0-N)
DB queries	8	15-20	30+	context(4) + expert(2-6) + persistence(N+M+8) + traces(2-3)
Embedding API calls	3	5-7	3+2M	agent(1) + domain(1) + chain(1) + artifacts(2M) + message(1)
JSON serialize ops	10	25-50	100+	context load + cap_payload + DB layer + Redis cache

Highest-Value Cuts to This Pipeline

1. **Background** `_persist_adaptive_after_run()` → saves 1 LLM + N+M+8 DB + 3+2M embeds from blocking
2. **Cache episode summaries** → saves 0-5 LLM in `_orch_transform()`
3. **Heuristic-first tool planning** → saves 1 LLM on simple/repeat queries
4. **Single-embed artifacts** → saves M embed calls (deduplicate)
5. **Parallel context queries** → saves 3× DB round-trip wait

Conservative estimate applying fixes 1-5: 43.7s → **25-30s avg/turn** (~30-43% improvement)

Appendix: File Index

File	Role	Issues Found
`backend/agents/orchestration.py`	Core orchestrator, 5 execution paths	C3, M3
`backend/agents/adaptive_persistence.py`	Post-run learning & storage	C1, C5, H1, H7
`backend/agents/persistence.py`	Conversation context loading	C4, H4
`backend/agents/tools/__init__.py`	Tool planning & execution	C2, H3, H6, M7
`backend/agents/embeddings.py`	Embedding service (OpenAI)	H2
`backend/agents/expert_router.py`	Expert selection per facet	H5
`backend/agents/model_clients.py`	Model gateway, dispatch & fallback	M2
`backend/agents/db.py`	Database layer, SQL + JSON serde	M1, M4
`backend/agents/response_postprocess.py`	Post-LLM string transforms	M5
`backend/agents/narrow_router.py`	First-stage task classification	M6
`backend/agents/facet_decomposer.py`	Facet decomposition (embedding-based)	L2
`backend/agents/router.py`	Entry point, orchestrator builder	—
`backend/agents/tools/self_tool_builder.py`	Dynamic to	

[...content truncated for whitepaper synthesis...]

Docs: Task Orchestration Context Policy

Task Orchestration Context Policy

Single source of truth for context packing, planning inputs, budget allocation, and telemetry in the task-orchestrated execution path.

Implementation and tests should reference this document by path.

1. Purpose & Scope

This policy governs how the task-orchestrated pipeline builds, packs, and observes inter-node context. It applies to:

- `TaskExecutionService` (DAG execution, input context building)
- `DependencyContextPacker` (multi-dep budget allocation)
- `_build_planned_nodes` / `_expand_web_research_nodes_impl` (graph planning)
- `TaskCompilationService` (merge and failure manifest)
- `_task_tool_executor` (tool bundle narrowing, structured data extraction)

It does **not** govern `ChainExecutorService` or non-task tool loops.

2. Consumer Classification

Every edge between task nodes has a **next consumer type**:

Type	Reader	Strategy
`llm`	Model call / tool loop LLM	Summarize
`aggregator`	`TaskCompilationService`, merge	Chunk
`tool`	Direct tool input	Pass-through

Default: `llm` unless `next_consumer_type` is set in `PlannedTaskNode.metadata`.

3. Summarize vs Chunk Decision Table

Next consumer	Raw fits budget?	Action
LLM	Yes	Pass verbatim
LLM	No	Summarize via `SummarizerService`
Aggregator	Yes	Pass verbatim
Aggregator	No	Chunk with `chunk_index/total`
Tool	Yes	Pass verbatim
Tool	No	Truncate at boundary

4. Planning Inputs

Allowed for task-graph building:

- `_extract_rule_based` in `StructuredExtractor` (regex, keyword lists)
- Orchestration-level regex/heuristics (compound prompt split, list detection)
- `SummarizerService` only to shorten long prompts before regex passes

Not allowed for planning: `_extract_with_llm`, `_EXTRACTION_PROMPT`, or any LLM-based JSON extraction for minting graph nodes.

5. Budget & Metadata Schema

Additive keys on `PlannedTaskNode.metadata` (no dataclass change):

Key	Type	Default
`dependency_context_budget`	`int` (chars)	8000
`dependency_context_weights`	`{dep_id: float}`	equal (1.0)
`dependency_context_floor_chars`	`{dep_id: int}`	200
`next_consumer_type`	`str` enum	`"llm"`
`web_task_role`	`str`	—
`web_task_index`	`int`	—
`url_rank`	`int`	—

6. v1 Packing Algorithm

Single-shot after all dependencies complete (`DependencyContextPacker.pack`):

1. **Measure** `size_raw(d)` for each dep summary + structured_data.
2. **Allocate** proportional to `dependency_context_weights` (default equal).
3. **Apply floors** from `dependency_context_floor_chars` (default 200).
4. **Fit**: summarize for LLM, chunk for aggregator.
5. **Trim order**: ascending weight first, respect floors.

Reserve 20% of budget for structured_data keys (URLs, dates, evidence).

7. Failure / Partial Handling

- **Scheduler default**: run child nodes even when parent status = `failed` (`_parallel_groups` tracks completion, not success).
- **Packer**: insert `[dep_id]: (failed)` placeholder for failed deps. Redistribute freed budget to remaining active deps.
- **Compile**: produce `failure_manifest` list with `{node_id, tool_name, error_class, message, retryable}` for each failed node.

8. Settings Catalog

Setting	Default	Description
`ORCH_WEB_CONTEXT_FANOUT_K`	3	Context child count for web fan-out
`ORCH_TASK_NODE_TIME_BUDGET`	45	Per-node wall-clock seconds
`ORCH_TASK_NODE_MAX_ROUNDS`	2	Max tool-loop rounds per node
`ORCH_TOOL_OUTPUT_BUDGET_FRACTION`	0.5	Model context fraction for tools

9. Telemetry Contract

`build_task_telemetry` emits per node:

- Standard: `task_id`, `status`, `latency_ms`, `retry_count`, `error`, `quality_signals`, `output_confidence`
- Planning: `task_type`, `domain`, `planning_confidence`, `tool_hints`, `dependencies`
- Metadata: `web_task_role`, `web_task_index`, `url_rank`, `task_role`, `primary_tool`, `next_consumer_type`
- Binding: expert/model/miner selection details

---

10. tool_hints ↔ _TOOL_HINTS_NEEDING_TOOLS Contract

`_node_needs_tools` uses **exact-match membership** in frozenset.

Current set:

```
web_search, search, web, browse, fetch,
micro_web_extract_content, micro_extract_chart_data,
web_search_discover, web_content_reader, direct_url_lookup,
sql_query, rest_api_query,
knowledge_base_search, hybrid_retrieval,
generate_chart
```

Rule: every new `tool_hints` value emitted by `_build_planned_nodes` or the web expander must have a matching entry added to the frozenset in the same change, or routing silently falls back to `_task_model_executor`.

---

11. Observability for Degraded Completion

When any node fails:
1. `failure_manifest` is produced at compile time.
2. Warnings list includes `task_failed:{node_id}`.
3. The failure manifest can be injected into the model prompt for honest partial-answer generation.

---

12. Integration / Linked-Resource Context

- **SQL/REST**: only inject `sql_query` / `rest_api_query` hints when `_prompt_requires_linked_sql_lookup` / `_prompt_requires_linked_rest_api_lookup` returns True AND tenant has linked resources.

- **Integrations (OAuth)**: only add OAuth tools when the org has active, configured connections. Fail-closed: omit tool + record diagnostic.
- **Platform APIs**: subject to tenant web policy (`filter_tool_specs_for_web_policy`).

---

13. Web Artifact Dates & Recency

- `content_collected_at`: ISO 8601 UTC, set at fetch/normalization time. Required for every web-sourced item.
- `published_at`: ISO 8601 UTC or null. Extraction order: HTML `<meta article:published_time>`, JSON-LD `datePublished`, OpenGraph, HTTP `Last-Modified`.
- Packer treats date fields as non-droppable structured keys.

Docs: Memory Tier Model

Memory Tier Model

Overview

All conversation memory is organised into five tiers, ranked by context priority. The guiding principle is:

> **Raw data is canonical and never replaced by summaries. Summaries are derived artifacts only. Performance work must not convert authoritative memory into compressed heuristics.**

The 5 Tiers

Priority	Tier	Name	Lifespan	Compression	Rule
1 (highest)	T1 — RECENT	Raw conversation turns	Current request + N recent	Never compressed	Always full-fidelity. Recent N turns passed verbatim to model.
2	T3 — SEMANTIC	Extracted facts & constraints	Cross-conversation	Never compressed in storage	Structured facts extracted from raw T1. Inviolable in storage; prompt inclusion is relevance-ranked and budgeted (top-K).
3	T2 — EPISODIC	Summarised conversation segments	Conversation lifetime	Lossy (derived from T1)	Compressed narrative for older history. Source T1 always retained.
4	T4 — TRACE	Execution telemetry	Permanent (audit)	Compactable for display	Raw payloads canonical; display views may be compressed.
5 (lowest)	T5 — KNOWLEDGE	Static KB artifacts & embeddings	Permanent	Canonical in storage; may be excerpted for prompt	Prompt views are derived, provenance-preserving, non-authoritative. Retrieved by similarity on demand.

Key hierarchy change: T3 (facts) sits ABOVE T2 (summaries). Facts are compact, high-value, and survive better across long conversations than lossy summaries. When budget is tight, facts must be preserved before summaries.

Dual Policies

T3 — SEMANTIC

Storage policy:
- Preserve all valid facts canonically — never overwrite with summaries.
- Facts can be deprecated (`status='superseded'`) but never silently deleted.
- All facts retain full provenance (`source_message_id` , `source_span`).

Prompt policy:
- Include top-K relevant facts (not all facts unconditionally).
- Rank by weighted score: prompt relevance > confidence > recency > permanence > scope match.
- Budget controlled by `ORCH_CONTEXT_FACT_TOP_K` .

T5 — KNOWLEDGE

Storage policy:
- Canonical. `raw_content` and `normalized_content` are never overwritten or compressed.

Prompt policy:
- May be excerpted or compressed for prompt inclusion.
- Prompt views are derived, provenance-preserving (linked via `source_ref` , `created_from_run_id`), and non-authoritative.

Safety Rules

1. **T1 is never overwritten by T2.** Summaries supplement raw turns; they do not replace them in storage.
2. **T3 is additive-only.** Facts can be added or deprecated, never silently dropped.
3. **T3 must be extracted from raw T1 data, never from T2 summaries.** Pipeline: `raw messages/tool outputs → fact extraction → fact storage → fact retrieval` .
4. **T2/T4 compression is display-only.** The raw source (T1/T4 payload) must remain queryable.
5. **Every derived artifact must link back to source records** via conversation_id + turn range or trace_id.
6. **Extractive summarisation (B2) is only allowed on T4 trace display contexts.**

Fact Lifecycle

Status values

Status	Meaning
`active`	Confirmed, high-confidence fact
`tentative`	Extracted but not yet confirmed by repeated observation
`superseded`	Replaced by a newer fact (link via `superseded_by`)
`expired`	Time-bound fact past its relevance window

Scope values (`fact_scope`)

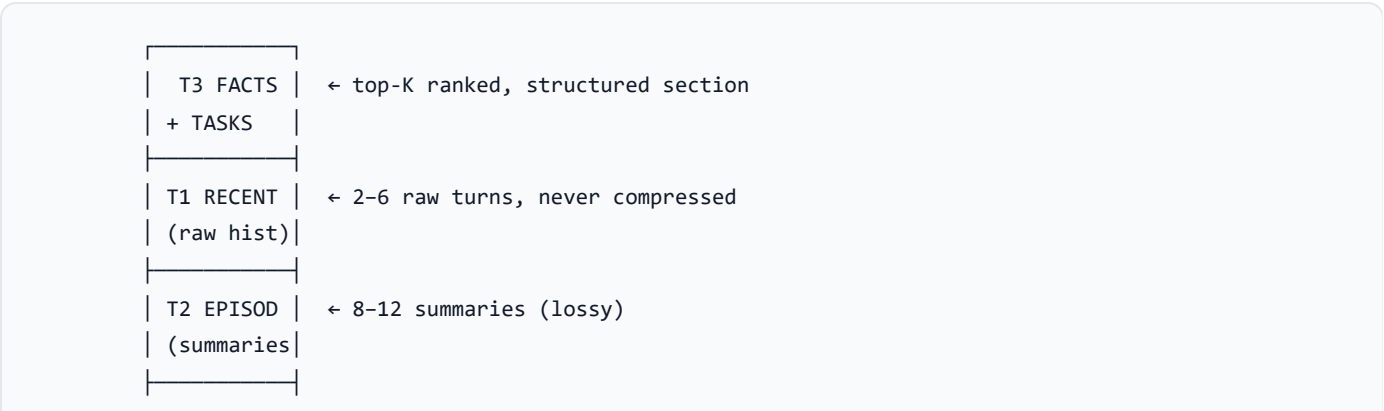
Scope	Retrieval breadth
`conversation`	Only for the originating conversation
`user`	Across all conversations for the user
`project`	All conversations in the project
`global`	All conversations

Contradiction handling

When a new fact contradicts an existing one (same `fact_key`, different value), the old fact is marked `superseded` with `superseded_by` pointing to the new fact's ID and `superseded_at` set. The superseded fact remains queryable for audit.

Context Pyramid

Assembly order (pyramid top = highest priority):



```
          │ T5 KNOWL  │  ← injected by similarity, outside budget
          │ (KB/embed)│
          ├───────────┤
          │ T4 TRACE  │  ← only for debug/self-repair
          │ (telemetr)│
          └───────────┘
```

Prompt assembly order

1. Current user prompt
2. Recent raw turns (T1 — highest fidelity)
3. Relevant semantic facts (T3 — compact, durable)
4. Relevant tasks / goals / preferences (T3 — active tasks)
5. Episodic summaries for older history (T2 — lossy, continuity)
6. Retrieved documents / artifacts (T5 — on demand)
7. Trace hints (T4 — rarely, only for debug/repair)

Code Mapping

Tier	DB Tables	WorkingContext Field	ContextPacker Treatment
T1	`messages`, `run_log.payload_json`	`history`	`history[-recent_limit:]` — tier-adjusted window
T3	`conversation_facts`, `conversation_tasks`	`semantic_facts`, `pinned_facts`	Top-K by ranked score; tasks in separate section
T2	`conversation_memory_snapshots`	`episode_summaries`	`episode_summaries[-8..12:]` — tier-adjusted count
T4	`router_traces`, `step_traces`, `model_pair_traces`	`compaction`, `working_memory.context_compaction`	CompactedContextPack
T5	`knowledge_artifacts`, `message_embeddings`	`working_memory["recalled_artifacts"]`	Outside packer budget — injected via retrieval

---

Source Traceability

Traceability is a **test invariant**, not just a design principle:

- Every `conversation_facts` row must have a non-null `source_message_id`.
- Every `conversation_memory_snapshots` row must have a source range or source IDs.
- Every fact packed into the prompt by `ContextPacker.pack()` must be traceable to a DB record.

Context Labels → Tier Classification

See `backend/agents/memory_tiers.py` for the authoritative `CONTEXT_LABEL_TIER` mapping.

Docs: Miner Pools and BYOK

Miner Pools, BYOK, and Dispatch Flows

This document describes how the miner dispatch system works: what pools exist, how BYOK (Bring Your Own Key) settings function for users and organizations, and how all of these interact during a chat or API request.

What Is a Miner?

A **miner** is a compute endpoint that handles model inference requests. There are two physical kinds:

- **Real miners** — machines running the Elis miner software. They maintain an active session and heartbeat with the platform, and receive jobs over a WebSocket dispatch channel.
- **Virtual miners** — database records that route a request directly to a provider API (OpenAI, Anthropic, Google, etc.) using an encrypted API key stored by the platform. BYOK miners are always virtual.

Miners are stored in the `miners` table. The key identity fields are:

Field	Description
`id`	Unique miner ID — prefix signals pool type (see below)
`organization_id`	Set for org-dedicated real miners; NULL for community miners
`owner_user_id`	User who registered the miner (BYOK) or NULL for real miners
`reserved_owner_user_id`	When set, only requests from this user can select this miner
`trust_tier`	`community` | `enterprise` | `internal` — affects scheduling priority
`reliability_state`	`healthy` | `degraded` | `suppressed` — set by failure tracking
`selection_suppressed_until`	Timestamp; miner is skipped until this time elapses

Miner Pools

There are four distinct pool types, identifiable by the miner ID prefix and `organization_id`:

1. Community Pool

- **ID prefix:** none (e.g. `a3f9b2...`)

- `organization_id`: NULL
- **Who can access:** everyone — personal users and all orgs (unless the org disables it)
- **Trust tier:** `community`

Community miners are physical machines run by volunteers or infrastructure partners. They are globally shared. Any request that allows community miners can land on any of these. They sort last behind BYOK virtual miners in the scheduler and are filtered by `reserved_owner_user_id = NULL`.

2. User BYOK Pool (`byok-`)

- **ID prefix:** `byok-`
- `organization_id`: NULL
- `owner_user_id`: the registering user
- **Who can access:** depends on `is_shared` (see below)

Virtual miners that call a provider API using the individual user's encrypted key.

3. Org BYOK Pool (`orgbyok-`)

- **ID prefix:** `orgbyok-`
- `organization_id`: the owning org
- **Who can access:** depends on `is_shared` (see below)

Virtual miners that call a provider API using the org's encrypted key. Earnings settle to the org wallet via `payout_company_id`.

4. Org Dedicated Pool

- **ID prefix:** none (e.g. `d7c1e4...`)
- `organization_id`: set to the owning org
- **Who can access:** that org exclusively
- **Trust tier:** `enterprise` or `internal`

Physical machines running the Elis miner software, provisioned and operated by the organization. These are the only miners that count as "Dedicated Miners" for billing and compliance purposes.

User BYOK Settings

Enabling BYOK

BYOK is **off by default** and requires explicit opt-in. The setting lives in `user_preferences.byok_enabled`. When enabled, the platform surfaces the BYOK management UI and `/miners` tab.

Registering a BYOK Miner

A user registers a miner via `POST /api/miners/byok/register`, providing:
- `provider` — `openai` | `anthropic` | `google` | `groq` | `mistral` | `cerebras`
- `model_name` — the specific model this key will serve (e.g. `gpt-4o`, `claude-opus-4-6`)
- `api_key` — stored AES-GCM encrypted, never returned in responses
- `label` — optional display name
- `is_shared` — whether to contribute to the community pool (default `false`)

The `is_shared` Toggle (Share / Private)

Each user BYOK miner has an `is_shared` flag that can be toggled at any time via `PATCH /api/miners/byok/<id>`.

Shared mode (`is_shared = true`)
- `trust_tier` → `community`
- `reserved_owner_user_id` → cleared (empty)
- The miner enters the global community pool and is selectable for any user's requests
- The owner earns credits for every completed job that routes through their key
- Useful for contributing capacity in exchange for platform credits

Private mode (`is_shared = false`)
- `trust_tier` → `enterprise`
- `reserved_owner_user_id` → set to the owner's user ID
- Only requests from the key owner can select this miner
- No credit payouts (the user is consuming their own key)
- Useful for cost control — all your requests use your own key instead of the platform's

Disabled (deactivated)
- `is_active = false` — set automatically when the key returns a 401 or 403 from the provider
- `key_error` — stores the reason string from the failed response
- The miner stops accepting jobs; the UI shows the deactivation reason
- Reactivate by updating the key via `PATCH /api/miners/byok/<id>` with a fresh `api_key`

`byok_only_mode`

`user_preferences.byok_only_mode` is a per-user escalation flag. When enabled:
- Requests from this user are routed **exclusively** through their own BYOK miners
- The community pool and platform API keys are bypassed entirely
- Useful for users who need all inference to use their own provider account (cost attribution, compliance, etc.)
- Only meaningful when the user has at least one active private BYOK miner registered; otherwise requests will fail

---

Org BYOK Settings

Registering an Org BYOK Miner

Organization admins register org BYOK miners via `POST /api/companies/<id>/miners/byok/register`. The API and fields are identical to user BYOK, but:
- Miner ID gets an `orgbyok-` prefix
- The key is stored in `company_miner_api_keys` (separate table from `user_miner_api_keys`)
- Earnings settle to the org wallet via `miners.payout_company_id`

The `is_shared` Toggle for Org BYOK

Behaves identically to user BYOK sharing:

Shared (`is_shared = true`) — the org's key joins the community pool. Other tenants' requests can route through it. The org earns credits.

Private (`is_shared = false`) — scoped to `organization_id`. Only requests from members of this org can use it. The org is consuming its own key for its own members.

> **Important:** An org BYOK miner is a **virtual miner** — it calls a provider API using your key. It is not the same as a Dedicated Miner. Org BYOK is available on all plans. Dedicated Miners require running the Elis miner software on your own hardware or cloud VMs.

Org-Level Pool Settings (`companies.settings`)

Two org settings control which pools requests from this org may use:

`allow_community_miners` (default: `true`)

When `true`, org members' requests can be served by community miners (and the platform's own API keys) in addition to the org's own miners.

When `false`, requests are locked to the org's own pool (dedicated + org BYOK miners). If no org miners are online for the requested model, the request fails rather than falling back to community

capacity.

`dedicated_miners_only` **(default:** `false`**)**

When `true` (also triggered by `deployment_type = "dedicated"` or `deployment_mode = "onsite"`):
- Only the org's dedicated real miners may serve requests
- Org BYOK miners are excluded
- Community miners are excluded
- Platform API keys are excluded
- If no dedicated miner is online for the model, the request returns `dedicated_miner_unavailable` immediately

This is the strictest enforcement mode — it guarantees inference stays on the org's own managed hardware.

Dispatch Flow — Personal Chat (No Org)

A personal chat has `user_id` and `conversation_id` but no `company_id`. The dispatch logic:

1. `company_id` is empty → `_effective_allow_community_miners("")` returns `true`
2. `effective_dedicated_miners_only` is `false`
3. The scheduler queries `list_schedulable_miners(model=..., organization_id="")` which returns community miners only (`organization_id IS NULL`)
4. If a community miner is available for the requested model → dispatched via WebSocket
5. If no community miner is available → falls through to the platform's direct API key for that provider
6. If the user has `byok_only_mode = true` → only their private BYOK miners are eligible; falls through to those instead

Pool available to a personal chat: community miners + user's own private BYOK miners (if `byok_enabled`) + platform API key fallback.

Dispatch Flow — Org Chat

An org chat has `company_id` set in the run packet (injected by `_packet_for_miner()` in orchestration).

1. `_effective_dedicated_miners_only(company_id)` — resolves from `companies.settings`
2. `_effective_allow_community_miners(company_id)` — resolves from `companies.settings`

3. Scheduler queries `list_schedulable_miners(model=..., organization_id=company_id)` which returns:
   - Community miners (`organization_id IS NULL`)
   - This org's own miners (`organization_id = company_id`)
4. The scope check (`_has_accessible_miner_scope`) then filters:
   - If `dedicated_miners_only=true` → only accept miners where `miner.organization_id == company_id`
   - If `allow_community_miners=false` → reject any miner with `miner.organization_id = NULL`
5. If the scope check passes and a miner is dispatched → done
6. If `dedicated_miners_only=true` and no miner dispatched → `dedicated_miner_unavailable` error
7. If `allow_community_miners=false` and scope check fails → `no_accessible_miners` error
8. If `allow_community_miners=true` and no miner dispatched → falls through to platform API key

Pool available to an org chat depends on settings — see the matrix below.

Settings Interaction Matrix

The table below shows what pool is available for a request given the combination of org settings. "Platform API" means the platform's own provider API keys (always available as a last resort unless locked out by policy).

`dedicated_miners_only`	`allow_community_miners`	Org dedicated miners	Org BYOK miners	Community miners	Platform API
`false`	`true` (default)	✅	✅	✅	✅ fallback
`false`	`false`	✅	✅	❌	❌
`true`	any	✅ only	❌	❌	❌

Community-only org (no dedicated, no BYOK, allow_community=true)
Requests go to community miners. If none online for the model, the platform API key handles it. This is the default for new orgs.

Closed-pool org (allow_community=false, dedicated_miners_only=false)
Requests stay within the org's own miners (dedicated + org BYOK). If none available, the request fails. No community exposure.

Fully dedicated org (dedicated_miners_only=true)
Requests must land on the org's own physical dedicated miners. No BYOK, no community, no platform API. Hard fail if no miner online.

Org with BYOK + community allowed (default + org BYOK registered)
Org BYOK miners are preferred (sorted before community by scheduler). Community miners fill in when no org BYOK miner is online. Platform API is final fallback.

Miner Reliability States

The scheduler tracks failures and updates `miners.reliability_state`:

State	Meaning	Behaviour
`healthy`	No recent failures	Eligible for dispatch normally
`degraded`	Elevated failure rate	Still eligible but logged; UI shows warning
`suppressed`	Too many consecutive failures	Skipped by scheduler until `selection_suppressed_until` elapses, then auto-recovers

When a BYOK key returns a 401 or 403 from the provider, the miner is deactivated (`is_active = false`) and the `key_error` field is written with the provider's error message. The UI surfaces this reason on the miner card and detail page. The miner stays deactivated until the admin rotates the key.

Narrator vs Chat Dispatch

The **narrator** (background task generation) builds a descriptor with an empty miner ID (`provider:model#`) and loops through multiple model candidates. It never hits the `_has_accessible_miner_scope` hard-fail because it is not dispatched through the miner WebSocket path — it goes directly to provider API.

Chat completions go through the full miner dispatch path: WebSocket first, then the scope check, then platform API fallback. This is why a personal chat could historically show "no organization miners are currently online" even while narrator responses succeeded — the scope check would hard-fail before reaching the platform API fallback if no community miner happened to be online for the exact model. That path is now fixed to fall through when `allow_community_miners=true`.

Key Tables

Table	Purpose
`miners`	All miner records — real and virtual
`miner_sessions`	Active WebSocket sessions for real miners
`user_miner_api_keys`	User BYOK key storage (encrypted)
`company_miner_api_keys`	Org BYOK key storage (encrypted)
`user_preferences`	Per-user flags: `byok_enabled`, `byok_only_mode`
`companies`	Org settings including `allow_community_miners`, `dedicated_miners_only`
`company_miner_daily_usage`	Per-org BYOK token usage by day for quota tracking

Here’s a cleaner logic-chain version:

```
For each setting, validate scope first:
- User settings apply only to personal chats.
- Org settings apply only to chats inside that organization.
- Org settings must not affect personal chats.

BYOK mode OFF:
- For users: personal chats can access community miners.
- For orgs: org chats can access community miners.

BYOK mode ON + private/share-restricted mode:
- For users: only that user can access their BYOK miners.
- For orgs: only that org can access its BYOK miners.

BYOK mode ON + community share mode:
- For users: the user’s BYOK miners are added to the community pool.
- For orgs: the org’s BYOK miners are added to the community pool.
- Once added to the community pool, everyone can use those miners.

BYOK-only mode:
- For users: personal chats must only use that user’s BYOK miners.
- For orgs: org chats must only use that org’s BYOK miners.

Scope enforcement:
- Org miner settings are only valid inside that organization’s chats.
- Org miner settings cannot impose behavior on personal chats.
- Personal chats must use the user’s own miner settings.
```

Site: /docs

Documentation

Learn how the Elis AI distributed intelligence network processes your requests.

How Elis AI Works

Elis AI is a distributed intelligence network that breaks complex questions into smaller, focused tasks and routes them to specialized AI agents running across a decentralized pool of compute miners. Instead of relying on a single large model, every request passes through a dynamic pipeline that picks the best-fit agent, dispatches inference to the best available miner, and returns a fully traced answer — so you always know how a response was produced.

🤖 Elis AI Mode

Automatically builds and reuses specialized agents. Each agent runs a multi-step workflow — expertise lookup, context retrieval, execution — using the best available model tier for each step.

🔓 Elis Unlocked

Advanced multi-pass workflow with planning, fact-checking, and iterative refinement. Best for complex tasks that need verification and deep research.

⛏️ Miners

Miners host local LLMs and earn tokens for verified responses. The network supports multiple model tiers for speed and quality tradeoffs.

🛡️ Verification

Every inference step is traced end-to-end. The scheduler selects the best available miner using weighted round-robin, trust tiers, and load awareness — ensuring fair distribution and reliable execution.

Why the network stays sustainable

Elis is built on a token economy where compute contributors are rewarded for useful work. Miners that return high-quality results quickly earn more assignments. Because miners compete regionally, users get served by nearby high-performing nodes — improving latency and experience.

For Consumers

Contribute background compute to earn tokens that offset usage costs.

Background contribution earns tokens.

Tokens are spent on model and tool usage.

Quality verification keeps rewards fair.

For Enterprise

End-to-end encryption, automated PII redaction, audit trails, and self-hosted deployment options. See Enterprise section →

Process Flow

1. User submits prompt → 2. Router chain selects agent → 3. Agent builds workflow → 4. Miners run inference → 5. Quality verification → 6. Traced response returned

Ready to try it?

Create an account and start asking questions — every answer comes with a full trace.

Get Started → View Whitepapers

Site: /tutorials

Guided Walkthroughs
Tutorials shaped like the landing page, but built for real product work.
Step-by-step guides to help you get the most out of Elis AI. Choose the track that matches how you use the platform.
User Guide
For everyone. Learn how to use the platform day to day, set up teams, and manage the resources tied to your workspace.
Sign Up & Log In
Create your account and access the platform.
Open
Starting a Conversation
Send your first prompt and see AI responses stream in.
Open
Document Editor
Writer documents (50 cap), AI rewrite & selection tools, links, merge, history, and side chat.
Open
Managing Messages
Edit, resend, copy, and delete messages.
Open
Creating an Organization
Set up an organization to collaborate with your team.
Open
Managing Members
Invite team members, set roles, and manage access.
Open
Uploading Documents
Upload files for your organization's knowledge base.
Open
Connecting Databases
Link an external database for context-aware queries.
Open
Creating & Managing Projects
Organize work into projects with linked resources.
Open
Viewing the Audit Log
Review all activity within your organization.
Open

Settings & API Keys
Manage your profile and create API keys.
Open
Billing & Tokens
Purchase token packages and view transaction history.
Open
Subscription Management
Cancel or restore personal and organization subscriptions before access changes take effect.
Open
Organization Seat Management
Lower seat blocks safely by choosing which members or pending invites lose access.
Open
Browsing Models
Explore the model directory and compare options.
Open
Accepting Invitations
View and respond to pending organization invites.
Open
Project Conversations
Create and manage conversations within a project context.
Open
Using Dashboards
Create dashboards, iterate drafts, publish versions, and manage favorites.
Open
AI Edit a Dashboard
Edit dashboards in plain English with a live preview, clarify questions, and per-message restore points.
Open
Dashboard Editor Deep Dive
Understand each editor section: planning, AI spec edits, tasks, buckets, datasources, draft thread, and assistants.
Open
Using Automations
Organization-only recurring widgets with mobile support, chart modes, and alert dispatch.
Open
Deep Data Collection
Crawl websites, extract structured records, share public datasets, and earn credits.
Open
Enterprise & Dedicated Hosting

Enterprise
Dedicated infrastructure, model governance, BYOK/BYOA credentials, organization-scoped miners, and lifecycle operations for enterprise deployments.
Dedicated Hosting Upgrade
Activate dedicated deployment, select shared vs. dedicated at org creation, and monitor cluster provisioning lifecycle.
Open
Dedicated User Management
Invite members with cluster-aware links, review pending signup invites, and adjust paid seat capacity without losing track of reserved seats.
Open
Enterprise Controls
Configure model allow/block lists, web domain allowlist, BYOK API keys, and BYOA OAuth credentials.
Open
Provider API Keys (BYOK)
Sign up with OpenAI, Anthropic, Google, Groq, Mistral, and Cerebras, then store each provider key on your organization.
Open
Dedicated Miners
Assign organization-scoped miners, generate registration tokens, and enforce dedicated-only inference.
Open
OAuth & SSO Setup
Configure BYOA OAuth credentials and optional SSO enforcement for enterprise cluster members.
Open
Dedicated Lifecycle Operations
Operate Stripe webhook lifecycle, grace mode, data export, and cluster decommission procedures.
Open
Developer Guide
For admins and developers. Go deeper on miner management, API integration, observability, and operator tooling.
On-Prem Installation
Purchase the annual license, download the Docker bundle, install dependencies, and activate the platform.
Open
Setting Up a Miner
Hosted or self-hosted: BACKEND_URL, MODEL_NAME, pairing key, Docker or Windows.
Open

Managing Miners

Monitor health, filter workers, enable/disable, validate pairing, unclaim when needed.

Open

Using the API

Authenticate and make programmatic requests.

Open

Automations API

Create, run, and monitor recurring research widgets via REST.

Open

Conversation Explorer

Browse and filter conversations with run telemetry.

Open

Agent Directory & Detail

Search agents, view stats, and explore mutations.

Open

Runs Dashboard & Comparison

Filter runs and compare them side by side.

Open

Run Detail Deep Dive

Inspect Overview, Routing, Steps, Context, and Integrity tabs.

Open

Anomaly Dashboard

Filter by anomaly type and severity, then drill into flagged runs.

Open

Downloads

Access compiled packages and Docker images.

Open