ELI — Complete Technical Manual

ELI v2.1.14

July 2026

Contents

ELI — The Complete User Manual **5**

Table of contents . . . 5
1. What ELI is (in one minute) . . . 6
2. How ELI works when you ask it something . . . 6
3. Getting started . . . 7
4. Your first hour with ELI . . . 8
5. Talking to ELI . . . 9
6. A tour of the window . . . 9
7. What ELI can do — by everyday task . . . 10
8. ELI remembers you (memory) . . . 12
9. ELI learns your routines (habits) . . . 12
10. ELI helps before you ask (proactive) . . . 13
11. Voice, wake word, and dictation . . . 13
12. Seeing your screen and images (vision) . . . 13
13. The ELI server & using it from your phone . . . 14
14. Models: picking, switching, and training your own . . . 16
15. How hard ELI thinks (reasoning modes) . . . 17
16. Settings you'll actually touch . . . 17
17. Privacy & staying offline . . . 17
18. Troubleshooting . . . 17
19. Quick phrasebook (cheat sheet) . . . 18
20. Glossary . . . 18
21. Appendix A — Under the hood: how ELI thinks (the full pipeline) . . . 19
22. Appendix B — Under the hood: how ELI remembers (the full memory system) . . . 21
23. Appendix C — Under the hood: how ELI stays safe and yours (security) . . . 23

Blueprint — ELI MKXI Architecture **24**

0. Design principles (the non-negotiables) . . . 25
1. Repository map (recursive LOC, role) . . . 25
2. Entry points & launch . . . 26
3. The request lifecycle (the spine) . . . 27
4. Routing layer (`eli/execution`) . . . 27
5–6. Agent Bus & the agents (`eli/cognition/agent_bus.py`) . . . 28
7. The Orchestrator — full 12-stage (`eli/cognition/orchestrator.py`) . . . 28
8. Inference layer (`eli/cognition`) . . . 29
9. Grounding & anti-confabulation spine (`eli/runtime` + `eli/core/grounding.py`) . . . 29
10. Execution layer (`eli/execution/executor_enhanced.py`) . . . 29
11. Memory subsystem (`eli/memory`) . . . 30
12. Perception (`eli/perception`) . . . 30
13. Persona system (`eli/cognition/persona*`) . . . 30

14. Daemons & background loops . . . 31
15. Learning / self-training (`eli/learning`) . . . 31
16. EliWorld (`eli/world`, `eli/gui` EliWorld tab, `kernel/world_model.py`) . . . 31
17. Core infrastructure (`eli/core`) . . . 31
18. GUI (`eli/gui`) . . . 31
19. On-disk layout (`artifacts/`, runtime) . . . 32
20. Known seams & weak points (honest) . . . 32
21. Where to make a change (quick index) . . . 32

DAG orchestrator — project-wide execution engine (2026-06-07) 35
The orchestrator (`eli/core/dag.py`) . . . 35
Wired through the pipeline . . . 36
Tests . . . 36
Test/project run → review (GUI + chat) . . . 36
Generative actions on the coding agent / planner DAG . . . 36
SELF_IMPROVE routed through the coding agent . . . 37
Multi-command + timed commands (2026-06-08) . . . 37

ELI Orchestration & Agents — Full Topology 37
Two agent stacks (this is the key thing to understand) . . . 37
Planning artifacts — 4 of them, only 1 drives execution . . . 38
Where it is actually weak (corrected) . . . 39
Recommended fixes (prioritized) . . . 39
Update — 2026-06-09 . . . 40

ELI Agent Algorithms — what each of the 15 (+1) agents actually computes 40
Roles at a glance . . . 41
The 15 specialist algorithms . . . 41
How the DAG sequences the RAG (your "search → summarize → check") . . . 43
Aspirational next steps (genuinely novel combinations) . . . 43

ELI Memory Subsystem 44
Files . . . 44
The `Memory` class (`memory.py`) . . . 44
Schema (24 tables) . . . 44
`recall_memory` — the hybrid retriever (memory.py:1722) . . . 44
Vector store (`vector_store.py`) . . . 45
Knowledge graph (`knowledge_graph.py`) . . . 45
Truth layer (`memory_truth.py`) . . . 45
Weight decay & consolidation . . . 45
Honest assessment . . . 45
Update — 2026-06-09 . . . 46

ELI Perception — Vision, Voice, Wake Word, Tone, OS Control 46
Files . . . 46
Vision (`vision.py` + `ambient_vision.py` + `analyze_image.py`) . . . 46
Speech-to-text (`audio_stt.py`, `local_whisper_stt.py`) . . . 47
Wake word (`wakeword.py`, 2026-06-08) — self-trained, robust over music . . . 47
Voice profile + tone (`voice_profile.py`, 2026-06-08) — foundation for emotion . . . 47
Text-to-speech (`tts_router.py`) . . . 48
OS control (`os_controller.py`) . . . 48
File analysers . . . 48
Honest assessment . . . 48
Update — 2026-06-09 (VRAM-aware STT; drag-drop keeps the full path) . . . 48

ELI Grounding & Evidence Layer 49

The core idea . . . 49
Components . . . 49
How it fits the pipeline . . . 50
Honest assessment . . . 50
Update — 2026-06-09 . . . 51
Update — 2026-06-09 (deterministic reports surfaced verbatim; examiner correctness) . . . 51

Blueprint — Code-Mode Execution Layer for ELI (gap analysis) 51
BUILT (2026-06-11) . . . 51
0. Correction up front . . . 52
1. What ELI ALREADY HAS (audited, with files) . . . 52
2. The genuine DELTA (the only new work) . . . 52
3. Phased plan (composing what exists) . . . 53
4. Verified in full (2026-06-11) . . . 53
5. What stays uniquely ELI . . . 54

ELI Coding Agent (`eli/coding/`) 54
The loop . . . 54
Components . . . 54
Capability checklist (as requested) . . . 55
Wiring & control . . . 56
Honest scope (what's verified vs model-dependent) . . . 56

ELI Inference & Hardware Boot 56
Inference (`cognition/gguf_inference.py`, 2.7k LOC + `inference_broker.py`) . . . 56
Hardware profiling (`core/hardware_profile.py`, 1040 LOC) . . . 57
Boot optimizer (`core/startup_hardware_optimizer.py`, 599 LOC) . . . 57
Settings (`core/runtime_settings.py`, 1096 LOC) . . . 57
Paths (`core/paths.py`, 551 LOC) . . . 57
Honest assessment . . . 57
Update — 2026-06-09 (STT no longer starves the main model) . . . 58

Adaptive Inference Governor — Plan 58
0. The one-sentence problem . . . 58
1. What ELI already does dynamically (verified — do NOT rebuild) . . . 58
2. The gaps (each verified, with code + log evidence) . . . 59
3. Design — the Adaptive Inference Governor . . . 61
4. Cross-machine behaviour matrix (what "done" looks like) . . . 62
5. Phased implementation plan . . . 62
6. Verification strategy (must stay model/hardware-agnostic) . . . 63
7. Risks & non-goals . . . 63
8. Appendix — exact references (commit `a641471`) . . . 63

ELI Learning — LoRA Self-Training Pipeline 64
Files & the pipeline . . . 64
What it is — and isn't . . . 65
Honest assessment . . . 65

LoRA fine-tuning — audit & pipeline (2026-06-07) 65
What was found (audit) . . . 65
What was wired/fixed . . . 66
Still phi-profiled (recorded, your call) . . . 66

ELI Background Tasks & Unified Code Generation 66
1. Unified code generation . . . 66
2. Background task manager (`eli/runtime/background_tasks.py`) . . . 67

Control summary . . . 67
Scope / honesty . . . 68

ELI Runtime Surfaces, Planning, World, Tools & Plugins **68**
Runtime surfaces (`eli/runtime/`, the response/introspection group) . . . 68
Planning / proactive (`eli/planning/`, 3.9k LOC) . . . 68
World / autonomy model (`eli/world/`, 1.5k LOC) . . . 69
Tools (`eli/tools/`, 8.9k LOC) . . . 69
Plugins (`eli/plugins/`, 1.9k LOC) . . . 69
Honest assessment . . . 69

Blueprint — Operational Realities **69**
1. Startup / cold-start latency . . . 70
2. Resource contention (daemons vs interactive) . . . 70
3. Memory limits & eviction (two distinct policies) . . . 70
4. Failure recovery — degradation, not rollback . . . 70
5. Testing strategy (two layers + the gaps) . . . 71
6. Recommended cheap, high-value fixes (grounded in §1–§2) . . . 71
7. Runbook B — multi-GPU: run a 405B / 671B / ~1T MoE . . . 74
8. Privacy and ethos at scale . . . 74
9. The hardware is coming to you . . . 75
10. Cost cheat-sheet and "see it big this weekend" checklist . . . 75

ELI Security Posture **75**
1. Shell command gate (`runtime/security.py` `SecurityManager`) . . . 75
2. Filesystem & app gates (`SecurityManager`) . . . 75
3. Prompt-injection guard (`kernel/engine.py`) . . . 76
4. SQL identifier validation (`memory/memory.py`) . . . 76
5. Custom-agent trust registry (`cognition/agent_bus.py`) . . . 76
6. Code generation & self-modification — three paths . . . 76
Honest assessment . . . 77
Update — 2026-06-09 (RUN_CMD terminal is real; mock leak fenced) . . . 77
Web app & multi-user security (2026-06-28) . . . 77
Update — 2026-07-02 (stable phone token, admin-gated model switch) . . . 78
Update — 2026-07-03 (token moved to the URL fragment; test reds cleared) . . . 79

Blueprint — ELI MKXI ASCII Diagrams **79**
1. Full pipeline (request lifecycle) . . . 79
2. Memory subsystem `eli/memory` + `cognition/working_memory.py` . . . 80
3. Gating stack (in path order — every guard the request passes) . . . 81
4. One-screen overview (how it all connects) . . . 82

ELI — Capabilities & Actions (with activation phrases) **82**
Conversation & persona . . . 82
App & window control . . . 83
Input control . . . 84
Media . . . 84
Vision & screen . . . 85
Gaze (eye-tracking) . . . 85
Voice . . . 85
Files & notes . . . 86
Generation (docs/code) . . . 87
System status & info . . . 87
Grounded introspection . . . 88
Memory & profile . . . 89
Self-maintenance . . . 89

Tasks, time & planning . . . 90
Proactive . . . 90
Plugins . . . 91
Shell & advanced . . . 91
New / undocumented (add a phrase above) . . . 91
Aliases & internal actions (not directly user-triggered) . . . 91
Coverage . . . 92

ELI — The Complete User Manual

Plain-English Edition · with flowcharts

A friendly, no-jargon guide to everything ELI can do and how to make it do it.

This manual assumes **zero** technical background. You talk to ELI in normal English — by **typing** in the chat box or **speaking** — and it does things on your computer, answers questions, remembers what matters to you, and learns your routines. Everything runs **on your own machine**; nothing is sent to the cloud.

> **The one rule worth knowing:** ELI is **offline by default**. It only touches the internet when *you* ask for something that needs it (a web search, the news, downloading a model). Your conversations, files, and memories never leave your computer.

How to read this manual: - **Just want to use it?** Read §1–§5 and the cheat sheet (§18). That's enough to be productive. - **Want the whole picture?** Read straight through — every feature is here, in plain terms. - **/ / tags** at the start of some sections mean *anyone / a little technical / advanced*.

Table of contents

1. What ELI is (in one minute)
2. How ELI works when you ask it something
3. Getting started
4. Your first hour with ELI
5. Talking to ELI
6. A tour of the window
7. What ELI can do — by everyday task
8. ELI remembers you (memory)
9. ELI learns your routines (habits)
10. ELI helps before you ask (proactive)
11. Voice, wake word, and dictation
12. Seeing your screen and images (vision)
13. The ELI server & using it from your phone
14. Models: picking, switching, and training your own
15. How hard ELI thinks (reasoning modes)
16. Settings you'll actually touch
17. Privacy & staying offline
18. Troubleshooting
19. Quick phrasebook (cheat sheet)
20. Glossary
21. Appendix A — Under the hood: how ELI thinks (the full pipeline)
22. Appendix B — Under the hood: how ELI remembers (the full memory system)
23. Appendix C — Under the hood: how ELI stays safe and yours (security)

---

1. What ELI is (in one minute)

ELI is a **personal AI assistant that lives on your computer**. Think of it as a capable helper that can:

- **Talk with you** like a knowledgeable person — by text or voice.
- **Operate your computer** — open apps, play music, manage windows, type for you.
- **Look at things** — your screen, images, PDFs, spreadsheets — and explain them.
- **Write for you** — documents, notes, code, scripts.
- **Remember** what matters about you, and **learn** your daily routines.
- **Act on its own** when helpful — a morning briefing, a heads-up, a suggestion.

The big difference from cloud assistants: **it's yours**. It runs locally, stays private, has no subscription, and you can even retrain its "voice" yourself (§14).

Here's the whole thing at a glance:

```
mindmap
  root((ELI))
    Talk
      Type or speak
      Wake word
    Do
      Open apps
      Play music
      Manage windows
      Type for you
    See
      Your screen
      Images and PDFs
      Spreadsheets
    Create
      Documents
      Notes
      Code and scripts
    Know you
      Memory of facts
      Living profile
      Learned habits
    Act on its own
      Morning briefing
      Suggestions
      Overnight tasks
    Private
      Runs locally
      Offline by default
      No accounts
```

---

2. How ELI works when you ask it something

Anyone — this is the mental model that makes everything else click.

When you say or type something, ELI follows the same honest loop every time:

```
flowchart TD
    A([You ask ELI something<br/>type or speak]) --> B{Is it a command<br/>or a conversation?}
    B -->|A command<br/>'open spotify'| C{Does it need<br/>the internet?}
    B -->|A conversation<br/>'how are you?'| H[ELI replies in its<br/>own voice]
```

```
    C -->|No| D[ELI does it<br/>on your computer]
    C -->|Yes, e.g. news/search| E{You allowed<br/>online for this?}
    E -->|Yes| F[ELI fetches it,<br/>tells you it went online]
    E -->|No| G[ELI stays offline<br/>and says so]
    D --> I{Did it actually<br/>work?}
    F --> I
    I -->|Yes| J([ELI confirms what it did])
    I -->|No / unsure| K([ELI tells you honestly -<br/>never pretends it worked])
    H --> J
```

The promise in that diagram: ELI **never fakes an action**. If it says it played a song or fixed a file, it really did — and if it couldn't, it says so plainly instead of bluffing. This is built into the software, not just good manners.

---

3. Getting started

A little technical, but only once.

Installing (v2.1.14 and later — one download per platform)

Get the latest from **GitHub Releases**. Every release is built in CI and **launch-tested on Windows, macOS and Linux** before it can publish.

- **Windows:** run `ELI-Setup-<version>.exe` — installs per-user, no admin password. If it sees an NVIDIA card it offers GPU acceleration during setup; if it finds an older ELI it offers a **fresh install** (resets settings & memory; keeps downloaded models). You get three shortcuts: **ELI**, **ELI Server**, **Uninstall ELI**.
- **macOS (Apple Silicon):** open the `.dmg`, drag **ELI** into Applications. First launch is right-click → **Open** (the app is unsigned). Apple-GPU acceleration is built in — nothing to set up.
- **Linux:** `chmod +x ELI_v2-<version>-x86_64.AppImage` and run it. On first launch it offers **applications-menu entries** (ELI, ELI Server, and a working Uninstall).

```
flowchart TD
    A([Run the installer / AppImage]) --> B[First launch]
    B --> C{GPU detected?}
    C -->|NVIDIA| D[Offer: CUDA engine<br/>one-time download, verified]
    C -->|AMD / Intel Arc| E[Offer: Vulkan engine<br/>one-time download, verified]
    C -->|Apple| F[Metal built in]
    C -->|None / decline| G[CPU engine - always works]
    D --> H{Chat model installed?}
    E --> H
    F --> H
    G --> H
    H -->|No| I[Offer: starter model<br/>sized to your hardware]
    H -->|Yes| J
    I --> J[Quick interview]
    J --> K([Ready - say hello])
```

What arrives on first launch

Asset	When	Size
Piper voices (speech out)	already inside the installer	—
Nomic embedder (memory/recall)	fetched automatically	~80 MiB
Whisper (speech in)	fetched on first voice use	~460 MiB

Asset	When	Size
GPU engine (optional)	if you accept the GPU offer	~90 MiB (AMD) · ~1–2 GiB (NVIDIA)
Chat model	if you accept the model offer	sized to your hardware (e.g. ~4.7 GiB)

Everything ELI knows and downloads lives in **one per-user folder** that **survives upgrades** — like a save-game: `%LOCALAPPDATA%\ELI_v2` (Windows), `~/Library/Application Support/ELI_v2` (macOS), `~/.local/share/ELI_v2` (Linux). Databases start as a **true blank slate** — no personal data ships in any installer.

Starting over, or uninstalling

- **Clean slate:** tick *Fresh install* in the Windows setup, or run `ELI --fresh-start` anywhere (keeps downloaded models unless you add `--purge-models`).
- **Uninstall:** Windows → the **Uninstall ELI** shortcut. Linux → the **Uninstall ELI** menu entry (removes the entries, offers to delete your data, then delete the AppImage file). macOS → drag the app to Trash; your data folder is listed above.
- **GPU later / undo:** `ELI --install-gpu-pack` (add `--vulkan` for Intel Arc) · `ELI --remove-gpu-pack`.
- **From source (developers):** `bash install.sh` from a clone works exactly as before.

First launch — the quick interview

The very first time you open ELI, it doesn't know you yet, so it asks **three short questions** (your name, what you do, how you like to be spoken to). Ten seconds, and you can **skip it** entirely. From there, ELI builds its understanding of you naturally as you talk.

Opening ELI

- **Desktop app (full experience):** the **ELI** shortcut / menu entry (installer builds), the AppImage itself (Linux), or `scripts/eli_launch.sh` from a source clone.
- **From your phone:** the **ELI Server** shortcut (or `ELI --server`) starts the home server with the connect QR — see §13.

---

4. Your first hour with ELI

A gentle on-ramp. Try these in order — each builds confidence.

1. **Say hello.** Type `how are you?` — get a feel for ELI's voice.
2. **Ask what it can do.** `what can you do?` lists its capabilities.
3. **Open something.** `open the file manager` or `open firefox`.
4. **Play music.** `play lo-fi on YouTube`, then `pause`, then `skip`.
5. **Look at your screen.** `what's on my screen?`
6. **Tell it a fact.** `remember that my dog's name is Rufus`. Later: `what's my dog's name?`
7. **Set a reminder.** `set a timer for 5 minutes`.
8. **Get a briefing.** `what's the news?` (this one goes online — ELI will say so).
9. **Check on ELI itself.** `what are you running on?` shows its model and hardware.
10. **Go hands-free.** Say the wake word **"computer"**, then a request.

By the end you've touched conversation, control, vision, memory, reminders, the web, voice, and introspection — the whole surface.

---

5. Talking to ELI

There are two ways, freely mixed:

- **Type** in the chat box and press Enter.
- **Speak** — click the microphone, or say the **wake word** (default **"computer"**) then your request. Change the wake word to anything (§11).

You don't need special phrasing. Talk normally:

> "open spotify and play some lo-fi" "what's on my screen?" "remember that my sister's name is Anna" "set an alarm for 7am"

Chaining: ask for several things at once — > "close steam and set an alarm for 7am" > "open spotify then play Juicy by Notorious B.I.G."

If ELI isn't sure what you meant, it asks — it won't guess wildly or pretend. And if a request needs the internet, it tells you before going online.

If you're having a hard time: ELI is tuned to notice genuine first-person distress (it's built to ignore ambient audio and dark jokes, not to over-react). When it does, it steps out of task-mode and gently checks in — pointing you toward real human support rather than ploughing on with commands or reciting a canned line. It's a steer, not a script.

---

6. A tour of the window

ELI Pro is organised into **tabs**. You'll mostly live in **Chat**; here's the rest in plain terms:

Tab	What it's for
Chat	Talk to ELI, by voice or text. Home base.
Quick Actions	One-tap buttons for the things you do most.
Memory	Browse what ELI remembers about you; add or remove facts.
Habits	Routines ELI has learned; turn on/off, edit times, add your own.
Proactive	Controls for ELI's "act on its own" behaviour (suggestions, summaries, insights).
Tasks	Scheduled & overnight jobs and their status.
Images	Generate images — procedural (fast) or diffusion (photoreal).
Screen	Watch/read your screen, take screenshots, OCR.
Coding / IDE	Write, examine, and fix code; a built-in editor.
Self-Improve	What ELI is proposing to improve in *itself* — approval-gated.
Report Builder	Generate longer documents/reports with sources (evidence → outline → draft → revise).
Elis World	ELI's own world-model view.
Labs	Power-user workshop — notebook, Jupyter, calculator, physics, file-chat, **projects/workspaces**, sim-IDE, orchestration, test review.
Settings	Model (incl. the model-switch dropdown), hardware, voice, and privacy options (§16).

You rarely need the tabs directly — almost everything is reachable by **asking** in Chat: "show my habits", "what do you remember about me", "show background jobs", "make an image of …", and ELI opens or reports the right thing.

---

7. What ELI can do — by everyday task

Real things you can say. ELI understands many phrasings — these are examples.

Music & media

- "play lo-fi on YouTube" · "play Juicy by Notorious B.I.G. on Spotify"
- "pause" · "skip" · "next song" · "previous track" · "shuffle" · "repeat this song"
- "what's playing?" · "stop the music" · "skip the ad" (YouTube, when skippable)

Apps & windows

- "open spotify" · "launch firefox" · "open the file manager" · "open github.com"
- "close chrome" · "focus the browser" · "switch to workspace 2"
- "tile windows" · "maximise the window" · "show desktop" · "next window"

Volume, typing, mouse

- "volume up" · "mute" · "set volume to 30"
- "type hello world" · "press enter" · "left click" · "move the mouse up"

Files & notes

- "create a file notes.txt" · "make a folder called projects" · "read notes.txt"
- "list the files in ~/Documents" · "write a note saying buy milk" · "search notes for budget"
- "what's in my clipboard?" · "summarise report.docx" · "convert report.md to pdf"

Writing, documents & code

- "write a report on renewable energy" · "generate a document about X"
- "write a bash script to monitor the GPU" · "solve this: implement a function that …"
- "fix the bugs in foo.py" · "examine eli/memory/memory.py for errors" → ELI scans, **offers** fixes, and applies them only if you say "yes, fix it". · "show the diff"
- "fabricate a test dataset for X" → ELI generates synthetic/test data on a topic.
- The **Report Builder** tab turns "generate a document on X" into a real report: it gathers evidence, plans an outline, drafts each section, then **reviews and revises** it — with document-type quality profiles. It's generation-first, not a one-shot dump.

Screen, images, PDFs, spreadsheets

- "what's on my screen?" · "find the submit button on screen" · "take a screenshot"
- "describe cat.jpg" · "read the text in image.jpg" (OCR)
- "summarise report.pdf" · "analyse data.csv" · "watch my screen" / "stop watching"

Information

- "what time is it?" · "what's the date?" · "weather in Wexford"
- "what's the news?" / "catch me up" → a **synthesised** briefing, not a raw dump
- "search the web for X" (goes online — ELI tells you) · "gpu status" · "system stats"

Reminders, timers, calendar, scheduling

- "set an alarm for 7am" · "set a timer for 10 minutes"
- "add an event tomorrow at 3pm" · "list my events" · "start a pomodoro"
- "research the best solar inverters overnight" · "build me a script at 2am" → **background jobs**
- "show background jobs" · "check job 5"

Asking ELI about itself (honest introspection)

- "what are you running on?" (model, context, GPU) · "how does your memory work?"
- "what do you know about me?" · "what have you been working on?" · "status"

Eye control (gaze)

- "calibrate gaze", then "enable gaze". With it on, ELI clicks **where you're looking** when you say "open" / "left click" / "right click" / "hit enter" — hands-free pointing.
- **Fully hands-free *and* voice-free — dwell-click.** Say **"enable gaze clicking"** (or turn on **Dwell-click** in Settings) and ELI clicks wherever you **rest your gaze for about a second** — no mouse, no voice needed. This is built for people who can't use their hands *or* speak reliably. Tune the hold time, wander radius, and sensitivity in Settings (`gaze_dwell_seconds`, `gaze_dwell_radius`). Turn it off any time; it's opt-in.
- "gaze status" · "disable gaze"

Making images

- "make an image of a red bike by the sea" · "generate a picture of …"
- Two engines: a fast **procedural** one (always available) and **diffusion** (SSD-1B) for photoreal results — ELI briefly frees the model's VRAM to run it. The **Images** tab has the controls; results save locally.

Coding & building things

- "solve this: implement a function that …" → ELI plans it, writes it, then **verifies and repairs** it.
- "generate a project that does X" (a full multi-file scaffold) · "write a bash script to …"
- "examine eli/memory/memory.py for errors" → a tiered scan; it **offers** fixes and applies them only if you say "yes". · "show the diff" · "generate tests for your code" (it writes *and* sandbox-verifies them). The **Coding** tab and the built-in **IDE** are home for this.

ELI improving itself

- "improve yourself" → a self-improvement patch cycle — proposed, and **gated by your approval**.
- "show your self-improvement log" · "patch yourself" · "upgrade yourself" (git pull → deps → rebuild indexes) · "run your self-tests". The **Self-Improve** tab shows what it's proposing.

Plugins & your own agents

- "list plugins" · "enable the X plugin" · "install the X plugin" · "plugin status"
- You can **build your own agent** just by describing it — ELI runs a short dialog, validates it, and registers it live (the create-agent flow).

Overnight & scheduled work

- "research the best solar inverters overnight" · "build me a script at 2am" · "every night, regenerate the test report" → durable **background jobs** that survive restarts.
- "show background jobs" · "check job 5". The **Tasks** tab lists them.

Shaping ELI's voice (persona)

ELI's personality is **emergent** — it forms from your profile, your history, and the way you talk to it, not from a hand-written script. Most of the time you leave it alone and it just *becomes* itself with you. When you want to steer it, you can, without erasing that: - "lock your persona to a terse senior engineer" · "what persona are you locked to?" · "clear persona lock". A lock **layers on top** of the emergent voice for the session; clearing it hands the wheel back to the natural persona. - "refresh your persona" reloads it from your latest profile (after ELI's learned something new about you). - **Set how it talks to you** *(new, 2.1.14)* — say *"speak to me more bluntly"*, *"be funnier"*, or *"respond as Rick"* and it **sticks** (say *"reset your tone"* to clear it). You can also set it in **Settings → Identity → Tone / how ELI speaks**. This explicit preference takes priority over the tone

ELI infers, but leaving it blank keeps ELI's natural emergent voice in charge. - ELI also keeps a small set of **core values** (memory policy, verbosity, safety) that are part of its persona and evolve alongside it.

The rule of thumb: **the emergent voice is the default and the truth of who ELI is with you; a lock is a temporary hat it wears on request.** ELI won't drift into a scripted character on its own, and a lock never overwrites the underlying personality — it sits on top until you clear it.

---

8. ELI remembers you (memory)

ELI builds a private, evolving understanding of **who you are** and reads it every time you talk, so you never repeat yourself.

```
flowchart TD
    A([You're talking to ELI]) --> B{What kind of thing<br/>did you say?}
    B -->|'Remember that ...'| C[Saved as a durable fact<br/>kept until you remove it]
    B -->|Just chatting<br/>interests, focus, tone| D[Quietly updates ELI's<br/>living profile of you]
    C --> E[(Local memory<br/>on your computer)]
    D --> E
    E --> F([Read every turn so ELI<br/>stays in context])
    G[/Old, abandoned topics<br/>fade over time/] --> E
```

Tell it something to keep: "remember that my sister's name is Anna" · "my name is Alex" **Ask what it knows:** "what do you remember about my car?" · "what do you know about me?" · "explain everything you know about me and where it's stored" (a sourced, detailed answer).

It remembers the thread, not just facts: when a conversation ends, ELI quietly summarises it and carries a digest of your recent sessions forward — so it recalls what you've been *working through* over days, not just isolated one-off facts.

Everything is stored **locally** in small private databases. Review and prune it in the **Memory** tab. A brand-new install is a **blank slate** — ELI knows nothing about you until you talk.

---

9. ELI learns your routines (habits)

If you do the same thing at roughly the same time on **several different days**, ELI notices and **offers** to automate it. It never activates anything without your "yes".

```
flowchart TD
    A[You open your email app<br/>around 9am, several days] --> B{Same action, same time,<br/>on 3+ diffe
    B -->|No| X[ELI stays quiet<br/>no nonsense suggestions]
    B -->|Yes| C[ELI offers:<br/>'Add this as a habit? yes/no']
    C --> D{Your answer}
    D -->|No| E[Dropped, won't ask again]
    D -->|Yes| F{Has today's time<br/>already passed?}
    F -->|Yes| G[Runs it now once,<br/>then daily from tomorrow]
    F -->|No| H[Runs at the time, daily]
    G --> I([Active habit - edit in Habits tab])
    H --> I
```

Manage everything in the **Habits** tab (add, edit times, turn off), or say "show my habits". The key safeguards: it only proposes routines you've **genuinely repeated across days**, and it **always asks first**.

---

10. ELI helps before you ask (proactive)

With **proactive mode** on, ELI can do helpful things on its own — a **morning briefing**, a gentle suggestion, or a goal it thinks would help. It's conservative and easy to control:

- "start proactive mode" / "stop proactive mode" · "proactive status"
- "morning report" / "give me my briefing" (news, weather, your agenda)
- "do you have any proposals for me?" / "what's on your agenda?"

Schedule real unattended work too: > "research X overnight" · "generate tests for your code tonight"

These appear in the **Tasks** tab and survive restarts.

The autonomy tick — what "on its own" actually means

When you allow it, ELI runs a quiet **autonomy tick** inside the proactive daemon — roughly every half hour, and only if it's switched on (kill switch: `ELI_AUTONOMY_TICK=0`). One tick does three things, all **read-only or proposal-only**:

1. **Code monitor** — it glances over its own code health and flags anything that's drifted.
2. **Self-model refresh** — it updates its understanding of itself (what it's running, what it can do) so its introspection stays honest.
3. **Goal autogenesis** — it may propose a goal, or a scheduled task, it reckons would help you.

The guardrail that matters: every autogenerated goal is **proposal-only**. It shows up as a suggestion you accept or dismiss — nothing destructive ever runs unattended, and anything risky still goes through the **approval gate** (Appendix C). Autonomy makes ELI *thoughtful* on its own, not *loose*.

---

11. Voice, wake word, and dictation

ELI is fully usable hands-free.

```
flowchart LR
    A([Say the wake word<br/>'computer']) --> B[ELI starts listening]
    B --> C[You speak your request]
    C --> D[ELI transcribes it<br/>on your computer]
    D --> E[ELI does the task]
    E --> F([Optionally speaks the answer back])
```

- **Wake word:** say **"computer"** then your request. Change it: "change the wake word to athena". ELI trains its listener on the new word automatically — works even over background music.
- **Personalise to your voice:** "enroll my wake word" (records you saying it and adapts).
- **Dictation:** "start dictation" / "stop dictating" — speak and ELI types into the active field.
- **Transcribe a recording:** "transcribe audio.wav"
- **ELI speaking back:** "say good morning"
- **Teach ELI your voice & tone:** "train my voice" — it learns your pitch/energy and even emotion cues, then adapts how it speaks to you.
- **If voice misbehaves:** "run voice diagnostics".

---

12. Seeing your screen and images (vision)

ELI can actually **look**:

- **Your screen:** "what's on my screen?" · "what does this error say?" · "find the submit button"
- **Image files:** "describe cat.jpg" · "read the text in this image" (OCR)
- **Documents:** "summarise report.pdf" · "analyse data.csv" · "analyse the pdfs in that folder"
- **Continuous watching:** "watch my screen" (and "stop watching")

The first time you use vision, ELI may fetch a small vision model (you can pre-download it — §14). With a webcam set up, ELI even supports **eye-tracking control**: "enable gaze control", "calibrate gaze", then click where your eyes rest.

---

13. The ELI server & using it from your phone

Slightly technical, but only at setup.

ELI includes a built-in **server** and **web app**. Start the server on your computer and chat with ELI from your **phone or tablet's browser** over your home Wi-Fi — while all the actual thinking stays on your computer.

```
flowchart TD
    A([Start the ELI Server<br/>on your computer]) --> B{Who should reach it?}
    B -->|Just this computer<br/>default| C[Private to localhost]
    B -->|My phone/tablet<br/>on home Wi-Fi| D[Start in LAN mode]
    D --> E[It prints an address<br/>+ a security token]
    E --> F[Open that address in<br/>your phone's browser]
    F --> G[Enter the token once]
    G --> H([Chat with ELI from your phone<br/>thinking stays on the PC])
    C --> H
```

- Start it from the desktop app — **Settings → Web Server** (a "this computer" button and a "phone / Wi-Fi" button that shows the exact link to open). Or run `scripts/eli_serve.sh` (Mac/Linux) / `scripts/eli_serve.ps1` (Windows). Full details: `docs/SERVER_AND_WEB_APP.md`.
- The web app is **installable** — your phone's "Add to Home Screen" makes it open like an app, and the shell works offline. Light/dark themes; you can recolour the chat.

Starting the server, and what it prints

Start the server and it does two friendly things so you never have to hunt for the address: it **prints a clickable link** (Ctrl/Cmd-click it right in the terminal), and it **auto-opens your local browser** to the dashboard. (Set `ELI_API_NO_BROWSER=1` if you'd rather it didn't.)

- **This computer only (default):** the dashboard opens at `http://127.0.0.1:8081/`. No token — you're on the machine, so you're trusted automatically.
- **Phone / Wi-Fi (LAN mode):** it prints your computer's LAN address (e.g. `http://192.168.1.20:8081/`) with a **one-time token** baked into the link, plus a **QR code** in the Connect tab. Scan it or open the link and you're in.

The token, and the rotate button

The phone's token is **stable** — it's saved (permissions `0600`, under your config dir) and survives server restarts, so a paired phone stays paired; no re-scanning every reboot. Lost the phone, or just want a clean slate? Hit **rotate**: it mints a fresh token and instantly cuts off anything still using the old one.

How the link keeps your token safe

The token rides in the link's **fragment** — the part after the `#` (`…:8081/#token=…`). Browsers **never send the fragment to the server**, so your token can't land in the server's access log (or any proxy log in between). When the page loads it reads the token from the fragment, stores it locally, then **wipes it from the address bar** so it isn't left in your history either. Older links that used `?token=` still work — the page accepts both — but everything ELI prints or displays now uses the safe fragment form, on the desktop panel and both launcher scripts alike.

Voice from the phone needs HTTPS

Browsers block the microphone on a plain `http://` LAN address, so the phone **mic** needs a secure page. Start with `--https` (or the HTTPS toggle) and ELI runs a *second* server alongside on port **8443** with a self-signed certificate — accept the one-time "not private" warning and the mic works. Plain HTTP stays the primary path (it's what QR scanners open reliably); HTTPS runs *in addition*, purely for voice.

Switching the model from the dashboard

Settings → Model lists every model installed on your computer, with sizes. Pick one and ELI **hot-swaps** it live — no restart. Only real, installed files show up, so you can't point it at something that won't load, and only an **admin** can switch it.

Two servers, one machine

Worth knowing: "the server" is really **two** local services. The **web/API server** (port 8081) serves the dashboard, the phone app, and a small API. The **device server** is a separate MQTT service that talks to your smart-home gear (see the Home tab, below). Both run locally; both are yours. Everything about who's allowed in — tokens, roles, the loopback trust — is laid out in **Appendix C (Security)**.

Under the hood (for scripts & power users)

The web/API server is a **FastAPI** app served by **uvicorn** (`api/server.py`). You rarely touch it directly — the desktop app and the launcher scripts wire everything up — but if you script it, a handful of environment variables control it: - `ELI_API_HOST` — `127.0.0.1` (default, this-computer-only) or `0.0.0.0` (LAN mode). - `ELI_API_PORT` — the HTTP port (default `8081`); the HTTPS voice server uses `8443`. - `ELI_API_TOKEN` — set your own token; if unset, ELI uses the saved stable one (`config/api_token`, permissions `0600`) so a paired phone survives restarts. - `ELI_API_ALLOW_TOKENLESS` — honoured only on loopback; LAN mode always requires the token. - `ELI_API_NO_BROWSER=1` — start without auto-opening the local browser.

The launchers wrap all of this: `scripts/eli_serve.sh` (Mac/Linux) and `scripts/eli_serve.ps1` (Windows) start LAN mode, then print the phone link (with the safe `#token=` fragment) and the **exact** firewall command for your OS — the quickest way in from a terminal. The API itself is a small REST surface (chat, status/telemetry, model list/switch, voice, memory, research, audit); full endpoint detail lives in `docs/SERVER_AND_WEB_APP.md`.

If a phone can't connect

Nine times out of ten it's the firewall. When it starts in LAN mode ELI prints the **exact** command to open the port for your OS — run that, and make sure the phone is on the **same Wi-Fi**. It's all scoped to your local network; nothing is exposed to the internet.

What's in the web app

It has a sidebar with several sections: - **Overview** — a live dashboard (clock, GPU/CPU/RAM gauges, status, recent activity). - **Chat** — streamed replies with markdown/code, saved sessions, stop/regenerate, and voice in/out. - **Commands** — a searchable list of everything ELI can do. - **Home** — ELI's own **home-AI** (see below). - **System** — live, measured GPU/CPU/RAM/model telemetry. - **Research** — shared document corpora you can build and query **together**, with citations. - **Audit** — a tamper-evident log of every action. - **Admin** — (admin only) per-user activity, the risk-gate policy, and user management.

Roles (optional)

By default the person on the computer is the operator (full admin). If others share it, you can create accounts with roles — **viewer** (read-only dashboards), **member** (use everything), **admin** (also the Admin console) — in the Admin tab. Each gets a one-time link; the audit trail then attributes actions to the right person.

The Home tab — ELI as your home assistant

ELI controls smart devices **directly over MQTT** (ESPHome / Tasmota / Zigbee2MQTT) — no Home Assistant, no cloud: - **Find devices** on your network with one click (mDNS), or add one by its topics. - **Control** lights/switches, group them into **rooms**, and run room-wide on/off — from the app or by voice ("turn on the desk lamp", "turn off the kitchen"). - **Scenes** — save a set of device states as "Movie mode" and activate it (by voice too). - **Automations** — *when* something happens (a time, **sunrise/sunset**, or a device turning on/off), *do* something (control a device or run a scene), optionally *only if* a condition holds. - ELI **learns** how you use your home and **suggests** automations you can accept with one tap.

---

14. Models: picking, switching, and training your own

The **model** is the "brain" ELI thinks with — a file on your computer. Bigger = smarter but needs more graphics memory (VRAM). ELI picks a suitable one at install; change it anytime.

Which model should I get?

```
flowchart TD
    A([How much graphics memory<br/>VRAM do you have?]) --> B{Amount?}
    B -->|None / CPU only| C[qwen2.5-3b<br/>small and fast]
    B -->|~8 GB| D[qwen2.5-7b default<br/>or qwen3-8b]
    B -->|~12 GB| E[falcon3-10b<br/>or phi-4]
    B -->|24 GB+| F[qwen3.6-35b-a3b<br/>or falcon-h1-34b]
    C --> G([ELI auto-sizes it to your<br/>hardware - you tune nothing])
    D --> G
    E --> G
    F --> G
```

Downloading

- **Pick from a menu (choose any number of models):** `python -m eli.core.model_download --choose` — this is what the installer uses; you tick the ones you want (or `all` / `auto` / `none`).
- Let ELI choose one best-fit for your hardware: `python -m eli.core.model_download --auto`
- See the menu without downloading: `python -m eli.core.model_download --list`
- Grab a specific one: `python -m eli.core.model_download qwen2.5-7b`

You're not limited to one — download several and switch between them anytime (§"Switching models").

Command name	Model	Roughly needs
`qwen2.5-3b`	small & fast	4 GB graphics / or CPU
`qwen2.5-7b` *(default)*	great all-rounder	8 GB graphics
`qwen3-8b`	newer, big context, good reasoning	8 GB graphics
`falcon3-10b`	a step up	12 GB graphics
`phi-4`	strong reasoning for its size	12 GB graphics
`qwen3.6-35b-a3b`	very capable (mixture-of-experts)	24 GB / or slow on CPU
`falcon-h1-34b`	largest option	24 GB / or slow on CPU

You can also just **drop any `.gguf` file** into the `models/` folder and ELI finds it.

Switching models

In **Settings → Model** (or "Model menu / Auto Detect"), pick the file. ELI sizes it to your hardware automatically — you don't tune anything.

Training your own ELI-flavoured model (advanced, optional)

You can fine-tune a model so it speaks in **ELI's voice** out of the box.

See `blueprints/finetuning_guide.md` — a full step-by-step from choosing a base model, through extracting a voice dataset and training, to producing a ready-to-load `.gguf`. Crucially, ELI's **personality stays live** (never frozen into the model); training only shapes the *manner* of speech.

---

15. How hard ELI thinks (reasoning modes)

ELI has five effort levels. Faster modes answer quickly; deeper modes think longer and use more of ELI's internal helpers for hard problems.

Mode	Feel	Use it for
quick	snappy	greetings, simple commands, quick facts
normal	balanced (default)	everyday questions and tasks
advanced	more thorough	tricky, multi-step asks
research	digs deep	gathering and synthesising a lot
expert	maximum effort	the hardest problems

Ask "what reasoning mode are you in?" or "explain all your reasoning modes". For most people the default is right — ELI also deepens on its own when a problem clearly needs it.

---

16. Settings you'll actually touch

Most people never need these. The few that matter:

- **Model** — which brain ELI uses (§14).
- **Voice** — wake word, text-to-speech on/off, microphone.
- **Hardware / startup** — ELI auto-tunes; advanced users can nudge how much graphics memory to hold back, or how much context to use. Leave alone unless you know why.
- **Privacy** — ELI is offline by default; this is where anything network-related lives.

Two graphics cards? ELI can use both — set in the GPU profile settings; single-card just works.

---

17. Privacy & staying offline

- **Nothing leaves your computer** unless you ask for something online (web search, news, weather, downloading a model). ELI tells you when it goes online.
- **Your data is yours** — conversations, memories, files, profile live in local databases. Delete anytime.
- **No accounts, no telemetry, no subscription.**
- A fresh install is a **blank slate** — ELI starts knowing nothing about you.

---

18. Troubleshooting

Most issues map to one of these. When in doubt, **ask ELI** — "run a full runtime audit" or "status" — it reports honestly.

"ELI is replying very slowly"

```
flowchart TD
    A([Replies feel slow]) --> B[Ask: 'what are you running on?']
    B --> C{Running on GPU<br/>or CPU?}
    C -->|CPU| D[Reinstall with graphics support,<br/>or pick a smaller model]
    C -->|GPU but big model| E[Switch to a smaller model<br/>see section 14]
    C -->|GPU, right-sized| F[That's normal speed for<br/>your hardware]
    D --> G([Faster])
    E --> G
```

Everything else

Problem	Try this
Replies look like gibberish	A model whose context is too small. Pick a recommended one (§14); ELI sizes context automatically.
"I couldn't find a model"	`python -m eli.core.model_download --auto`
Memory/recall seems empty	Normal on a fresh install. Ensure the embedder downloaded: `python -m eli.core.model_download --aux`
Voice/wake word not responding	"run voice diagnostics"; check the microphone in Settings → Voice.
Won't go online for news/search	By design (offline-first). The network is allowed per-request; check Settings → Privacy.
A habit didn't run	Open **Habits** — confirm it's enabled and the time is right; or "show my habits".
Something seems broken	"run a full runtime audit" or "status".

---

19. Quick phrasebook (cheat sheet)

Everyday - "open spotify and play lo-fi" · "pause" · "skip" - "what's on my screen?" · "summarise report.pdf" - "set an alarm for 7am" · "set a timer for 10 minutes" - "what's the news?" · "weather in Dublin" · "what time is it?"

Memory & you - "remember that …" · "what do you know about me?" · "my name is Alex"

Habits & proactive - "show my habits" · "yes" / "no" (to an offer) · "morning report" · "start proactive mode"

Writing & code - "write a report on X" · "write a script to …" · "fix the bugs in foo.py"

Voice - wake word "computer" + request · "change the wake word to athena" · "start dictation" · "say good morning" · "train my voice"

Check on ELI - "what are you running on?" · "status" · "what have you been working on?"

Models (in a terminal) - `python -m eli.core.model_download --list` · `python -m eli.core.model_download --auto`

---

20. Glossary

Term	Plain meaning
Model / GGUF	The "brain" file ELI thinks with. `.gguf` is just its file format.
VRAM	Memory on your graphics card. More VRAM → you can run a bigger, smarter model.
Embedder	A tiny helper model that powers ELI's memory/recall. Installed automatically.
Context	How much of the conversation ELI can "hold in mind" at once. ELI sets this for you.
Vision model	The model that lets ELI understand screens and images.
Wake word	The phrase that makes ELI start listening (default "computer").
Proactive mode	ELI doing helpful things on its own (briefings, suggestions).
Habit	A routine ELI learned and you approved, run on a schedule.
Fine-tuning	Optional advanced step to teach a model ELI's voice (§14).
Offline-by-default	ELI never uses the internet unless you ask for something that needs it.

---

21. Appendix A — Under the hood: how ELI thinks (the full pipeline)

For the curious / technical. You never need this to use ELI — but here is the complete, verified path from your words to ELI's reply.

Every request runs through one funnel — `CognitiveEngine.process()` — which takes one of three paths depending on what you asked and the reasoning mode. The diagram is the **real** pipeline (verified against the running code): a deterministic fast-path for plain commands, a **quick** path via the Agent Bus (the default), and a **deep** 12-stage Orchestrator for hard work.

```
flowchart TD
    U([Your input - text or voice]) --> R[Router<br/>regex priority pipeline<br/>+ LLM-intent fallback]
    R --> P[CognitiveEngine.process<br/>action · args · confidence · mode]
    P --> FP{Plain deterministic command?<br/>open app · media · volume · status · job}
    FP -->|Yes · PHASE45 fast-path| FX[execute_action] --> VERB([Reply returned verbatim<br/>no LLM, inst
    FP -->|No| MODE{Reasoning mode}

    MODE -->|quick · default| BUS[Agent Bus dispatch]
    BUS --> AG[15 specialist agents run over a DAG<br/>memory · knowledge-graph · system · file-code<br/>
    AG --> DR[DispatchResult<br/>grounding_confidence · agents_used · memory_context]
    DR --> GE{Low grounding on a<br/>factual question?}
    GE -->|Yes| ESC[Escalate: web tier / local tier / honest hedge] --> GOV
    GE -->|No| KIND{What kind of result?}
    KIND -->|self-contained action| GOV
    KIND -->|grounded control| GSYN[Grounded synthesis] --> GOV
    KIND -->|conversation / CHAT| SYS[Build enhanced system prompt<br/>persona + memory brief + context]

    MODE -->|normal/advanced/research/expert · deep| ORCH[12-stage Orchestrator]

    subgraph DEEP[The 12-stage Orchestrator]
      direction TB
```

```
    O1[1 · Intent routing] --> O2[2 · Persona lock]
    O2 --> O3[3 · HyDE query expansion]
    O3 --> O4[4 · Planner - channels + budgets for the mode]
    O4 --> O5[5-7 · Parallel retrieval<br/>keyword · FTS5 turns · FAISS vector · RAG · KG]
    O5 --> O8[8 · Hybrid merge]
    O8 --> O9[9 · Cross-encoder rerank]
    O9 --> O10[10 · Context assembly]
    O10 --> O105[10.5 · Persona handoff]
    O105 --> O11[11 · LLM generation]
  end
  ORCH --> O1
  O11 --> O12{12 · Confidence vs threshold}
  O12 -->|pass| GOV
  O12 -->|too low| REPAIR[Repair / regenerate] --> GOV

  GEN[LLM generation<br/>broker.infer / stream] --> GOV[Output governor<br/>+ No-Fake-Actions guard<br/
  GOV --> OUT([Reply to you])
  VERB --> OUT
```

The retrieval stages (5–7) are where memory plugs in — that's Appendix B. The two guarantees baked into the end of every path: the **Output governor** cleans the reply, and the **No-Fake-Actions guard** ensures ELI never claims it did something it didn't.

A.1 — The router (before any path is chosen)

Your words first hit the **router**. It runs a **regex-first priority pipeline**: an ordered set of fast, exact checks (web realtime lookups, media controls, file/PDF, memory, OS control, …), each carefully shaped so it only fires on a real command — e.g. media controls need a whole word in command shape, so "metabolise" never triggers a media "tab". If nothing matches, an **LLM intent classifier** is the fallback. The router emits `{action, args, confidence}` and hands off to `CognitiveEngine.process()`. The router also remembers the *last used path* so follow-ups ("do it again", "the second one") resolve correctly.

A.2 — The three paths, and when each is taken

- **Fast-path (PHASE45) — no AI involved.** Plain, deterministic actions (open an app, pause music, set volume, report status, check a job) are executed directly and the result is returned **verbatim**. There's no model call, so these are instant and can't be "hallucinated". This is why "pause" never produces a chatty paragraph.
- **Quick path (the default) — the Agent Bus.** Conversational and lightly-grounded requests go to the **Agent Bus**, which runs up to **15 specialist agents** in parallel over a dependency graph, each contributing evidence (memory hits, knowledge-graph facts, system readings, code search, capability info, …). Their combined result carries a **grounding confidence** — an honest "is this actually backed by anything?" score.
- **Deep path — the 12-stage Orchestrator.** When you ask for depth (advanced/research/expert modes, or a hard question), ELI runs the full pipeline below: more retrieval, reranking, and a confidence check with automatic repair.

A.3 — The Agent Bus, in full

The bus **selects a relevant subset** of agents for your intent (or fans out broadly when unsure), runs them concurrently, and **aggregates** their findings into a `DispatchResult`. Key agents and what they bring:

Agent	Contributes
Memory	recalls facts/turns from SQLite + full-text + vector search
Knowledge-Graph	entities and how they relate (who/what/where)

Agent	Contributes
System	runs executor actions (web search, audits, status)
File-Code	searches the actual codebase for grounded answers about ELI itself
Introspection	live runtime/self state
Capability	what ELI can do (the capability manifest)
Habit / Proactive / Reflection / Self-Improve	routines, signals, insights, fixes
Plugin / Voice / Frontier / Orchestrator	plugin dispatch, speech state, deeper reasoning hooks, planning

After the bus returns, a **grounding-escalation hook** checks: *is this a factual question that came back weakly supported?* If so, ELI escalates — a web tier (if you allow online), a local tier, or an **honest hedge** ("I'm not certain, but…") rather than a confident guess. Then the result is finalised: a self-contained action returns directly, a grounded control action gets a short grounded synthesis, and a conversation goes to generation with the full persona + memory context.

A.4 — The 12 stages, one by one (the deep path)

1. **Intent routing** — settle exactly what you're asking for.
2. **Persona lock** — fix ELI's identity/voice for this turn so it stays consistent.
3. **HyDE query expansion** — ELI drafts a *hypothetical answer* and searches with **that** too, which dramatically improves what memory retrieves (you find better matches by searching with an answer-shaped query, not just the bare question). Skipped for very short queries.
4. **Planner** — decide *which* retrieval channels to use and how big the budgets are, tuned to the reasoning mode (fast / balanced / deep). 5–7. **Parallel retrieval** — several searches run at once: keyword, **full-text (FTS5)** over your conversation history, **FAISS vector** (semantic) search, optional RAG, and the **knowledge graph**. (This is the seam where Appendix B's memory system feeds in.)
5. **Hybrid merge** — combine the hits from every channel into one candidate set.
6. **Cross-encoder rerank** — a smarter model re-scores the merged candidates for true relevance to your question, and the best rise to the top. (Skipped for non-conversational asks.)
7. **Context assembly** — build the grounded context block that will inform the answer. 10.5. **Persona handoff** — wrap that context with ELI's live persona brief for generation.
8. **LLM generation** — the model writes the reply (streaming or one-shot). For the deepest modes this is a two-part private-reasoning-then-condense step.
9. **Confidence check** — score the answer against a threshold; if it's too low, ELI **repairs/regenerates** rather than shipping a weak answer.

A.5 — The two end-of-path guarantees

Every path, no matter which, ends at: - **Output governor** — normalises and cleans the reply (strips artefacts, enforces formatting). - **No-Fake-Actions guard** — if ELI's reply *claims* it did something (played a song, fixed a file), the system verifies it actually happened; if not, the claim is replaced with the honest truth. This is enforced in code, not left to the model's goodwill.

---

22. Appendix B — Under the hood: how ELI remembers (the full memory system)

For the curious / technical. ELI's memory is a genuine hybrid — relational + full-text + vector + graph — all local. This is the complete input → storage → recall path (verified against the code).

```
flowchart TD
    subgraph IN[INPUT - everything ELI may write]
      direction LR
      I1[Conversation turns]
```

```
  I2["'Remember X' durable facts"]
  I3[Observations + user patterns]
  I4[Habits + habit events]
  I5[Failures · corrections · improvements]
  I6[Knowledge-graph entities + relations]
  I7[User Model brief]
end
IN --> MEM[Memory class<br/>memory.py routes every write]

MEM --> D1[(user.sqlite3<br/>memories · turns · habits<br/>patterns · user_model · KG)]
MEM --> D2[(agent.sqlite3<br/>dispatch + metrics)]
MEM --> D3[(system_index.sqlite3<br/>apps · files · dirs)]
MEM --> D4[(coding_memory.sqlite3<br/>bug fixes)]
MEM -->|embeds new text| V[[FAISS vector index<br/>nomic embedder<br/>serialized - segfault-safe]]
MEM -->|indexes text| F[[FTS5 full-text<br/>memories + turns + KG]]
MEM -->|entities/relations| KG[[Knowledge graph<br/>subject - predicate - object]]

Q([Recall request<br/>from pipeline stages 5-7]) --> RC[recall_memory<br/>hybrid retriever]
V --> RC
F --> RC
RC --> S1[1 · FAISS vector search - primary]
S1 --> S2{Cold index or<br/>fewer than limit/2 hits?}
S2 -->|Yes| S3[2 · FTS5 / LIKE keyword supplement]
S2 -->|No| S4
S3 --> S4[3 · Noise filter<br/>drop ELI's own insights / reflections /<br/>session summaries / rows c
S4 --> S5[4 · Importance-weighted ordering<br/>COALESCE importance 0.5]
S5 --> CTX([Grounded context handed to the prompt])
KG -.->|context_for_prompt| CTX
I7 -.->|read fresh every turn| CTX

subgraph MECH[UPKEEP MECHANISMS]
  direction LR
  M1[Weight decay<br/>old, low-importance rows fade ×0.98]
  M2[Embedder lock<br/>one-at-a-time embedding]
  M3[recall_log<br/>records what was recalled]
end
M1 -.-> D1
```

Why it's trustworthy: the **noise filter** stops ELI's own past replies and reflections from resurfacing as if they were *your* facts; the **embedder lock** keeps the vector engine from crashing under concurrent use; and **weight decay** is the gentle "forgetting" that lets stale topics fade while active ones stay fresh. Nothing here leaves your computer.

B.1 — What ELI writes (the inputs)

Everything that flows into memory passes through one **Memory** module, which decides where each item belongs: - **Conversation turns** — what you said and what ELI replied, with timestamps. - **Durable facts** — anything you explicitly ask it to "remember". - **Observations & user patterns** — quiet signals about your interests, focus, and tone that build the living profile. - **Habits & habit events** — your repeated actions and the routines learned from them. - **Failures, corrections, improvements** — ELI's own mistakes and fixes, kept for self-learning (these are deliberately **excluded** from normal recall — see the noise filter). - **Knowledge-graph entities & relations** — named things and how they connect. - **User Model brief** — a compact, pre-rendered summary of who you are, refreshed over time.

B.2 — Where it's stored (four local databases + three indexes)

Everything is plain local files on your machine — no server, no cloud: - `user.sqlite3` — the main store: memories, conversation turns, habits, patterns, the User Model, and the knowledge graph. - `agent.sqlite3` — ELI's internal agent activity and performance metrics. - `system_index.sqlite3` — an index of your apps, executables, files, and folders (so "open the file manager" is instant). - `coding_memory.sqlite3` — remembered code bug-fixes.

On top of the relational data sit three search indexes: - **FAISS vector index** — every stored text is turned into an "embedding" (a list of numbers capturing meaning) by a small local **nomic embedder**, so ELI can find things by *meaning*, not just exact words. Embedding is done **one-at-a-time** behind a lock (the embedder isn't thread-safe; serialising it prevents crashes). - **FTS5 full-text index** — fast exact/keyword search over memories, conversation turns, and the knowledge graph. - **Knowledge graph** — a subject → predicate → object web (e.g. *you → own → a Tesla*) that ELI can walk to answer relational questions.

B.3 — How recall works (step by step)

When the cognition pipeline (stages 5–7) needs memory, it calls `recall_memory`, a genuine hybrid retriever that runs in this order: 1. **Vector search first (FAISS).** The semantic index is the primary path — it finds things that *mean* the same even if worded differently. 2. **Keyword supplement (FTS5/LIKE) — only if needed.** If the vector index is cold/empty or returns fewer than half the requested results (e.g. a very short query), a keyword search tops it up. (When another part of the system already did the vector search, this step is skipped to avoid double-searching.) 3. **Noise filter.** Results are scrubbed of ELI's *own* output — assistant insights, reflections, session summaries, the "orchestrator" source, and any over-long blobs (>1500 chars). This is the safeguard that stops ELI's past musings from masquerading as *your* facts. 4. **Importance-weighted ordering.** What survives is ranked by an importance score so the most significant memories surface first.

Alongside this, the **knowledge graph** contributes a lightweight relational context, and your **User Model brief** is read **fresh every single turn** — that's why ELI stays in context about who you are without you repeating yourself.

B.4 — Upkeep (the mechanisms that keep memory healthy)

- **Weight decay** — old, low-importance memories are gently faded (multiplied down over time), so abandoned topics naturally recede while active ones stay prominent. This is ELI's "forgetting".
- **Embedder serialization** — the single lock around the embedder that keeps the vector engine stable under concurrent access.
- **Recall log** — a record of what was retrieved, used for tuning and for honestly explaining *where* a piece of knowledge came from ("how do you know that?").

B.5 — Privacy, restated

Every database and index in this appendix is a local file under your user profile. There is no sync, no upload, no account. Delete the files (or use the **Memory** tab) and that knowledge is gone. A fresh install starts with the full structure but **zero** personal rows — a true blank slate.

---

23. Appendix C — Under the hood: how ELI stays safe and yours (security)

For the curious / technical. ELI is meant to be safe by construction, not by good intentions — every guard below is enforced in code, not left to a setting you might forget to flip. All of this is verified against the running codebase.

C.1 — Offline by default (the socket failsafe)

ELI doesn't just *prefer* to stay offline — it's physically stopped from reaching the internet unless you asked for something that needs it. At startup, `netguard` installs a **process-wide socket guard**: it patches the low-level

`connect()` call so that *any* outbound connection to a non-loopback address raises an `OfflineError` while ELI is offline. Loopback and local sockets (the phone web server, the local model) always work. So even if some stray bit of code *tried* to phone home, the operating-system call itself is blocked — belt and braces. When you do turn networking on for a task, that flip is written to the audit ledger; "internet on" is never silent.

Ask for something online while offline and you don't get a crash or an invented answer — you get an honest refusal: *"I can't fetch that right now, I'm offline."*

C.2 — The fail-closed command gate

ELI can run shell commands and open apps, so that path is gated hard. Before any command runs it's checked against an **allowlist** by the SecurityManager. The part that matters: if the security layer can't load for any reason, the check **fails closed** — it returns *denied*, not *allowed*. A broken guard blocks the action; it never quietly waves one through. Sandboxed runs are limited to a fixed set of allowed executables.

C.3 — The approval gate (nothing risky happens behind your back)

Anything ELI proposes on its own — an autonomous action, a self-improvement patch, a scheduled job — goes through the **approval engine** before it can execute. Each proposal carries an *action class*, and the engine decides: auto-approve the harmless ones, or hold the riskier ones until **you** confirm. Who is even allowed to propose what is also gated — a background daemon can't quietly emit a dangerous action. There's a **Full Control** mode that lifts the confirmation barrier if you want ELI to just act, but that's your explicit choice, and it's off by default.

C.4 — The phone: who's allowed in

The web server (§13) is locked down. On the same machine (loopback) you're trusted automatically. From the network, every request must carry a **bearer token** — and that token is never stored in plain text, only its SHA-256 hash. The token travels in the link's URL **fragment** (after the `#`), which browsers never transmit to the server, so it can't appear in the access log; the page reads it locally and immediately clears it from the address bar. Define one or more users and **RBAC** switches on: each token maps to a role (admin / member), and admin-only actions — changing settings, switching the model — reject anyone who isn't admin. With no users defined it's single-operator mode: just you. The **rotate** button issues a fresh token and kills the old one, so a paired phone can be cut off in one tap.

C.5 — The audit ledger

Security-relevant events — settings changes, the network being turned on or off, research collaborations — are written to a **tamper-evident audit ledger**: a local, append-only trail so there's always an honest record of what changed and when.

C.6 — Privacy, restated

Every guard, token, ledger, and database above is a local file under your own user profile. No account, no sync, no telemetry. ELI's default posture is *closed* — offline, gated, and yours. You open doors deliberately, one at a time.

That's the whole assistant — inside and out. You don't need any of the last three appendices to use ELI; just talk to it in plain English, and ask "what can you do?" whenever you're curious. Everything here happens on your computer, for you, privately.

Blueprint — ELI MKXI Architecture

The full structural/architectural map of ELI MKXI. Every claim here is grounded in the actual source tree (file paths are real and clickable). Where something is observed-at-runtime rather than read-from-code it is marked *(runtime)*.

ELI is a **local-first, offline-by-default, model-agnostic** cognitive runtime + assistant GUI (and, as of 2026-06-28, a self-hosted web app — see `server_frontier_roadmap.md`). No cloud, no APIs on the inference path, no hardcoded model. ~143k LOC across 370 Python files (+ `api/server.py`); 208 capabilities (2026-07-01).

---

0. Design principles (the non-negotiables)

1. **Local-first / offline-by-default.** All outbound network is fail-closed at the socket boundary (`eli/core/netguard.py`); networked features opt in.
2. **Model-agnostic.** No model name/size is hardcoded on the inference path; ELI discovers and loads whatever GGUF exists. The model is swappable substrate, not a constant.
3. **Grounded / anti-confabulation.** A large dedicated subsystem exists purely to stop a local model stating things it can't back up (see §9).
4. **Emergent persona.** ELI has a distinctive voice; persona drift/banter is intended. Functional bugs are fixed; the personality is not scripted away.

---

1. Repository map (recursive LOC, role)

Per-package files / LOC measured 2026-06-08.

Package	LOC	Files	Role
`eli/execution`	22.3k	14	Router + executor: intent → action → side effects (executor god-file)
`eli/runtime`	21.1k	79	Grounding spine, evidence, response/introspection surfaces, daemons, self-improvement
`eli/gui`	20.1k	20	PySide6 desktop app, panels, startup/first-boot wizard
`eli/kernel`	13.7k	8	`CognitiveEngine` (the orchestrating core), scheduler, task bus
`eli/cognition`	13.0k	26	Agent bus, 12-stage orchestrator, inference, persona, reasoning
`eli/tools`	8.8k	29	Image engine, news, registry/capabilities, document tools
`eli/memory`	6.9k	13	SQLite + FTS5 + FAISS + knowledge graph + working memory

Package	LOC	Files	Role
`eli/perception`	6.8k	20	Vision (VL), STT (whisper), TTS, **wake word**, **voice/tone**, OS control, gaze
`eli/core`	5.9k	23	netguard, paths, settings, hardware profile, model download, **full_control**
`eli/learning`	3.3k	12	LoRA self-training pipeline (Phi-3 base), dataset build/eval
`eli/planning`	3.3k	21	Proactive daemon, habit scheduler, job/proposal/attention queues
`eli/plugins`	2.0k	28	Plugin manager + bundled plugins
`eli/coding`	1.5k	9	`CodeAgent` — plan→search→verify→repair coding pipeline
`eli/world`	1.5k	26	EliWorld event bus, local world bridge, avatar/ontology
`eli/integrations`	1.0k	9	Optional Ollama client + integration adapters (not on the default path)
`eli/utils`	0.9k	3	logging, shared helpers
`eli/contracts`	0.7k	3	typed pipeline contracts
`eli/system`	0.3k	2	system-level helpers
`eli/cli`	0.1k	2	headless REPL
total	**~143k**	**370**	

The four god-files (refactor targets — see §20)

File	LOC
`eli/execution/executor_enhanced.py`	14340
`eli/kernel/engine.py`	13374
`eli/gui/gui_eli_original.py`	10992
`eli/execution/router_enhanced.py`	7076
`eli/runtime/deterministic_grounding_gate.py`	4292
`eli/memory/memory.py`	4461

---

2. Entry points & launch

Command	Target	Notes
`eli`, `eli-mkxi`, `eli-cli`	`eli.gui.app:main`	GUI (default)
`eli-gui`	`eli.gui.app:main`	GUI script
`python -m eli`	`eli/__main__.py`	dispatches to GUI or headless
`python -m eli --headless` / `-H`	`eli.cli.headless:run_headless`	terminal REPL, no Qt
`./eli.sh`	`python -m eli`	venv launcher
`bash install.sh`	—	venv + deps + clean config seed + verify

Boot path (GUI): `app.py` → (first-boot wizard if no model, `eli/gui/panels/startup.py`) → `gui_eli_original.py:main()` builds `EliMainWindow` → constructs the **`CognitiveEngine` singleton** → loads GGUF per `config/settings.json` → starts daemons.

---

3. The request lifecycle (the spine)

Everything funnels through **`CognitiveEngine.process()`** (`eli/kernel/engine.py`). Two paths diverge by reasoning mode:

```
user input


[Router]  eli/execution/router_enhanced.py :: route()
      priority pipeline → {action, args, confidence, meta.matched_by}

CognitiveEngine.process(action, args, ...)

     PHASE45 fast-path?  (deterministic OS/media/status/job actions)
         → execute_action() → return VERBATIM (no LLM)         [§10]

     non-quick mode OR forced → Internal Orchestrator (full 12-stage)  [§7]

     quick mode (default) →
          AgentBus.dispatch()   [§6]  → bus_result {grounding, agents, ctx}

                Grounding escalation hook  [§9]  (CHAT + low grounding + factual)
                    → web tier / local tier / hedge → may RETURN here

                self-contained action? → return executor result verbatim
                grounded control action? → grounded synthesis
                CHAT → _build_enhanced_system() → broker.infer()/stream
                         → output_governor → response
```

Pipeline stages logged at runtime: `Stage 1 Intent → … → Stage 12 Confidence`. Quick mode short-circuits stages 2–4/HyDE; the orchestrator runs all 12.

---

4. Routing layer (`eli/execution`)

- **`router_enhanced.py`** — `route(text) -> {action, args, confidence, meta}`. Regex-first **priority pipeline** ("explicit priority pipeline installed"): ordered prepasses (web realtime lookup, media, file/PDF, memory, control…) then a core router, with an LLM-intent fallback (`cognition/llm_intent.py`). Holds module state like `_last_used_path` for referential follow-ups.

- **`execution_planner.py`** / `execution_intent_packets.py` — typed `ExecutionPlan` / `RouteDecision` artifacts the bus consumes.
- **`route_authority.py`, `route_contracts.py`, `tool_execution_authority.py`, `operator_policy.py`** — guardrails on what may be routed/executed.
- **`portable_intent_contract.py`, `router_plugin_intents.py`, `executor_plugin_handlers.py`, `media_runtime.py`, `operator_actions.py`** — plugin/media/OS intent wiring.

Key router behaviours (each is a fixed bug → guarded by `tools/eval/cases.yaml`): media controls require **whole-word + command-shape** (no substring "tab" in "metabolise"); bare "do a search" needs a subject; meta-questions stay CHAT; realtime facts ("release date") → WEB_SEARCH.

5–6. Agent Bus & the agents (`eli/cognition/agent_bus.py`)

`AgentBus.dispatch(user_input, intent, …) -> DispatchResult`. Selects an agent set (`_select_agents_for_intent`, or broad fan-out), runs them over a dependency DAG (falls back to flat parallel), aggregates.

`DispatchResult` fields: `agent_results[]`, `action_result`, `memory_context`, `aggregated_confidence` (route+grounding blend), **`grounding_confidence`** (agent-evidence only — the honest "is this backed by anything" signal), `agents_used`, `confidence_label`, `orchestrator_plan`.

14 registered bus agents (`_ALL_AGENTS`):

Agent	Role
`BusMemoryAgent`	recall from SQLite/FTS/FAISS into context
`KnowledgeGraphAgent`	entities/relations from the KG
`SystemAgent`	runs executor actions (WEB_SEARCH, RUNTIME_AUDIT, …)
`OrchestratorAgent`	plan/coordinate grounded synthesis
`FileCodeAgent`	searches the codebase for code-grounded answers
`IntrospectionBusAgent`	live runtime/self introspection
`CapabilityAgent`	what ELI can do (capability manifest)
`ReflectionAgent`	session reflection/insights
`ProactiveAgent`	proactive patterns/signals
`HabitAgent`	learned habits
`SelfImprovementAgent`	self-improvement state
`FrontierAgent`	frontier reasoning hooks
`PluginAgent`	dispatches to registered plugins
`VoiceAgent`	STT/TTS engine state

+ the 15th: `CodeAgent` (`eli/coding/agent.py`) — a *separate* coding pipeline (plan → search → verify → repair, beam/DAG), invoked for CODE_SOLVE / GENERATE_SCRIPT, not a bus agent.

---

7. The Orchestrator — full 12-stage (`eli/cognition/orchestrator.py`)

Runs for non-quick modes (and forced cases). Stages (as logged):

1. **Intent Routing** → action
2. **Persona Lock** → persona/identity fixed for the turn
3. **HyDE** → hypothetical-doc query expansion (`cognition/hyde.py`)
4. **Planner** → which retrieval channels + limits for the mode 5/6/7. **Parallel Retrieval** → keyword + **FTS5** (conversation_turns) + **FAISS** vector + RAG + **KG**
5. **Hybrid Merge** → combine channel hits
6. **Rerank** → `cognition/reranker.py`

7. **Context Assembly** → assembled grounded context 10.5. **Persona Handoff** → persona brief built for generation
8. **LLM Generation** → stream/oneshot via the broker
9. **Confidence** → score vs threshold (PASS/repair)

Quick mode (default) skips 2–4/HyDE and uses the AgentBus directly; the orchestrator is the "deep" path.

---

8. Inference layer (`eli/cognition`)

- **`inference_broker.py`** — the **only canonical inference path**. `broker.infer()`.
- **`gguf_inference.py`** — llama-cpp wrapper, live runtime params, `chat_completion()`, runtime snapshot publishing. Model-agnostic; loads whatever GGUF is configured.
- **`reasoning_modes.py`** — modes: `quick`, `chain_of_thought`, `self_consistency`, `tree_of_thoughts`, `constitutional_ai` (private execution strategy, not shown raw).
- **`chat_model.py`, `context_builder.py`, `context_synthesiser.py`, `user_info_builder.py`** — prompt/context assembly.
- **`llm_intent.py`** — LLM fallback for ambiguous routing.
- Token/context budgeting lives in `engine.py::_build_enhanced_system` (budget-aware persona trim) + `core/dynamic_runtime_budget.py`.

---

9. Grounding & anti-confabulation spine (`eli/runtime` + `eli/core/grounding.py`)

ELI's signature subsystem — keeps a local model from confidently stating unbacked facts.

- **`deterministic_grounding_gate.py`** (4.3k) — the gate; decides when an answer must be backed by evidence vs may be free CHAT; blocks raw evidence/telemetry packets from leaking to the user.
- **`grounding_escalation.py`** — *tiered* escalation (built this cycle): a checkable factual turn with **low grounding** escalates — external fact → web agent, self/project fact → broad local agent fan-out — and **hedges honestly** if nothing grounds it. Trigger is grounding, *not* the (often-high) response-confidence. Env: `ELI_GROUNDING_ESCALATION`.
- **`evidence_ledger.py`, `evidence_store.py`, `evidence_arbitration.py`, `memory_evidence.py`** — what counts as evidence, where it's recorded, conflict resolution.
- **`persistence_gate.py`** — stops internal report-dumps/junk being written to memory (and replayed later).
- **`response_contracts.py`, `response_packets.py`, `response_policy.py`, `final_response_assembly.py`, `final_response_provider.py`, `user_visible_response_surface.py`** — shape/clean the final user-visible answer; format internal "surface packets" into readable text (never raw JSON).
- **`truth_report.py`, `eli_identity_audit.py`, `control_contracts.py`** — runtime truth evidence, identity grounding, control-action contracts.
- **`output_governor.py`, `response_governance.py`, `response_sanitizer.py`, `persona_hygiene.py`** (cognition) — final governance/sanitisation pass.

This is also where the **known weak seam** lives: internal-state→spoken-output leaks (identity confabulation, telemetry dumps) — partially defended, the active frontier (see §20).

---

10. Execution layer (`eli/execution/executor_enhanced.py`)

- `execute(action, args) -> dict` — **~174 action branches**; runtime capability manifest reports **208 capabilities** *(live, read from the manifest each boot)*.
- **PHASE45 fast-path** (`engine.py`) — deterministic OS/media/status/job actions (`VOLUME`, `MEDIA_CONTROL`, `NEXT_MEDIA`, `OPEN_APP`, `DATE`, `SHELL_EXEC`, `ANALYZE_IMAGE`, `CHECK_JOB`, `BACKGROUND_JOBS`, …) bypass the LLM and return the executor result **verbatim** — never paraphrased.
- Action families: media/OS control, file/document (`SUMMARIZE_FILE`, `ANALYZE_PDF[_FOLDER]` incl. per-file mode, `CREATE_DOCUMENT`), web (`WEB_SEARCH`), news, weather, memory (`MEMORY_STORE/RECALL`), code

(CODE_SOLVE, GENERATE_SCRIPT → §coding), background jobs, self/runtime reports (RUNTIME_STATUS, SELF_REPORT, RUNTIME_AUDIT, EXPLAIN_*), vision, image generation, scheduling/reminders.
- **Background jobs** (runtime/background_tasks.py) — submit/get/list; heavy PDF-folder analysis auto-backgrounds; CHECK_JOB surfaces the real result.

Plugins (eli/plugins, manager.py)

Bundled: calendar, document_reader, media, notes, pomodoro, system_stats, tts, weather, web, web_automation. State in config/plugins_state.json (gitignored). Custom agents/plugins are trust-gated.

---

11. Memory subsystem (eli/memory)

- memory.py (Memory, get_memory()) — long-term store over SQLite + **FTS5**, plus failure/observation/habit logging with anti-junk filters.
- vector_store.py — **FAISS** index at artifacts/vectors/index.faiss; embedder models/embeddings/nomic-embed- *(runtime)*.
- **Knowledge graph** — kg_entities / kg_relations (FTS5-backed) in the user DB.
- **Working memory** (cognition/working_memory.py) — session-pinned facts, junk-pin guard; restored across sessions.
- **Persona/personal memory** — runtime/personal_memory_*, persona_updater.

Two databases *(runtime)*: artifacts/db/user.sqlite3 (user-facing) and artifacts/db/agent.sqlite3 (agent/self-improvement). Observed tables include: memories(+_fts), conversation_turns, conversations, kg_entities(+_fts), kg_relations, recall_log, runtime_events, news_articles(+_fts), news_reflections, habits/habit_events/habit_rules, observations, learning_replay, working_memory_pins, user_patterns, session_summaries, corrections, failures, error_tracking, improvements, capability_proposals, code_patches, agent_dispatches, agent_metrics. Retrieval is hybrid: keyword + FTS5 + FAISS + RAG + KG, merged & reranked (§7).

---

12. Perception (eli/perception)

- **Vision** — vision.py (model-agnostic VL: Moondream / Qwen2.5-VL hot-swap; mtmd encoder forced to CPU to avoid CUDA clip segfault), analyze_image.py, ambient_vision.py (ambient toggle), screen_locator.py, gaze_engine.py, analyze_mesh.py, extract_equations.py.
- **STT** — local_whisper_stt.py (faster-whisper small.en, CPU int8, local_only when offline), audio_stt.py, eli_listen.py, voice_worker*.py (wake-word "computer", music-bleed filter, echo gate).
- **TTS** — tts_router.py (Piper voices, e.g. en_US-amy-medium).
- **OS** — os_controller.py (app/window/keyboard/mouse), analyze_csv.py, analyze_pdfs.py.

---

13. Persona system (eli/cognition/persona*)

- persona.txt (authored) + persona.auto.txt (overlay) — the voice.
- persona_updater.py — updates overlay/KG/user-profile each turn (tone signals).
- persona.py, persona_values.py, persona_status.py, persona_hygiene.py.
- tone_analyzer.py — writes tone preference signals (correction/humor/depth).
- Generation injects a budget-trimmed persona + situation brief (engine.py::_build_enhanced_system); a live-runtime fact is injected for model/identity questions so ELI reports the loaded model instead of denying it.

---

14. Daemons & background loops

- **Proactive daemon** (`planning/proactive_daemon.py`) — continuous learning; pattern signals (time habit, topic focus, recurring errors, active project).
- **Self-improvement loop** (`runtime/self_improvement.py`) — learns from logged failures (now filtered of transient/user errors).
- **Habit scheduler** (`planning/habits_scheduler.py`, `habits.py`) — learned routines; guarded against junk-rule replay.
- **Scheduler / task bus** (`kernel/scheduler.py`, `kernel/task_bus.py`).
- **Background tasks** (`runtime/background_tasks.py`) — async heavy jobs.
- **Code monitor** (`runtime/code_monitor.py`), **ambient vision loop**.
- **World event bus** (`world/world_event_bus.py`) — fed confidence/agent events.

---

15. Learning / self-training (`eli/learning`)

LoRA fine-tuning pipeline on a Phi-3 base: `bootstrap_phi3_base.py`, `base_model_resolver.py`, `dataset_builder.py`, `dataset_filters.py`, `merge_reviewed_datasets.py`, `export_trainable_dataset.py`, `training_preflight.py`, `lora_trainer.py`, `lora_trainer_guard.py`, `lora_eval.py`. Turns reviewed interactions into trainable datasets and adapters.

---

16. EliWorld (`eli/world`, `eli/gui` EliWorld tab, `kernel/world_model.py`)

An internal/spatial "world" the agent inhabits (rooms like the Reflection Chamber/Anomaly Room appear *(runtime)*). `world_event_bus.py` receives runtime events; `local_world_bridge.py` bridges to the model; snapshots/ledger/journal under `artifacts/world/`. Experimental/creative subsystem.

---

17. Core infrastructure (`eli/core`)

- **`netguard.py`** — offline-by-default: `guarded_urlopen`, process-wide socket guard (fail-closed on non-loopback while offline), `allow_network()` scoped context for deliberate user-initiated fetches (e.g. model download).
- **`paths.py`, `portable_paths.py`, `legacy_paths.py`, `db_paths.py`** — canonical path resolution (dev vs platformdirs; `ELI_*` overrides).
- **`runtime_settings.py`, `config.py`** — settings load/merge/heal; `DEFAULTS` are clean (offline, no model, wizard on). `config/settings.json` is per-user/gitignored, seeded from `config/templates/settings.template.json`.
- **`hardware_profile.py`, `startup_hardware_optimizer.py`, `dynamic_runtime_budget.py`** — detect GPU/VRAM, pick ctx/gpu_layers/batch.
- **`first_run.py`, `first_run_wizard.py`** — onboarding state.
- **`model_download.py`** — curated GGUF downloader (catalog, resumable, GGUF-magic + size validated, netguard-gated). Install-time menu only — inference stays model-agnostic.
- **`grounding.py`** — `is_grounded_query` etc. (shared grounding helpers).

---

18. GUI (`eli/gui`)

PySide6 (LGPL — not PyQt/GPL, see `pyproject.toml`). `gui_eli_original.py` (main window, 11k LOC), `app.py` (entry/boot), `labs_tab.py` (Experimental), panels in `eli/gui/panels/` (`startup.py` = StartupModelSelectionDialog + FirstBootWizard with curated download, HardwareTuningDock). EliWorld tab, Operator console, Proactive dock. `qt_compat.py` shim allows PyQt fallback from source.

---

19. On-disk layout (`artifacts/`, runtime)

```
artifacts/
   db/{user,agent}.sqlite3        # memory + agent state
   vectors/index.faiss            # semantic index
   conversations/ , conversations/archive/
   incidents/<date>.jsonl         # incident log
   proactive/ , world/{snapshots,ledger,journal}/
   runtime/users/<uuid>/user_profile…   # per-user profile
   runtime_snapshot.json          # live model/runtime truth
   documents/ , scripts/          # generated artifacts
   analyze_image_*/               # vision outputs
config/  settings.json(gitignored) · templates/ · plugins_state.json …
models/  <your>.gguf · embeddings/ · whisper/ · image/
voices/  Piper ONNX
```

---

20. Known seams & weak points (honest)

1. **God-files** (§1) — four files a third of the codebase; the active refactor target. Split by concern, hold a size ceiling, keep a repair-log.
2. **Internal-state → spoken-output seam** — the recurring failure class (identity confabulation, telemetry/packet leaks, ungrounded factual claims). The grounding gate + escalation defend it but don't fully close it on the plain CHAT path. This is *the* frontier item.
3. **Swallowed errors** — many `except Exception: pass`; convert to logged/raised so failures surface instead of hiding.
4. **No capability eval until now** — `tools/eval/` (see `blueprints/eval_harness.md`) is the new measurement layer; coverage grows by turning every bug-log into a case.
5. **Latency vs model size** — on an 8GB GPU a 24B at Q5 offloads few layers → minutes/reply; a 7–9B Q4 is the sweet spot. The model picker should warn on poor VRAM fit.

---

21. Where to make a change (quick index)

To change…	Edit…
how input is routed to an action	`eli/execution/router_enhanced.py`
what an action does	`eli/execution/executor_enhanced.py`
which agents run / how grounded	`eli/cognition/agent_bus.py`
the deep 12-stage retrieval	`eli/cognition/orchestrator.py`
prompt/persona budget, the engine spine	`eli/kernel/engine.py`
anti-confabulation / verify-or-hedge	`eli/runtime/grounding_escalation.py`, `deterministic_grounding_gate.py`
final answer shaping/cleanup	`eli/runtime/*response*`, `cognition/output_governor.py`
memory store/recall	`eli/memory/memory.py`, `vector_store.py`
model loading / inference	`eli/cognition/gguf_inference.py`, `inference_broker.py`
offline/network policy	`eli/core/netguard.py`
paths / settings / model download	`eli/core/{paths,runtime_settings,model_download}.py`
the GUI	`eli/gui/gui_eli_original.py`, `gui/panels/`
eval / regression board	`tools/eval/` (+ `blueprints/eval_harness.md`)

Blueprint - ELI MKXI Full Architecture (ASCII)

The entire system in one drawing, plus the module tree and data layout. Grounded in the real source (see `architecture.md` for prose, `diagrams.md` for the pipeline/memory/gating close-ups). Every layer and box maps to a real path.

A. Full system - layered block diagram

ELI MKXI — FULL ARCHITECTURE 100% local · offline-by-default · model-agnostic ~143k LOC · 370 files · desktop GUI + web app server

PRESENTATION GUI (PySide6) CLI headless Voice I/O gui/gui_eli_original.py cli/headless.py perception/audio_stt (STT) gui/app.py · gui/panels/ (python -m eli -H) perception/tts_router(TTS) labs_tab · EliWorld tab ./eli.sh wake-word "computer" text / command ROUTING execution/router_enhanced.py :: route() regex-first PRIORITY PIPELINE + cognition/llm_intent.py (fallback) route_authority · route_contracts · execution_planner · portable_intent_contract · router_plugin_intents {action, args, confidence, matched_by} KERNEL kernel/engine.py :: CognitiveEngine.process() ← the orchestrating core (12k LOC) scheduler · task_bus · pipeline · state · world_model · self_upgrade [A] FAST-PATH [B] ORCHESTRATOR [C] QUICK PATH (default) PHASE45 12-stage (deep) AgentBus.dispatch() verbatim, cognition/ NO LLM orchestrator.py COGNITION AgentBus (15 agents) + CodeAgent (16th, eli/coding) inference_broker gguf_inference (MODEL-AGNOSTIC GGUF) reasoning_modes: quick · CoT · self-consistency · ToT · const. hyde · reranker · context_builder/synthesiser · working_memory persona.py/_updater/*values*/hygiene (+ persona.txt overlay) GROUNDING SPINE (runtime/) the anti-confabulation core netguard persistence_gate deterministic_grounding_gate(4.3k) grounding_escalation (low-conf → deeper agent tiers + retry) evidence_planner (plan→gather→consume: code/web/memory/runtime) report_pipeline (multi-stage docs: outline→sections→review) evidence_ledger/store/arbitration response_contracts/packets/policy final_response_assembly user_visible_response_surface truth_report output_governor EXECUTION execution/executor_enhanced.py ~166 action branches / 208 capabilities (live) media_runtime · operator_actions · tool_execution_authority · background_tasks PLUGINS(10): calendar document_reader media notes pomodoro weather system_stats tts web web_automation eli/coding :: CodeAgent read / write MEMORY PERCEPTION LEARNING WORLD SQLite + FTS5 vision (VL hot-swap: LoRA self-train EliWorld FAISS vectors Moondream/Qwen-VL) (Phi-3 base) world_event_bus Knowledge Graph STT(whisper) · TTS dataset_builder local_world *working memory os_controller · screen lora_trainer/ bridge user.db + agent.db gaze · ambient_vision eval/guard kernel/world* model

CORE INFRA (cross-cutting — used by every layer) netguard (process-wide offline failsafe) paths/portable_paths/legacy_paths/db_ runtime_settings · config hardware_profile · startup_hardware_optimizer · dynamic_runtime_budget model_download (curated GGUF) first_run/_wizard

BACKGROUND DAEMONS (continuous, started at boot) proactive_daemon pattern signals self_improvement learns from failures autonomy tick (30-min, governed): code_monitor + self-model overlay refresh + goal/scheduler ticks → proposals (observe-only / memory-write; need approval) habits_scheduler/habits routines scheduler · task_bus jobs background_tasks async heavy work reflection loop ambient_vision loop scheduled_tasks (durable overnight/timed) world_event_bus confidence

events

```
OUTPUT    output_governor / sanitiser    inference    (any path)

                    TTS (Piper)  /  GUI render
```

B. The 15 agents

AGENT BUS (cognition/agent_bus.py :: AgentBus.dispatch → DispatchResult) selects a set → runs on a dependency DAG → aggregates grounding_confidence BusMemoryAgent recall: SQLite/FTS5/FAISS → context KnowledgeGraphAgent entities/relations from the KG SystemAgent runs executor actions (WEB_SEARCH, RUNTIME_AUDIT, …) OrchestratorAgent plan / coordinate grounded synthesis FileCodeAgent searches the codebase for code-grounded answers IntrospectionBus… live runtime + gathers identity/awareness audits as evidence (persona summarises — never a data dump) CapabilityAgent capability manifest (what ELI can do) ReflectionAgent session reflection / insights ProactiveAgent proactive patterns / signals HabitAgent learned habits SelfImprovementAgent self-improvement state FrontierAgent frontier reasoning hooks PluginAgent dispatch to registered plugins VoiceAgent STT/TTS engine state CriticAgent second-tier verifier: checks retriever agents' output + CodeAgent (16th) SEPARATE pipeline: plan→search→verify→repair eli/coding/agent (CODE_SOLVE / GENERATE_SCRIPT) — not a bus agent

---

C. Module tree (LOC · key files · role)

eli/ (~143k LOC, 370 files) · api/server.py (FastAPI web app + dashboard) **main**.py · · · · · · · · · · · · · · · · · entry dispatch (GUI | –headless) kernel/ 13.1k the core engine.py 12.0k CognitiveEngine.process() — the spine god-file scheduler.py timed jobs task_bus.py · pipeline.py · state.py · world_model.py · self_upgrade.py execution/ 20.5k route → act executor_enhanced.py 14.3k ~174 actions god-file router_enhanced.py 6.1k priority pipeline god-file execution_planner.py · route_authority.py · route_contracts.py tool_execution_authority.py · operator_actions.py · operator_policy.py media_runtime.py · router_plugin_intents.py · executor_plugin_handlers.py portable_intent_contract.py · execution_intent_packets.py cognition/ 12.3k think agent_bus.py 2.4k 15 agents + dispatch orchestrator.py 12-stage deep retrieval gguf_inference.py 2.1k · inference_broker.py model-agnostic inference reasoning_modes.py · hyde.py · reranker.py · llm_intent.py persona.py/_updater/_values/_status/*hygiene (+ persona.txt, persona.auto.txt)* *context_builder.py* · *context_synthesiser.py* · *working_memory.py* *output_governor.py* · *response_governance.py* · *response_sanitizer.py* *grounded_status.py* · *introspection_agent.py* · *tone_analyzer.py* · *chat_model.py* *runtime/ 18.0k* *grounding spine + daemons (68 files)* *deterministic_grounding_gate.py 4.3k* · *grounding_escalation.py* *evidence_ledger/store/arbitration.py* · *memory_evidence.py* *persistence_gate.py* · *truth_report.py* · *control_contracts.py* *response_contracts/packets/policy.py* · *final_response_assembly/provider.py* *user_visible_response_surface.py* · *personal_memory**.py · reflection.py background_tasks.py · self_improvement.py · code_monitor.py capability_sync.py · pending_proposal.py · runtime_policy.py memory/ 6.6k remember (13 files) memory.py 4.5k Memory · SQLite + FTS5 god-file vector_store.py FAISS index perception/ 5.4k sense (18 files) vision.py · analyze_image/csv/pdfs/mesh.py · ambient_vision.py audio_stt.py · local_whisper_stt.py · voice_worker(_streaming).py · eli_listen.py tts_router.py · os_controller.py · screen_locator.py · gaze_engine.py planning/ 3.1k proactivity (21 files) proactive_daemon.py · habits_scheduler.py · habits.py · jobqueue_cli.py coding/ 1.5k CodeAgent (plan→search→verify→repair, 9 files) learning/ 3.1k LoRA self-training (11 files) lora_trainer/eval/guard.py · dataset_builder/filters.py bootstrap_phi3_base.py · base_model_resolver.py · training_preflight.py plugins/ 1.9k manager + 10 plugins calendar · document_reader · media · notes · pomodoro · weather · system_stats · tts · web · web_automation world/ 1.5k EliWorld (world_event_bus, local_world_bridge)

core/ 4.9k infra netguard.py offline failsafe + allow_network() paths.py · portable_paths.py · legacy_paths.py · db_paths.py runtime_settings.py · config.py · grounding.py hardware_profile.py · startup_hardware_optimizer.py · dynamic_runtime_budget.py model_download.py · first_run.py · first_run_wizard.py gui/ 18.7k PySide6 desktop gui_eli_original.py 11.0k god-file · app.py · labs_tab.py 5.7k panels/ (startup.py: model picker + FirstBootWizard, HardwareTuningDock) tools/ 8.1k image_engine · news · document tools contracts/ 0.7k · cli/ 0.1k · system/ 0.3k · utils/ 0.9k

---

D. On-disk data layout

artifacts/ db/ user.sqlite3 memories(+_fts) · conversation_turns · conversations · kg_entities(+_fts) · kg_relations · recall_log · runtime_events · news_articles(+*fts)* · *news_reflections* · *habits/habit_events/habit_rules* · *observations* · *learning_replay* · *working_memory_pins* · *user_patterns* · *session_summaries* · *corrections* · *failures* *agent.sqlite3* *agent_dispatches* · *agent_metrics* · *improvements* · *failures* · *code_patches* · *error_tracking* · *observations* *vectors/index.faiss* *semantic index (embedder: nomic-embed-…Q4_K_M.gguf)* *conversations/ + archive/* *runtime/users//user_profile…* *per-user profile* *runtime_snapshot.json* *live model/runtime truth* *world/{snapshots,ledger,journal}/* · *proactive/* · *incidents/.jsonl* *documents/* · *scripts/* · *analyze_image/* *config/* *settings.json (gitignored)* · *templates/settings.template.json* · *settings.example.json* · *plugins_state.json models/ .gguf* · *embeddings/* · *whisper/* · *image/ voices/* .onnx (Piper)

E. The flow in one line

INPUT → ROUTER → ENGINE FAST-PATH OUTPUT (verbatim) ORCHESTRATOR retrieval QUICK AGENT BUS MEMORY (SQL/FTS5/FAISS/KG/WM) context GATES: netguard · persistence · grounding · escalation · confidence · governor INFERENCE (broker → gguf, model-agnostic) → OUTPUT → TTS/GUI BACKGROUND: proactive · self-improve · habits · learning(LoRA) · world · scheduler

```
> Companion docs: `architecture.md` (prose, every subsystem) ·
> `diagrams.md` (pipeline / memory / gating close-ups) ·
> `eval_harness.md` (the measurement layer).
```

DAG orchestrator — project-wide execution engine (2026-06-07)

`eli/core/dag.py` was a pure **scheduling** primitive (Kahn topological order, parallel layers, cycle detection, ancestors/descendants, critical-path). It was elevated into a full **execution orchestrator** and wired through the pipeline — additively, so every existing class/function and caller is unchanged.

The orchestrator (`eli/core/dag.py`)

`Orchestrator` / `run_graph(tasks)` execute a set of `Tasks` on their dependency DAG:

- **Parallel intra-layer execution** — independent nodes in a topological layer run concurrently (threads); `max_workers=1` deterministic sequential.
- **Dependency result-passing** — each node fn gets a `NodeContext` with `results` (completed upstream outputs) + a shared mutable `shared` bag.
- **Conditional nodes** — `when(ctx)->bool` skips a branch dynamically.
- **Retries + backoff** — per-node `retries` with exponential `retry_backoff`.
- **Fallback** — a `fallback` fn runs if all attempts fail.
- **Per-node timeout** — total node budget (best-effort; enforced via a worker future).
- **Memoisation** — pluggable `cache` keyed by `cache_key` (status `cached`).

- **Priority scheduling** — higher-priority nodes first within a ready layer.
- **Fail-fast + global time budget** — stop scheduling after a failure / budget.
- **Auto-skip** — a node whose upstream didn't complete is skipped with a reason.
- `critical` **semantics** — a non-critical node's failure doesn't fail the run.
- **Deterministic** `RunReport` — per-node status/attempts/timing + order + layers + critical path + failed/skipped lists + `to_dict()` for observability.

Pure stdlib; the orchestrator does no LLM/IO — the node callables do.

Wired through the pipeline

- **Agents (cognition):** the agent bus runs the 15 agents on the orchestrator (`_run_agents_orchestrated`, default `ELI_AGENT_ORCHESTRATOR=1`, falls back to the layered then flat paths). Dependency-ordered, intra-layer parallel, per-agent timeouts (mode + model-tier scaled), upstream results passed downstream, and a `RunReport` captured per dispatch. Behaviour-preserving: agent fns never raise out (errors → `ok=False`), so edges only order + pass upstream, never gate.
- **Router → executor (observability):** `ORCHESTRATION_STATUS` action ("orchestration status" / "show your agent dag") returns the live engine, execution layers, dependencies, critical path, and last-run telemetry — so ELI can explain its own orchestration honestly.
- **Evidence gathering:** `evidence_planner.gather()` now runs its channels (code / web / memory / runtime) **in parallel on the orchestrator** — each channel isolated (non-critical: one failing yields no evidence, never blocks others), results merged in deterministic `KNOWN_CHANNELS` order, with a sequential fallback. `memory/runtime` (no model calls) genuinely overlap; model-using channels serialise safely on the global `_LLM_CALL_LOCK`.
- **report_pipeline — deliberately NOT parallelised.** All model calls serialise on `_LLM_CALL_LOCK`, so parallel section drafting gives zero speedup (model-bound + globally locked) while adding risk. It stays sequential with its own retries.
- **GUI:** a read-only **Orchestration** sub-tab in Labs (and the `ORCHESTRATION_STATUS` chat action) surface the live layers/deps/critical-path + the last dispatch's per-node RunReport.
- Also on the DAG: the coding planner (`coding/plan_graph.py`) and the LoRA pipeline.

Tests

`tests/test_dag_orchestrator.py` (14) cover every feature; existing dag/coding/agent tests + the full suite stay green. Flags: `ELI_AGENT_ORCHESTRATOR`, `ELI_AGENT_DAG`.

Test/project run → review (GUI + chat)

`eli/runtime/test_review.py::run_and_review()` runs the suite (or a subset), **backs up the prior report**, **writes a timestamped errors file** capturing failing tests (possible errors) under `artifacts/test_reports/`, and returns a **results-driven options menu** (each option carries a chat `command` routing to examine/propose/generate/eval). Surfaced two ways: - **GUI:** a " Test & Review" Labs sub-tab — run the full suite in the background, see totals + failures + backup/error paths, get **ELI's natural-language summary**, and click result-driven option buttons that hand off to ELI in chat. - **Chat:** the `TEST_REVIEW` action ("test review" / "run the tests and tell me what to fix") returns the grounded report (LLM-summarised) + the options menu. - **Option buttons stream into the main chat** (`MainWindow.send_to_chat`) — clicking an option continues the conversation in the Chat tab, not the panel.

Generative actions on the coding agent / planner DAG

- **GENERATE_PROJECT** → `CodeAgent.solve(use_dag=True)`: decompose into a subtask DAG (`plan_graph` over `eli/core/dag`), solve nodes in order, VERIFY the composed module; saves to `artifacts/projects/`. Single-pass `chat` fallback (`ELI_PROJECT_DAG=0`).
- **GENERATE_SCRIPT / CREATE_SCRIPT / WRITE_SCRIPT** → already routed through the verified coding agent (plan → DAG/tree-search → execute → repair → bug-memory).

SELF_IMPROVE routed through the coding agent

`SelfImprovementEngine.propose_via_agent()` (mode `propose`, or auto-detected from "propose/verified fix/fix the failing") turns each recent failure into a coding task and runs the **CodeAgent** (decompose→solve→**verify** via tree-search + execution gate) over them **in parallel on `run_graph`**, returning **verified, propose-only** fixes (gated; nothing applied until "apply self-improvement patch"). This is the delegation pattern realised: SELF_IMPROVE composes the existing coding agent + orchestrator rather than reinventing repair. The Test & Review "Propose verified fixes" option routes here. Artifact-dir resolution is now consistent — `run_test_report.py` and the conftest report hook both honour `ELI_ARTIFACTS_DIR` (no real-folder pollution in tests).

Multi-command + timed commands (2026-06-08)

- **Multi-command chaining** (`eli/runtime/command_splitter.py` + router `multi_command_prepass` → `MULTI_COMMAND`): one utterance chaining imperative commands is split (the "plan"), then the executor routes & runs each segment in order and combines results — the planning step that lets the agents/executor handle several actions from one sentence. Sequential today; `run_graph` is ready for parallel fan-out of independent ("and"-joined) commands.
- **Timed commands** → background workers: any imperative + a future time defers to the durable scheduler (see `runtime_planning_world.md`).

ELI Orchestration & Agents — Full Topology

Supersedes the earlier `agent_bus.md`, which only documented the parallel specialist bus and missed the real `AgentOrchestrator`. Read-only reference; nothing here changes behaviour.

Source files: - `eli/cognition/orchestrator.py` — the real orchestrator (12-stage pipeline) - `eli/cognition/agent_bus.py` — the parallel 14-specialist bus - `eli/kernel/engine.py` — wiring / dispatch gate - `eli/execution/execution_planner.py` — declarative plan model (UNUSED) - `eli/planning/task_planner.py` — planner shim (stub)

Two agent stacks (this is the key thing to understand)

ELI does **not** have one agent system. It has two, selected by reasoning mode and action type:

1. `AgentOrchestrator` — the real orchestrator (`orchestrator.py`)

The 12-stage cognitive pipeline. The engine calls it via `_run_internal_orchestrator` (engine.py:7265, 8982). Components:

- **`PlannerAgent.plan_retrieval()`** (orchestrator.py:53) — produces a **mode-aware retrieval plan**:
  - `fast`: keyword only, no FAISS/RAG, KG only if identity, 1 ReAct iter, skip HyDE
  - `balanced` (default): keyword + semantic + KG, RAG if doc query, 3 ReAct iters
  - `deep`: everything, large budgets, full HyDE, 3 ReAct iters
- **`OrchestratorMemoryAgent`** (orchestrator.py:131) — performs the retrieval: HyDE expansion → keyword/FTS5 + FAISS semantic + document RAG + KG → `hybrid_merge` → `rerank`. **Sequential** by design — the llama_cpp embedder is not thread-safe and segfaults under concurrent daemon-thread calls.
- **`ExecutorAgent`** (orchestrator.py:119) — thin wrapper over `executor_enhanced.execute`.

Flow inside `AgentOrchestrator.run()`: - **Non-CHAT actions** (orchestrator.py:539–740): dispatches the **AgentBus** for specialist evidence (line 561), then runs a **ReAct observation loop** (598–644): runs the executor, asks the loaded LLM `ANSWER` or `TOOL:<action> <args>`, and **chains to the next tool**, accumulating observations. 1 iter in fast mode, up to 3 otherwise. For "grounded synthesis" actions the observations are assembled into context and passed to the LLM; for direct actions the executor result is returned as-is. - **CHAT** (orchestrator.py:742–897): Stage 3 HyDE → Stage 4 planner → Stage 5/6/7 retrieval → Stage 8 merge → Stage 9 rerank → Stage 10 context assembly → Stage 10.5 persona handoff → Stage 11 generation. Private reasoning modes (chain_of_thought, self_consistency, tree_of_thoughts, constitutional_ai) hand off

to `engine._run_chat_reasoning_loop` so the mode-specific algorithm runs. **Note: the AgentBus is NOT called on the CHAT path** — the orchestrator uses its own `OrchestratorMemoryAgent` retrieval instead.

2. `AgentBus` — the parallel 14-specialist fan-out (`agent_bus.py`)

15 agents in `_ALL_AGENTS` (agent_bus.py:3071), each a `_BaseAgent` subclass with `name` + `timeout_s`:

#	name	timeout_s	accesses
1	memory	5.0	SQLite + FTS5 + FAISS; self-gates (`_eli_memory_should_run`)
2	system	8.0	direct action execution (`SYSTEM_ACTIONS`)
3	habit	3.0	`user_patterns` / habit tables
4	self_improvement	3.0	LoRA / self-tuning introspection
5	proactive	3.0	suggestion / anticipation
6	frontier	5.0	frontier / awareness model
7	plugin	6.0	plugin registry (`PLUGIN_ACTIONS`)
8	capability	6.0	capability manifest
9	voice	5.0	TTS / voice subsystem
10	orchestrator	3.0	bus-level planner — emits a plan dict only
11	file_code	4.0	source tree / code introspection
12	reflection	4.0	reflection log / insights
13	introspection	4.0	runtime / cognition self-inspection
14	knowledge_graph	3.0	entity / relation graph

Execution (`AgentBus.dispatch`, agent_bus.py:1533): - **Selective fan-out**: tiny filler chat → `{memory, orchestrator}`; non-chat → `_select_agents_for_intent` minimal set (keyword/action ladder, line 470); plain CHAT → broad fan-out across all `_enabled` agents. - **Parallel**: `ThreadPoolExecutor`, one thread per agent (or `runtime_policy.budget("agent_workers", floor=4, ceiling=32)`). - **Per-agent hard timeout**: `future.result(timeout=agent.timeout_s)` (1585); timeout/exception → failed `AgentResult`, never blocks the response. - **Aggregation** (`_aggregate_confidence`, 1248): per-agent contribution = evidence_quality × evidence_density × learned per-(agent,action) calibration; single-agent cap; empty-bus ceiling; corroboration bonus at 3 contributors. This part is genuinely solid.

When each runs (engine gate `_eli_orch_should_run`, engine.py:8962)

Situation	Path
Quick-mode CHAT, or phatic	**AgentBus directly** (engine.py:8994)
Balanced/deep CHAT	**AgentOrchestrator** — own retrieval; bus *not* used
Non-CHAT action (any mode)	**AgentOrchestrator** → calls bus + ReAct loop
Orchestrator returns None / raises	Falls back to **AgentBus directly**

Planning artifacts — 4 of them, only 1 drives execution

1. **ReAct loop** (orchestrator.py:598) — the *only* planner that actually sequences execution (tool → observe → decide → next tool).
2. **`PlannerAgent.plan_retrieval`** (orchestrator.py:53) — plans *retrieval* budgets, not actions. Used every CHAT turn.
3. **Bus `OrchestratorAgent` plan** (agent_bus.py:1336) — emits an `orchestrator_plan` dict stored in `trace["orchestrator_plan"]` (engine.py:9026) for display / persona handoff / status. **Never executed.**
4. **`execution_planner.build_execution_plan`** (`ExecutionPlan`/`PlanStep`) — now **wired in** as the canonical typed plan. `AgentBus.dispatch` builds it each turn, injecting the proven `_select_agents_for_intent` result as `agent_profile`, and drives `active_agents` *through* `plan.agent_profile` (selection result

unchanged). The plan is surfaced on `DispatchResult.execution_plan`. `EXECUTE_GOAL` also builds a real plan via it.

5. `task_planner.TaskPlanner` — no longer a hardcoded shim; it now delegates to `execution_planner.build_execution_p` so there is a single real plan representation across the codebase.

Where it is actually weak (corrected)

1. **Two retrieval strategies — but NOT a real weakness (corrected).** Earlier framed as duplicate engines; on proper reading both bottom out on the *same* `eli/memory/memory.py` primitives (`recall_memory:1722`, `search_conversations:2533`). The bus's `BusMemoryAgent` is a thin wrapper over the all-in-one `recall_memory` (lightweight, for quick/parallel); the orchestrator's `OrchestratorMemoryAgent` decomposes the same primitives (`keyword_only=True`) + adds RAG + rerank (heavyweight, for deep). A core retrieval bug is fixed once in `memory.py` for both. The fast/deep split is deliberate; forcibly unifying would collapse it for no gain. Left as-is.
2. **The bus is still a single isolated round.** Within one `dispatch`, the 14 agents cannot consume each other's output. The ReAct loop chains *executor tool actions*, not bus agents — so a bus-level dependency (e.g. KG seeded by memory's entities) still cannot be expressed.
3. **Planning partially consolidated (improved).** `execution_planner` is now the canonical typed plan and drives bus selection; `task_planner` delegates to it. Remaining overlap: the engine's `_build_runtime_orchestrator_plan` (rich stage dict) and the bus `OrchestratorAgent` plan still coexist with the typed `ExecutionPlan`. The ReAct loop remains the only planner that actually *executes* a sequence. A future pass could fold the stage dict into `ExecutionPlan.steps` so there is one plan type end-to-end.
4. **ReAct loop fragility — FIXED.** Now validates the proposed `TOOL:<action>` against `SUPPORTED_ACTIONS` (143 registered actions) and stops the loop on an unknown/hallucinated action instead of switching to it; merges `intent["args"]` (preserving originals) instead of overwriting. (orchestrator.py ReAct block.)
5. **Custom-agent timeout override — FIXED.** The override is now `_apply_runtime_policy_timeouts()`, applied to built-ins and **re-applied after `_load_custom_agents()`**, so custom agents get hardware-adapted timeouts too.
6. **Timeouts don't cancel work.** `future.result(timeout)` only stops waiting; a timed-out write-capable agent (memory/habit) can still land a late DB write.
7. **Confidence coupled to a fixed evidence-key schema.** `_evidence_density` only counts known keys; a custom agent returning useful prose under an unknown key contributes 0.0 to grounding.
8. **No early-exit for direct actions; failures debug-only.** Bus waits on the slowest selected agent even when a direct `action_result` is already in hand; timeouts/exceptions log at `log.debug` with no health surface (despite the `agent_metrics` table existing).

Recommended fixes (prioritized)

Priority	Fix	Why
High	Unify retrieval: have the bus `BusMemoryAgent/KnowledgeGraphAgent` and the orchestrator's `OrchestratorMemoryAgent` call **one** shared retrieval module.	Kills the dual-stack divergence (#1) — the single biggest source of mode-dependent bugs.
High	Move `_load_custom_agents()` above the runtime-policy timeout loop (or re-apply the override after loading).	One-line fix for the redistribution timeout gap (#5).
High	Harden the ReAct loop: validate `<action>` against the known action set before chaining; merge (not overwrite) `intent["args"]`; cap/parse defensively.	Closes silent-break + arg-loss (#4).

Priority	Fix	Why
Med	Make the bus optionally two-round for dependent agents (round-1 results passed into round-2 `run()`), gated by a real plan.	Enables KG-seeded-by-memory etc. (#2).
Med	Either wire `execution_planner.ExecutionPlan` in as the bus's selection/sequencing source, or delete it + the shim `task_planner` to cut dead surface.	Resolves the 5-planner fragmentation (#3).
Med	Generic evidence-density floor: count any non-empty payload at low weight so unknown-key/custom agents aren't zeroed.	Makes custom agents first-class to confidence (#7).
Med	Cooperative-cancel token checked before write-capable agents commit.	Closes the late-write hole (#6).
Low	Early-return once a direct `action_result` exists; surface agent health (timeout rate, p95) from `agent_metrics` in RUNTIME_STATUS.	Latency + observability (#8).

Highest leverage: **#1 (unify the two retrieval stacks)** and **#4 (harden the ReAct loop)** — those are where correctness actually drifts today. The bus's confidence math and timeout enforcement are already good and don't need work.

Update — 2026-06-09

- **HabitAgent proactive disconnect fixed.** The bus `HabitAgent` used to read only `get_habit_rules()` (the usually-empty `habit_rules` table), so the chat path was blind to the behaviour the proactive daemon detects into the `habits` table. It now also reads `get_detected_habits()` and emits a `summary` + `detected_habits`; `AgentResult.has_evidence` recognises the new keys (it was judging the agent "no evidence" → dropped), and `DispatchResult.to_context_block` renders the habit summary into the chat context. Habit data is now consistent across the chat agent, `HABIT_STATUS`, and the persona overlay.
- **Goal autogenesis** (`planning/goal_autogenesis.py`) feeds the autonomy/goal-tick stack from ELI's own signals — see `runtime_planning_world.md`.

ELI Agent Algorithms — what each of the 15 (+1) agents actually computes

Created 2026-06-01.

Your framing is the right lens: **the DAG structures the reasoning steps; RAG is the engine that pulls the raw information during those steps.** ELI maps onto this cleanly —

- **DAG layer (structure):** the `AgentOrchestrator` 12-stage pipeline + the `AgentBus` topological layers (`eli/core/dag.py`) + the coding subtask DAG. This is the "search → summarize → check" skeleton.
- **RAG layer (raw information):** a *subset* of the agents are retrievers (`memory`, `knowledge_graph`, `file_code`) plus the orchestrator's own HyDE→FAISS→RAG→rerank pipeline. These pull evidence.

- **Effectors / reporters:** the rest act (`system`, `plugin`) or report grounded runtime facts (`capability`, `introspection`, `frontier`, …).

So there are **15 specialist agents in `_ALL_AGENTS`** plus the `AgentOrchestrator` that sequences them — that's the "14/15".

Roles at a glance

Role	Agents
Retriever (RAG engine)	`memory`, `knowledge_graph`, `file_code`
Effector (acts on the world)	`system`, `plugin`
Grounded reporter (read-only facts)	`capability`, `voice`, `introspection`, `frontier`, `reflection`, `proactive`, `self_improvement`, `habit`
Planner (emits structure)	`orchestrator` (bus-level, descriptive) + the real `AgentOrchestrator`

Every agent shares the same skeleton: a **relevance gate** (cheap keyword/action test → skip with `confidence=0` if irrelevant, so the bus stays fast) → its algorithm → an `AgentResult(ok, confidence, data)`. The bus runs them in **topological layers** with **per-agent hard timeouts**, then `_aggregate_confidence` fuses them: `contribution = evidence_quality × evidence_density × learned calibration`, single-agent-capped, corroboration-bonused.

The 15 specialist algorithms

1. `memory` — hybrid dense+sparse retriever (RAG core) · 5.0s

- **Gate:** `_eli_memory_should_run` (explicit memory actions, memory markers, or 4-word CHAT; blocks stale context bleeding into commands).
- **Algorithm:** `recall_memory` = **FAISS vector search first** (dense), with **FTS5/LIKE keyword search** as a supplement on cold/short queries (sparse); noise-filtered (drops ELI's own reflections/assistant-insights), importance- ordered. Plus `search_conversations` (FTS5), recent turns, session summaries.
- **Out:** `{results, conv_hits, memory_context}`. The primary RAG retriever.

2. `system` — deterministic action effector · 8.0s

- **Gate:** `action` `SYSTEM_ACTIONS` (90 OS/runtime actions); `LLM_ACTIONS` (GENERATE_SCRIPT, etc.) are **deferred** (never run inside the timed parallel phase — they'd time out + double-execute).
- **Algorithm:** delegates to `executor_enhanced.execute(action, args)` behind the security allowlist/sandbox. Confidence 0.92 ok / 0.20 fail. The hands.

3. `habit` — pattern-store reader · 3.0s

- **Gate:** habit/routine/schedule keywords or `OPEN_APP`.
- **Algorithm:** reads `habit_rules` (all) + `habit_events('app_launch', 14d)`; confidence by rule presence. (Frequency mining itself lives in the proactive daemon; the agent surfaces the rules + event volume.)

4. `self_improvement` — failure-cluster surfacer · 3.0s

- **Gate:** self/improve/fix/learn keywords or SELF_* actions.
- **Algorithm:** `analyze_failures(7d, clustered by occurrence_count)` + pending capability proposals. Feeds the self-upgrade loop (see `background_tasks.md`).

5. `proactive` — artifact reader · 3.0s

- **Gate:** proactive/morning-report/insight keywords or PROACTIVE_* actions.
- **Algorithm:** reads `artifacts/proactive/latest_{context,summary,action}.txt`
  - `PROACTIVE_STATUS`. Surfaces the background daemon's findings.

6. `frontier` — cross-system introspection report · 5.0s

- **Gate:** FRONTIER_STATUS / ELI_IDENTITY_AUDIT or "full system status".
- **Algorithm:** `build_frontier_status_report` / `eli_identity_audit` — an import + wiring matrix across runtime/memory/self/proactive/image/world/labs. Deterministic whole-system audit.

7. `plugin` — plugin effector · 6.0s

- **Gate:** `action` `PLUGIN_ACTIONS` (GET_WEATHER, LIST_EVENTS, ADD_EVENT).
- **Algorithm:** `execute(action, args)` into the plugin layer; silent skip otherwise.

8. `capability` — capability-manifest reader · 6.0s

- **Gate:** capability/awareness/"what can you do" or LIST_CAPABILITIES etc.
- **Algorithm:** `execute(LIST_CAPABILITIES | AWARENESS_STATUS | CODE_CHANGES)` → grounded capability surface (no confabulated feature lists).

9. `voice` — config reporter · 5.0s

- **Gate:** voice/tts/stt/mic keywords.
- **Algorithm:** reads `ELI_TTS_ENGINE`/`ELI_PIPER_MODEL`/`ELI_MUTE` + probes `faster_whisper` import. Reports, never emits audio (that's the TTS router).

10. `orchestrator` (bus-level) — descriptive planner · 3.0s

- **Gate:** multi-step trigger groups (e.g. {document, save, open}) or GROUNDED_SYNTHESIS_ACTIONS.
- **Algorithm:** emits a structured `plan` dict (now also surfaced as the typed `ExecutionPlan`). **Excluded from confidence aggregation** (it plans, it isn't evidence). The *executing* planner is the ReAct loop in `AgentOrchestrator`.

11. `file_code` — source-code RAG (grep retriever) · 4.0s

- **Gate:** grounded query or code/file/pipeline/db keywords.
- **Algorithm:** builds query-derived regex patterns, **greps a canonical `files_map`** of ELI's own modules (agent_bus, engine, orchestrator, memory, router, …), returns `file:line: content` snippets. Lets ELI answer "which file does X" from real source, not memory.

12. `reflection` — reflection-log reader · 4.0s

- **Gate:** reflection/pattern/noticed/insight keywords or grounded query.
- **Algorithm:** recent `observations` (8) + session summaries (3) → insight list. ELI's "what I've noticed about how you use me."

13. `introspection` — self-model reporter · 4.0s

- **Gate:** architecture/pipeline/"how do you work" or EXPLAIN_*_RUNTIME.
- **Algorithm:** `IntrospectionAgent` → `get_pipeline()` + `get_memory_stats()` + `get_runtime()` → grounded pipeline description. Powers honest self-explanation.

14. `knowledge_graph` — graph retriever (structured RAG) · 3.0s

- **Gate:** non-empty KG.
- **Algorithm:** `context_for_prompt(query)` over `kg_entities/kg_relations` (FTS5 fuzzy entity match + relation context). **DAG edge `memory → knowledge_graph`:** it now seeds the query with `intent["_upstream"]["memory"]` hits (topological layering), so the graph lookup is conditioned on what memory just surfaced.

15. `critic` — second-tier verifier (corroboration check) · 2.0s

- **Gate:** 2 grounded upstream sources — self-gates to nothing on trivial turns, so it never fires when there's nothing to cross-check.
- **Algorithm:** the one bus agent that reasons over **other agents' output** (`intent["_upstream"]`) rather than the raw query — **deterministic, no LLM**. It pulls the representative text each retriever surfaced this turn, measures **pairwise term agreement**, and emits a **corroboration-vs-contradiction** signal. **DAG edges:** depends on `memory/system/knowledge_graph`, so it runs in a **downstream topological layer** and sees their results — giving the agent DAG a real second tier instead of a flat fan-out.

How the DAG sequences the RAG (your "search → summarize → check")

1. **Route** → action + reasoning mode.
2. **`AgentOrchestrator`** (non-quick) runs the staged pipeline; for non-CHAT it dispatches the **bus** then runs a **ReAct loop** (tool → observe → decide → next tool) — the executing DAG of steps.
3. **`AgentBus`** builds the **agent DAG** (`_AGENT_DEPENDENCIES`), runs agents in **topological layers**, passing each layer's results downstream (`memory` → `knowledge_graph`). Retrievers pull evidence; reporters ground it; effectors act.
4. **Grounding/evidence layer** gates the synthesis (`output_violates_evidence`).
5. **Coding tasks** decompose into a **subtask DAG** (`plan_graph`) and heavy ones run on **background workers** — see `dag.md` + `background_tasks.md`.

That is the literal realisation of "DAG structures the autonomous reasoning steps while RAG pulls the raw information."

Aspirational next steps (genuinely novel combinations)

These are the "never-before-put-together" moves the current architecture is one step away from:

1. **Bias/verification agent as an explicit DAG layer.** **Shipped** — this is now the `critic` agent (#15 above): it depends on the retrievers (`memory/system/` `knowledge_graph`) and runs *after* them as a topological layer — literally "search → summarize → **check for corroboration/contradiction**" — feeding the grounding signal. (Originally proposed here as future work; now built.)
2. **A `web` retriever agent** (WEB_SEARCH as a first-class RAG node) so external retrieval joins memory/KG/file_code in the same layer, with the verify layer cross-checking sources.
3. **Calibration-aware selection.** The bus already learns per-(agent,action) calibration; feed it back into *selection* so chronically-zero agents are dropped from the DAG for that action (closes the learned-trust/unlearned- selection gap noted in `orchestration_and_agents.md`).
4. **Bus coding bug-memory unification.** The coding engine's `bug_memory` and the agents' `self_improvement` failure log are two halves of one "what went wrong and how we fixed it" store — merging them gives ELI a single experiential memory across chat and code.
5. **Orchestrator stage-dict → typed `ExecutionPlan` DAG.** Fold the engine's rich stage plan into `ExecutionPlan.steps` as a true DAG so one plan type drives the whole pipeline end-to-end.

ELI Memory Subsystem

`eli/memory/` — 7.0k LOC, 13 files. The persistent substrate: relational + full-text + vector + graph, all local SQLite/FAISS. Companion to `project_overview.md`.

Files

File	LOC	Role
`memory.py`	4.5k	the `Memory` god-class + `DBPaths` + module facade
`knowledge_graph.py`	564	entity/relation graph (KG)
`habits_memory_db.py`	470	
`vector_store.py`	411	FAISS vector index + embedder
`__init__.py`	0	
`system_index.py`	238	indexed apps/executables/files
`memory_truth.py`	190	
`memory_adapter.py`	131	compat adapter
`memory_service.py`, `sqlite_memory.py`, `stores.py`, `populate_memories.py`	small	helpers/compat

The `Memory` class (`memory.py`)

A single metaclass-backed class (~50 public methods) that owns essentially every persistent concern:

- **Semantic memory**: `store_memory`, `add_memory`, `recall_memory`, `search_memory(ies)`, `get_recent_semantic_memor` `adjust_weight`, `apply_weight_decay`.
- **Conversation**: `add_conversation_turn`, `store_conversation`, `get_conversation_history`, `get_recent_conversation`, `get_recent_turns_since`, `search_conversations`, `get_turns_for_day`, `save_session_summary`, `get_session_summaries`.
- **Habits**: `log_habit_event`, `get_habit_events`, `add_habit_rule`, `get_habit_rules`, `record_habit_run`.
- **Self-improvement / learning**: `log_learning_event`, `log_failure`, `log_correction`, `add_observation`, `log_improvement`, `add_capability_proposal`, `propose_capability`, `get_pending_proposals`, `get_recent_failures/improvements/observations`.
- **Episodic/semantic/reflective aliases**: `store_episodic`, `store_semantic`, `recall_semantic`, `store_reflective`.
- **Stats / routing**: `get_stats`, `get_dashboard_counts`, `get_db_routing_info`.

`vector_store` is a lazy property; the KG is integrated via `knowledge_graph.get_knowledge_graph()`.

Schema (24 tables)

`memories` (+ legacy `memory`), `conversation_turns` (+ `conversations`), `session_summaries`, `kg_entities`, `kg_relations`, `habit_rules`, `habit_events`, `habits`, `failures`, `corrections`, `improvements`, `observations`, `capability_proposals`, `learning_replay`, `user_patterns`, `desktop_apps`, `executables`, `recent_files`, `user_dirs`, `error_tracking`, `recall_log`, `events`, `semantic`. FTS5 virtual tables back conversation and KG search.

`recall_memory` — the hybrid retriever (memory.py:1722)

The shared retrieval foundation (see `orchestration_and_agents.md` for the two strategies on top):

1. **FAISS first** (Stage 5 vector primary). FTS5/LIKE runs only as a *supplement* when the vector index is empty/cold or returns `< limit//2` hits. `keyword_only=True` skips FAISS entirely (the orchestrator runs its own `semantic_search`, so running FAISS here would double-search with a mislabeled source).

2. **Noise filtering** (important): excludes `assistant_insight`/`episodic`/ `reflection` kinds, `orchestrator` source, and `reflection`/`assistant_insight`/ `session_summary` tags, and rows longer than 1500 chars — so ELI's own old responses/reflections never resurface as "recalled user memories". This was the fix for the "Immutable Techniques" contamination class of bug.
3. **Importance-weighted ordering** via `COALESCE(importance, 0.5)`.

Heavy inline column-detection (`_memory_table_columns`) guards against schema drift across versions — defensive, but a sign the schema has churned.

Vector store (`vector_store.py`)

FAISS `IndexFlat`, embeddings via a local nomic embedder (llama_cpp). Notable: - `_embed_lock` (RLock) serializes embedding — the embedder is **not thread-safe** and concurrent calls segfault (this is why the orchestrator retrieval is sequential). - Metadata canonicalized to **`meta.json`** (migrated from legacy `meta.pkl`). - Singleton via `get_vector_store()`; shutdown-aware (skips embedding during teardown); `reset_vector_store()` for rebuilds.

Knowledge graph (`knowledge_graph.py`)

`kg_entities(name,type,aliases,description,confidence)` + `kg_relations(subject_id, predicate, object_id, weight, source)` — a subject-predicate-object graph. FTS5 over entities (with insert/update/delete triggers) for fuzzy `search_entities`. `upsert_entity`, `context_for_prompt` (lightweight SQLite-only prompt context — no embedding). Stop-word list prevents common words becoming entities.

Truth layer (`memory_truth.py`)

`inspect_sqlite` / `inspect_vector_store` — read-only authority/inspection used by status surfaces; reads `vectors/meta.json` preferentially, falls back to legacy pickle. Backs `truth_report` and the memory-status surfaces.

Weight decay & consolidation

`apply_weight_decay(decay_factor=0.98, min_weight=0.05, older_than_days=7)` — a multiplicative SQL update (`weight = MAX(min, weight*factor)` for old, low-importance rows). **There is no real episodic→semantic→KG consolidation**: decay is the only forgetting mechanism, and promotion across stores is manual (`store_episodic`/`store_semantic` are aliases, not a pipeline). This is a known gap ELI's own grounding admits.

Honest assessment

- **Strong:** genuinely hybrid (vector + FTS5 + KG) on one local SQLite foundation; the noise-filtering in `recall_memory` is the right instinct and fixed real contamination; embedder serialization correctly avoids the segfault.
- **Weak:**
  1. `memory.py` is a **4.5k-line god-class** spanning ~8 unrelated concerns (semantic, conversation, habits, learning, failures, capabilities, system index). Wants to be split along those seams.
  2. **Schema sprawl / redundancy** — `memories` *and* `memory`, `conversations` *and* `conversation_turns`, plus a standalone `semantic` table. The inline column-detection everywhere is compensating for schema instability.
  3. **No consolidation pipeline** — only multiplicative decay; memories don't graduate into the KG automatically.

Update — 2026-06-09

- **Detected habits readable** (`Memory.get_detected_habits(min_count, limit)`): the proactive daemon fills the `habits` table via `detect_habits()`, but HABIT_STATUS, the persona overlay, and the bus `HabitAgent` all previously read only the (usually empty) `habit_rules` table → "no habits detected". `get_detected_habits` reads the real `habits` table with a meta/introspection denylist (filters SELF_REPORT/MEMORY_STATUS/EXPLAIN_* noise from the user testing ELI), so genuine behaviour (media, app launches, screenshots) surfaces.
- **FAISS persistence bug fixed:** `_eli_persist_loaded_vector_store` was writing vector metadata with `pickle.dump` to `meta.json` while `_load_meta` reads JSON — so a rebuild corrupted the index and the next load silently auto-rebuilt (accumulating phantom vectors). Both write sites now use the canonical `_dump_meta` (JSON); the index re-syncs cleanly to the memory count. (This also closes a pickle-RCE-on-load vector the codebase had deliberately retired.) KG populator enriched (broader, clean entity extraction from `user_patterns`).

ELI Perception — Vision, Voice, Wake Word, Tone, OS Control

`eli/perception/` (~20 files). ELI's senses and hands: local vision, speech-to-text, text-to-speech, a **self-trained wake-word detector**, a **voice-profile/tone** subsystem, and OS control. All local, no APIs, no third-party accounts.

Files

File	LOC	Role
`audio_stt.py`	~1.6k	STT + mic capture + ducking + adaptive pause + wake/voice capture
`wakeword.py`	~450	**self-trained, music-robust wake-word detector** (openWakeWord features + custom head)
`voice_profile.py`	~420	**prosody + labelled-emotion** (tone/question detection)
`tts_router.py`	667	
`vision.py`	692	
`os_controller.py`	573	
`screen_locator.py`	389	locate UI elements on screen
`gaze_engine.py`	298	
`log_rotation.py`	215	log housekeeping
`analyze_pdfs/image/mesh/csv.py`	~600	file-type analysers
`ambient_vision.py`	207	periodic screen glances (off by default)
`local_whisper_stt.py`	257	
`voice_worker(_streaming).py`, `eli_listen.py`, `extract_equations.py`	small	workers/helpers

Vision (`vision.py` + `ambient_vision.py` + `analyze_image.py`)

The working local-vision stack (see memory `eli-image-analysis`): - VL model (default Qwen2.5-VL, configurable) loaded via `Qwen25VLChatHandler`; **CPU clip forced** by monkeypatching `mtmd_cpp.mtmd_init_from_file(use_gpu=F` — the GPU mtmd clip path **segfaults** on compute-7.5 cards. - **Hot-swap**: unload text model → load VL → infer → restore in `finally`, holding `gguf_inference._LLM_CALL_LOCK`. - **Co-resident fast path**: a small model (Moondream2 Q4) loaded first, with the text model's ctx capped (`vision_coresident_text_ctx`) so

both fit 8GB; `prefer_fast=True` uses it without a swap. - Images downscaled to `vision_max_image_px` (1280) to avoid context overflow; `repeat_penalty/top_k/top_p` set to kill a repetition loop. - `ambient_vision.py`: optional periodic screen glances (OFF by default). A guarded daemon re-reads the toggle/interval each cycle and **skips a glance whenever the shared LLM lock is busy** (so it never steals the model mid-reply). Stores a short description as memory for rolling awareness.

Speech-to-text (`audio_stt.py`, `local_whisper_stt.py`)

A large, feature-dense STT module: - Mic capture + Whisper transcription (`local_whisper_stt`). - **Output ducking** — lowers system sink volume while listening (`_eli_duck_output`/`_eli_restore_output` via `wpctl`) so ELI doesn't hear itself. - **Echo/self-hearing suppression** (`_eli_echo_like_assistant_output`, `_eli_media_probably_audible`). - Transcript cleanup: `_cleanup`, `_collapse_repeated_phrase`, `_eli_fast_command_alias` (maps spoken phrases to commands), `_is_safe_direct`/`_allow_direct_chat` (gates which utterances bypass to chat). - ALSA stderr suppression to keep the console clean. - **Duration-adaptive end-of-phrase pause** (`_listen_adaptive_pause`, 2026-06-08) — stock `sr.listen()` reads `pause_threshold` once per phrase, so one value can't be both snappy for commands and tolerant of long dictation. A faithful copy of sr's capture with a single dynamic condition: short commands finalise after `ELI_STT_SHORT_PAUSE` (0.5s) of silence; a prompt speaking past `ELI_STT_LONG_AFTER` (12s) needs `ELI_STT_LONG_PAUSE` (2s), so a mid-sentence pause no longer cuts it. Flag-gated (`ELI_STT_ADAPTIVE_PAUSE`) with fallback to stock `listen()`. - **Generic mic-capture script** — one mechanism (no second microphone) drives both wake-word enrollment and voice/emotion training: a list of `{prompt, sink}` steps, each captured (past the TTS-echo guards) clip routed to its sink, the next cue spoken, a `done` callback after the last. `begin_capture_script` / `begin_wake_enrollment` / `begin_voice_training`. - **Acoustic wake hook** — when unarmed and a wake model is trained, the captured audio is scored by `wakeword` and, if it fires, the wake word is injected so the existing `VoiceGate` arms — catching the wake word even when whisper transcribed the music. **Per-turn tone hook** — on a real command, `voice_profile.classify_tone` is run and published on a side-channel for cognition (below).

Wake word (`wakeword.py`, 2026-06-08) — self-trained, robust over music

Transcription-based wake matching can't hear "computer" over loud music (whisper transcribes the music). Instead ELI **trains its own** detector, 100% locally: - **Positives**: the wake phrase synthesised by ELI's OWN **Piper TTS** across several voices and speeds. - **Augmentation (the robustness)**: each positive is mixed with noise/music at random SNRs, so the classifier learns to spot the wake word *through* a music bed. - **Features**: openWakeWord's open, bundled melspectrogram→embedding extractor (no account, no download barrier). - **Custom head**: a small torch classifier ELI owns; `train_model()` → `models/wakeword/eli_wake_head.pt` (gitignored). `WakeDetector.score_audio/is_wake` slide a 1.5s window. Validated: clean wake = 1.00, wake+loud music (3 dB) = 1.00, hard-negative/music-only = 0.00. - **User-settable phrase** — `get/set_wake_phrases` persists any phrase ("change the wake word to athena"); it feeds both this model and the transcription matcher. - **Personalisation** — enrolled real-mic clips of the user are folded in as heavily- weighted positives. - Actions: `WAKE_SET` / `WAKE_TRAIN` / `WAKE_ENROLL`. Fully fallback-safe (`ELI_WAKE_ACOUSTIC=0`; no model → the transcription matcher).

Voice profile + tone (`voice_profile.py`, 2026-06-08) — foundation for emotion

Deliberately SEPARATE from the wake word — this is *how* the user speaks, the basis for tone/emotion and question-vs-statement: - **Prosody (real, numpy only)**: per-clip autocorrelation **F0/pitch** track, energy, voiced ratio, speaking rate, and a **terminal-pitch slope** (`analyze_prosody`). - **Question vs statement** (working): rising terminal pitch question (`question_or_statement`). - **Labelled emotion**: `add_labelled_sample` + a nearest-centroid classifier (`train_emotion_classifier`) over neutral/happy/angry/excited/sad — robust for the small data a quick enrollment yields, upgradeable to a NN without touching callers. - `classify_tone` returns emotion + confidence, arousal, and the question/statement cue. `build_profile` learns the user's baseline so tone is scored *relative* to how they normally sound. - `TRAIN_VOICE` runs one unified guided session (wake reps + the same line said in each emotion); modules stay separate, the user does one flow. - **Wired into cognition**: STT publishes the per-turn read on a fresh, timestamped side-channel (`set_last_tone`); `engine._build_enhanced_system` reads `get_last_tone` and prepends a speaker-voice cue so ELI adapts its

delivery (warmer if upset, brisk if energetic). `ELI_VOICE_TONE=0` disables. Complementary to the text-based `cognition/tone_analyzer` (persona preferences), not a duplicate.

Text-to-speech (`tts_router.py`)

Multi-backend router with graceful fallback: **Piper** (packaged binary under `tts_piper/` or `models/tts/piper`, voice-dir resolution) → **espeak-ng / espeak**. Resolves voices from several candidate dirs so a packaged build or a source checkout both work.

OS control (`os_controller.py`)

Platform-aware (Linux/Windows/macOS) screenshot, volume, keyboard, clipboard. `take_screenshot(region)` picks the right tool per platform (`gnome-screenshot`, PIL `ImageGrab`, platform CLIs); fails gracefully where a capability isn't available (e.g. area screenshots on some Windows installs). `screen_locator.py` finds UI targets for click/automation.

File analysers

`analyze_pdfs`, `analyze_image`, `analyze_csv`, `analyze_mesh`, `extract_equations` — typed handlers behind the `ANALYZE_*` actions, feeding extracted content (text/OCR/structure) into the grounded pipeline.

Honest assessment

- **Strong:** this is the layer that makes ELI genuinely multimodal and local — eyes (VL + OCR + ambient), ears (Whisper + ducking), mouth (Piper/espeak), hands (cross-platform OS control). The vision co-residence + hot-swap + CPU-clip workaround is real engineering against hard 8GB-GPU constraints.
- **Weak / watch:**
  1. **Vision latency** — the hot-swap unloads/reloads the text model per glance; even the co-resident path runs CPU clip (~3.5s). Acceptable, not fast.
  2. `audio_stt.py` is a **1.48k-line** module mixing capture, ducking, echo suppression, command aliasing, and cleanup — wants splitting.
  3. **Residual "7B" comment** in `ambient_vision.py` ("the 7B vision model hot-swaps…") — cosmetic, but inconsistent with the model-agnostic line; should read "the vision model". (Flagged for a future scrub.)
  4. `gaze_engine.py` is experimental and off by default — fine, but it's carried weight.
  5. Platform tools are detected at call time; a machine missing `gnome-screenshot/wpctl/piper` degrades silently to fallbacks (good) but there's no single "what perception capabilities do I actually have here?" probe surfaced to the user.

Update — 2026-06-09 (VRAM-aware STT; drag-drop keeps the full path)

- **STT is VRAM-aware** (`local_whisper_stt`). faster-whisper defaults to GPU only when the card is big enough for whisper AND a typical main model (12 GB total VRAM), else CPU — so on an 8 GB card whisper no longer preloads ~2 GB and starves the main model (observed: free_vram 4083 MB → gpu_layers=11; after the fix free_vram 7092 MB → gpu_layers=99). `small.en` int8 on CPU is ~1–2 s for a short command. Explicit `ELI_WHISPER_DEVICE` still wins; `ELI_WHISPER_GPU_MIN_MB` tunes the threshold.
- **Drag-and-drop keeps the FULL path.** A file dropped into the chat input now expands to `[File: <full path>]` followed by the inlined content (was content + basename only), so "fix/examine/edit this file" can resolve the file on disk while "summarise this" still has the content. PDFs keep their path too; combined with the FIX_FILE last-file memory, a bare follow-up "fix it" recovers the file named a turn earlier.

ELI Grounding & Evidence Layer

The anti-confabulation system — a deterministic evidence scaffold wrapped around the probabilistic model. This is ELI's most distinctive subsystem and the part closest to genuinely frontier. Spans `eli/runtime/` and `eli/cognition/`.

The core idea

For "grounded" actions (status, runtime, memory, identity, control), ELI does **not** trust the LLM to state facts. It (1) gathers deterministic evidence from the live system, (2) can answer directly from that evidence without the LLM at all, and (3) when the LLM does generate, validates the output against the evidence and rejects/repairs contradictions. The LLM becomes a phraser, not a source.

> **Correction (2026-06-08): the "bypass" is PARTIAL and MODE-GATED — not a blanket bypass.** Earlier wording here and in `what_eli_is.md` ("bypasses the language model entirely", "without the LLM at all") overstated it. The running reality, verified against the engine: the deterministic verbatim return fires for a **subset** of actions and is **mode-gated** — - a small **`_verbatim_always_actions`** set (deep introspection like `EXPLAIN_MEMORY_RUNTIME`) returns verbatim in **every** reasoning mode; - **`_deterministic_direct_payload_actions`** (status/reports + router-fast/command actions: DATE/TIME, VOLUME, WRITE_NOTE, window/media/file ops, NEWS/MORNING_REPORT, self-report …) return **verbatim in *quick* mode** and **synthesise in non-quick** modes (`engine._bypass_persona` + `_is_grounded_control_nonquick`); - everything else is the model's to phrase.
>
> So the model IS in the loop for most conversational turns; the "phraser, not a source" principle holds where the deterministic path actually fires, not universally. A 2026-06-08 fix corrected the membership: the command actions were in the wrong set, so quick mode re-synthesised them, corrupting results ("Wrote note"→"Bought note") and adding latency. They are now verbatim in quick mode, synthesised in others.

Components

`runtime/deterministic_grounding_gate.py` (4.3k LOC)

The deterministic renderer. `render_action(action, args, user_input, mode_label)` produces a grounded answer for control/status actions directly from runtime data (settings, runtime snapshot, DB counts, GPU line) — bypassing the model. `install(CognitiveEngine)` wires it into the engine. `_eli_v14_runtime_data()` assembles the live config block (model_path, n_ctx, gpu_layers, …).

> **Code-health flag:** the file contains **seven** `render_action` definitions (lines 350, 1058, 2837, 3288, 3653, 3875, 4222), each marked `# type: ignore[override]`. They are stacked successive redefinitions where the last one wins — the file grew by appending new versions rather than editing in place. It works, but it's the single clearest example of the "added beside, not folded in" pattern, and it makes the effective code path hard to trace. Prime consolidation candidate.

`runtime/control_contracts.py` (943 LOC)

The deterministic control path: - `is_control_action` / `route_control_text` — recognise control/status intents. - `build_control_evidence(engine, action, args, …)` — gather the evidence packet (runtime paths, DB state, bus result, trace). - `output_violates_evidence(text, evidence_text)` — the gate that returns True when LLM output contradicts/omits the evidence; the engine uses this to reject a hallucinated answer. - `compact_evidence_answer` / `finalise_control_result` — assemble the final grounded response.

`runtime/evidence_ledger.py` (394 LOC)

A persistent SQLite ledger of evidence events: `record_event`, `recent_events`, `repeated_event_signals` (detect recurring issues over N days), `status_evidence`, `artifact_snapshot`. Gives ELI a durable, queryable record of what actually happened.

`runtime/evidence_arbitration.py` (195 LOC)

`EvidenceItem` + `arbitrate_evidence(limit)` + `build_evidence_context_text` — scores and merges competing evidence into a single context block. Pairs with the agent-bus confidence aggregation (`_score_tool_result`).

`runtime/memory_evidence.py` + `runtime/retrieval_packets.py`

`collect_memory_evidence` / `build_memory_evidence_text` turn memory hits into an evidence block. `retrieval_packets` builds `StagePackets` for each retrieval stage (parallel-retrieval, hybrid-merge, rerank, source-trace) — provenance so the pipeline can show *where* a fact came from.

`cognition/output_governor.py` (1224 LOC)

Post-generation governor: `govern_output(text, is_grounded)`, `normalize_assistant_text`, `validate_against_evidence` plus a family of drift-repair functions — `strip_generic_ai_identity_drift` (kills "As an AI language model…"), `repair_local_persona_drift`, `repair_self_user_confusion` (fixes the model conflating itself with the user), `clean_response_style`. The last line of defence before text reaches the user.

`cognition/grounded_status.py` (644 LOC)

`direct_grounded_answer(user_text)` — fully deterministic answers for identity / memory-inventory / status questions, assembled from the DBs (`format_user_identity`, `format_memory_inventory`, table distributions). No LLM.

`runtime/grounded_remediation.py` (1.6k LOC)

The failure→repair loop. When an action fails (app won't open, path missing, browser/IDE absent), it: `diagnose_app/path/browser/ide` → `build_repair_plan` → `offer_for_result` ("want me to install X?") → `handle_confirmation` (consumes yes/no) → `execute_pending_plan`. Stateful pending-repair tracking + `explain_last_failure`. This is what makes failures conversational and recoverable instead of dead ends.

How it fits the pipeline

1. Router classifies action. Control/grounded actions enter the deterministic path.
2. `build_control_evidence` / `collect_memory_evidence` gather facts.
3. PHASE45 deterministic bypass: for many status actions `render_action` / `direct_grounded_answer` answer **without** the LLM.
4. If the LLM generates, `output_violates_evidence` + `validate_against_evidence` gate it; `govern_output` cleans drift.
5. `grounded_remediation` handles action failures with an offer/confirm loop.
6. `evidence_ledger` records the event for future `repeated_event_signals`.

Honest assessment

- **Strong (genuinely):** very few local-LLM projects build a deterministic evidence layer at all, let alone one this thorough — direct-answer bypass, output-vs-evidence rejection, drift repair, a persistent ledger, and a conversational remediation loop. This is the subsystem that most justifies the "frontier" label.
- **Weak:**
  1. The **seven stacked `render_action` overrides** in a 4.3k file — the effective behaviour is whatever the last definition does; the earlier six are dead weight that obscure the real path. Consolidate to one.
  2. **Overlapping surfaces** — `grounded_status`, `control_contracts`, `deterministic_grounding_gate`, and the `runtime/*_response` / `*_surface` modules all render grounded answers with partial overlap. The boundaries between "who renders the final grounded string" are blurry.
  3. Confidence/grounding is heuristic (additive score + threshold + `output_violates_evidence`), not a formal proof — fine, but it's a gate, not a guarantee.

Update — 2026-06-09

- **Self-patch only patches what it can ground.** `generate_code_patch` extracts the target file from the error traceback; when a failure has no in-project file (e.g. an HTTP/connection error, "No commands to run"), it used to ask the model anyway, which **invented** a path (phantom `api_client.py`) that then failed to apply. It now returns `no_groundable_file` and routes the failure to goal-autogenesis / self-heal instead; when a file IS grounded it is **pinned** in the prompt so the model can't drift to a different path.
- **Banter guard:** the `llm_intent` resolver occasionally maps playful input to a generative action; the engine now downgrades GENERATE_SCRIPT/PROJECT/DOCUMENT/CODE_SOLVE to CHAT when the text carries no create-intent keyword (real "write me a script" is preserved).
- **Compact grounded synthesis carries a VOICE primer** (pulled live from the canonical persona) so factual/introspection answers sound like ELI without losing the EXACT-FACTS contract that pins every number/path/table/DB to the evidence.

Update — 2026-06-09 (deterministic reports surfaced verbatim; examiner correctness)

- **Verbatim guard for deterministic grounded reports.** `_get_chat_response` now returns the EXAMINE_CODE tiered report and the FILE_AUDIT file-count VERBATIM when they appear in the evidence, and surfaces the FIX_FILE success event as a real outcome — instead of letting the quick chat path re-narrate them. This killed three live confabulation classes: EXAMINE_CODE inventing "findings" that were actually existing code comments; FILE_AUDIT inventing files-with-bugs that don't exist; FIX_FILE narrating "I cannot fix" after it had already written the file.
- **Examiner correctness (`code_examiner.py`).** Tier-2 now drops pyflakes star-import noise ("may be undefined, or defined from star imports" — ~800 false positives on the GUI → 0). Tier-3 reviews the WHOLE file in line-correct windows (was only the first ~MAX_FILE_CHARS, so god-files were ~90% unreviewed), each source line prefixed with its REAL number, and rejects any finding citing a line outside the shown window (kills the "engine.py:132 undefined 'ov' " hallucination class — line 132 is `or result.get("error")`).
- **Routing:** "is there any issues with the files?" / "check the code for bugs" now route to EXAMINE_CODE (the real examiner), not FILE_AUDIT; fix-intent ("fix the bugs in foo.py") stays FIX_FILE.

Blueprint — Code-Mode Execution Layer for ELI (gap analysis)

Status: draft. Date: 2026-06-11. REVISED after auditing the codebase: an earlier version of this doc proposed building machinery ELI already has. This version is a GAP ANALYSIS — what exists, and the small genuine delta — not a greenfield design. Original ELI work throughout; the only outside idea is the general principle of deterministic gating, which ELI already implements its own way.

BUILT (2026-06-11)

Core implemented, gated, and tested (17 tests, `tests/test_code_mode.py`): - **Restricted exec (the one new piece):** `eli/coding/restricted_exec.py` — Full-Control-gated (`is_full_control()`, the GUI toggle), AST-whitelist (10 escape classes rejected: import/ eval/open/getattr/dunder-attr/`__import__`/def/lambda/with/exec), restricted namespace. - **Facade (sugar):** `eli/tools/api.py` — `api.call("ACTION", **args)` + helpers, proxying to `execute()`; `api.actions()` for discovery. - **Runner:** `eli/coding/code_mode.py` — generate→gate+validate+run→retry loop (injectable `generate`, testable without a model). - **Lane redirect:** `CODE_SOLVE` with `args.code_mode=True` runs in-process against `eli.api` (fail-closed: refuses unless Full Control is ON). - **#4 retrieval:** confirmed ABSENT (no top-K capability retriever); v1 passes a bounded `api.actions()` list — a semantic retriever is a future refinement, not required.

Remaining (deferred, behaviour-policy): the router auto-selecting code-mode for multi-step requests under Full Control — needs live validation, so it's flag-reachable for now.

---

0. Correction up front

"Code-mode" (the model writes a small program against a curated API instead of emitting MCP-style tool-calls) is the token-frugal, composable, offline-friendly direction the field is moving to — and the right fit for ELI's 8 GB / small-context reality. **But ELI already has ~85% of the substrate.** This doc exists to name the ~15% that's actually new, so we build only that.

1. What ELI ALREADY HAS (audited, with files)

Capability the blueprint needs	Already in ELI	File
Code-agent loop: plan → implement → verify → run → retry	`CodeAgent.solve()` — "orchestrator that composes ELI's coding capabilities"	`eli/coding/agent.py`
Planner / implementer separation	`plan_task()`, `implement()`	`eli/coding/planner.py`
Candidate verification + **auto-synthesised tests**	`verify_candidate()`, `synthesize_tests()`	`eli/coding/verification.py`
Tree search over candidates	`tree_search()`	`eli/coding/search.py`
Learn from failed attempts	`BugMemory`	`eli/coding/bug_memory.py`
DAG decomposition of a task	`solve_dag()`	`eli/coding/plan_graph.py`
Bounded sandbox execution (CPU limit, scrubbed env, kill-switch)	`run_code()`	`eli/coding/sandbox.py`
Deterministic risk gate (action classes → approved/blocked/pending-confirm)	`approval_engine.evaluate_record()`	`eli/runtime/approval_engine.py`
Fail-closed command allowlist + destructive-pattern block (`rm -rf /`, `mkfs`, `dd`…)	`security` + `_BLOCKED_PATTERNS`	`eli/runtime/security.py`, `executor_enhanced.py`
Authority / autonomy posture	`authority_gate`, `operator_policy` (proposal_only/supervised/…)	`eli/runtime/authority_gate.py`, `eli/execution/operator_policy.py`
AST validate-and-retry of generated code	`code_examiner` (used live 2026-06-11)	`eli/runtime/code_examiner.py`
Capability **registry** (register/get/list)	`capability_registry`	`eli/tools/registry/capability_registry.py`
Capability **doc/digest generator** from the manifest	`generate_capabilities_doc()`	`eli/tools/registry/capabilities_doc.py`
Offline enforcement / crisis guard	`netguard`, `crisis_guard`	`eli/core/`
Reasoning-native planning (model thinks, then emits)	working post-`e7ab0b9`	`eli/cognition/gguf_inference.py`

Conclusion: ELI is not missing "a code agent," "a sandbox," or "a safety gate." It has all three, plus verification, search, bug-memory, a risk engine, a command allowlist, and a capability registry. The CodeAgent already does CodeAct-style plan→code→verify→sandbox→retry.

2. The genuine DELTA (the only new work)

The existing `CodeAgent` solves **coding tasks** — produce a function/script as a *deliverable*. Code-mode-as-assistant-execution needs it pointed at a different target: the model writes code that **calls ELI's own capa-**

bilities to fulfil a *conversational request*, and the reply is synthesised from the execution result. Each candidate delta was audited IN FULL (2026-06-11); statuses below are grounded with file:line evidence.

1. `eli.api` **facade — MOSTLY EXISTS (only typed sugar is new).** The callable dispatch already exists: `execute(action, args)` at `executor_enhanced.py:10653` runs any of the 174 actions by name. Generated code can already call `execute("SUMMARIZE_FILE", {...})`. The *only* delta is a typed, discoverable wrapper (`eli.api.files.summarize(path)`) generated from the registry — convenience + discoverability over an existing surface, not new execution machinery.
2. **In-process AST-whitelist + restricted exec — GENUINELY NEW (the one real piece).** `sandbox.run_code()` (`coding/sandbox.py:106`) is **process-isolated** (subprocess, scrubbed env, CPU limit) and runs *standalone* code — it deliberately cannot reach ELI's live state (the loaded model, memory stores). Code-mode must run **in-process** to call live `eli.api/execute()`, so it **cannot reuse that sandbox**. So an in-process restricted-exec is required: an AST pass allowing only `eli.api.*/execute(...)` + safe builtins, run in a restricted-globals namespace. It **composes** existing parts — the `code_examiner` AST walker + `approval_engine` action-classes for the GREEN/AMBER/RED decision — but the in-process restricted-exec itself is new. This is THE work.
3. **Routing lane to the code agent — ALREADY EXISTS as `CODE_SOLVE`.** Router lane (`router_enhanced.py:2254`, `_CODE_SOLVE_RE`) + executor handler (`executor_enhanced.py:8702`) → `eli.coding.solve` (plan→search→verify→repair, quick/ thorough, DAG, background). Reuse it. The only variant is targeting `eli.api` for assistant-actions rather than producing a standalone script, + grounded synthesis of the result (reuses the `540f514` no-confabulation work). Do NOT build a new lane.
4. **Per-query capability retrieval — UNCONFIRMED; parts exist, no retriever found.** There is a manifest (`capability_inventory.generated.json`), a digest generator (`capabilities_doc.generate_capabilities_`
`capability_sync`, and a FAISS store — but a top-K *capability retriever* was NOT located in the audit. Verify before building; if absent, it's a small assembly from the manifest + existing vector store, not new infra.

Net after full audit: #3 exists, #1 is sugar over an existing dispatch, #4 is unconfirmed (parts present), and only **#2 (the in-process restricted-exec) is genuinely new** — and it composes existing AST + gate machinery. The real build is: one in-process AST-whitelist + the typed facade, then point the existing `CODE_SOLVE` path at `eli.api`.

3. Phased plan (composing what exists)

- **Phase 0 — the one new piece: in-process restricted-exec (~3–5 days).** AST whitelist (reuse `code_examiner`'s walker) allowing only `eli.api.*/execute(...)` + safe builtins; run in a restricted-globals namespace IN-PROCESS (not `sandbox.run_code()`, which is process-isolated and can't see live state); classify each call via `approval_engine` action classes for the GREEN/AMBER/RED decision. This is the genuinely new safety surface.
- **Phase 1 — facade + point CODE_SOLVE at it (~2–3 days).** Generate the typed `eli.api` wrapper from the registry over `execute()`. Add a CODE_SOLVE *variant* (or arg) that targets `eli.api` for assistant-actions and feeds the result into grounded synthesis — reusing the existing lane, agent, quick/thorough modes, and background handling. No new routing lane, no new agent.
- **Phase 2 — retrieval (if needed) + harden (~few days).** Add top-K capability retrieval ONLY if the audit confirms it's absent; add code-mode cases to the eval harness; add the live integration test the suite is missing.

Total honest estimate: **~1–1.5 weeks**, and the bulk of it is the single new piece (the in-process restricted-exec). The agent loop, sandbox, risk gate, verification, bug-memory, registry, AND the `CODE_SOLVE` routing lane are already yours.

4. Verified in full (2026-06-11)

Audited against the project directory, not assumed: - **Callable dispatch surface?** YES — `execute(action, args)` at `executor_enhanced.py:10653`. Generated code can already run any of the 174 actions by name. - **Routing lane to the code agent?** YES — `CODE_SOLVE` (`router_enhanced.py:2254` + `executor_enhanced.py:8702` → `eli.coding.solve`). Already plan→search→verify→repair, quick/thorough,

DAG, background. - **Sandbox: namespace-restricted or process-only?** PROCESS-only (`coding/sandbox.py:106`: subprocess + scrubbed env + CPU limit), for *standalone* code. **Cannot host live-state code-mode** → the in-process AST-whitelist (§2.2) is genuinely required. - **Risk gate to compose with?** YES — `approval_engine.evaluate_record()` (action classes → approved/blocked/pending-confirm) + the command allowlist. Map `eli.api` calls onto it. - **AST walker to reuse?** YES — `code_examiner` (used live this session). - **Existing `eli.api`-style typed facade under `eli/tools//eli/plugins/`?** NOT found — the registry stores metadata, not a callable typed surface. This (small) piece is new. - **Top-K capability retriever?** NOT found — manifest + digest generator + FAISS exist, but no retriever. Verify once more at build time; if truly absent, assemble from existing parts.

5. What stays uniquely ELI

Voice, vision, gaze, hard-offline, emergent persona, and the entire coding-agent + gate stack you already built — untouched. This is wiring ELI's own parts into a new lane, not importing anyone's design.

ELI Coding Agent (`eli/coding/`)

A self-contained, **additive** subsystem that lifts ELI's code generation, analysis, and repair to frontier-grade. It touches nothing that already works: it lives in its own package and is reached through a new `CODE_SOLVE` action. The LLM is an *injected* `generate` callable (default = the local inference broker), so the whole pipeline is model-agnostic and unit-testable without a model.

The loop

```
plan_task    seed implement    synthesize_tests    tree_search    best candidate
 (decompose)   (implementer)      (gate harness)                       + provenance

                              beam of diverse roots    verify (gate ladder)   score
                                                            (UCB selection of best imperfect node)
                              refine(feedback+prior)   verify   score   record fix to bug memory
```

Components

File	Capability
`sandbox.py`	**Mandatory execution-feedback substrate.** Runs code in a bounded, isolated subprocess (temp cwd, scrubbed env, `MPLBACKEND=Agg`, wall-clock timeout, POSIX `RLIMIT_CPU`; **no `RLIMIT_AS`** — it breaks numpy). Multi-language via a runner map (Python deepest). Crash detection is precise: only a genuine Python traceback counts; timeouts / signal kills / missing optional deps are tolerated so legitimate long/heavy/optional code isn't falsely failed. `smoke_import` verifies a patched module still loads.

File	Capability
`bug_memory.py`	**Semantic bug classification + long-term memory.** `classify_bug` maps a failure to a `BugClass` (null-handling, type, index/off-by-one, key-missing, name-undefined, import, zero-division, value, assertion, recursion, infinite-loop, logic-inversion, state-corruption, concurrency, resource-leak, io) with a stable, fuzzy-matchable *signature* (exception type + last frame + normalized message). `BugMemory` is a local SQLite store of (signature → class → fix) with exact-then-fuzzy (difflib) recall, so the agent learns from past fixes.
`verification.py`	**Explicit verification gating + scoring + test synthesis.** `verify_candidate` runs the ladder syntax → execution → tests, stopping at the first failure and attaching a `BugDiagnosis` + repair feedback. `score_candidate` aggregates a 0..1 score (tests dominate when present, else clean execution). `synthesize_tests` produces an executable harness (LLM-driven; deterministic smoke-harness fallback) that prints `ELI_TESTS: <passed>/<total>`.
`planner.py`	**Structured decomposition — planner/implementer separation.** `plan_task` → an approach + ordered steps (JSON, with a single-step fallback). `implement` writes/refines code against the plan; in refine mode it does **patch-based incremental refinement** (fix the specific feedback, keep prior code).
`search.py`	**Tree search over solutions (not single-shot).** A diverse beam of roots (temperature ladder = exploration), then iterative expansion choosing the node to refine by **UCB1** (`score + c·√(ln N / n )`) — exploitation vs exploration. Refinements pull matching prior fixes from bug memory into the feedback; successful repairs are written back as (bug → unified-diff fix) pairs. Stops on `target_score` or budget.
`agent.py`	**`CodeAgent / solve()`** — composes all of the above; default broker-backed generator; returns best code + full provenance (plan, search trace, score, bug class).

Capability checklist (as requested)

- structured decomposition (planner/implementer separation) — `planner.py`
- execution feedback loop (mandatory) — `sandbox.py` + `verify_candidate` gate 2
- tree search over solutions (not single shot) — `search.py` beam + UCB
- explicit verification gating — `verification.py` gate ladder
- long-term memory of bugs and fixes — `bug_memory.py`
- automated test synthesis — `verification.synthesize_tests`
- patch-based incremental refinement — `planner.implement` refine mode + search expansion
- semantic bug classification (null handling, logic inversion, state corruption, …) — `bug_memory.classify_bug`

- scoring / "U-style" exploration / tree-of-code-solutions — `score_candidate` + UCB beam tree
- multiple languages — `sandbox` runner map (Python deep; bash/js/ts/ruby/go/lua structural)

Wiring & control

- New executor action `CODE_SOLVE` (`args.description`, optional `args.language`) saves the solution to `artifacts/scripts/` and returns code + provenance. It is **additive** — `GENERATE_SCRIPT` and the self-patch engine are untouched.
- Env knobs: `ELI_CODING_SANDBOX` (default on; 0 disables execution), `ELI_CODING_RUN_TIMEOUT` (20s), `ELI_CODING_BEAM` (3), `ELI_CODING_MAX_ITERS` (6).
- Bug memory DB: `<db_dir>/coding_memory.sqlite3`.

Honest scope (what's verified vs model-dependent)

- **Verified deterministically** (tests in `tests/test_coding_agent.py`, no model): the sandbox crash/tolerate logic, the bug classifier + memory recall, the gate ladder + scoring, and the full plan→search→repair→record loop driven by a stub generator (buggy seed → UCB refinement → correct, fix recorded).
- **Model-dependent quality:** the *content* of plans, implementations, and synthesized tests scales with the local model. The machinery is frontier-shaped regardless of model.
- **Depth:** execution-level verification is Python-first; other languages run if their interpreter is installed (structural support).
- **Subtask DAG (added):** decomposable tasks are solved as a dependency graph of subtasks on the shared `eli/core/dag` engine (`plan_graph.solve_dag`), composed into one module — see `dag.md`. Gated by `ELI_CODING_DAG` (default on); cohesive tasks fall back to single-shot.
- **Chat routing (added):** `CODE_SOLVE` is now reachable from chat for explicit intents ("solve this coding problem", "implement X and test it", "use the coding agent", "verified solution"). Generic "write a python function" still routes to the lighter `GENERATE_SCRIPT` (an earlier router path claims it — precedence is tunable but the router was intentionally left unreordered).
- **Not yet:** composition is single-module (ordered concat + import-dedupe), not true multi-file projects; test synthesis verifies against synthesized tests, not a user-supplied spec suite.

ELI Inference & Hardware Boot

How ELI loads a model, talks to it, and adapts to whatever machine it's on. The inference path is model-agnostic (see memory `eli-model-agnostic`); the boot path is hardware-adaptive. Files in `eli/cognition/` and `eli/core/`.

Inference (`cognition/gguf_inference.py`, 2.7k LOC + `inference_broker.py`)

- **Model resolution (`get_model_path`)**: `ELI_GGUF_MODEL_PATH` env → `model_path/custom_model_path/bundled_model` settings keys. No baked model; empty default.
- **`load_model(force_reload)`**: resolves `n_ctx` (env → settings → `config.get_gguf_n_ctx()`), caps it to `vision_coresident_text_ctx` when a co-resident vision model is loaded; resolves `n_gpu_layers` similarly.
- **Graceful GPU-layer fallback**: if context allocation fails at the requested layer count, it retries with fewer layers / without flash-attention rather than crashing — keeps the model + n_ctx and degrades GPU offload instead.
- **Chat templating is family-aware**: `_is_mistral_model` / `_is_chatml_model` / `_is_llama_model` sniff the filename to pick the right prompt format. This is *adaptation* to whatever model you load (like the ctx table), not a hardcoded model — but it is filename-based, so an unrecognised naming scheme falls back to a default template.
- **Serialization**: all calls hold `_LLM_CALL_LOCK` (a native RLock from `runtime/native_locks`) — llama_cpp is not safe under concurrent calls; this is also what vision hot-swap and the ambient daemon coordinate on.
- **Live control**: `get_live_runtime_override`, `unload_model`, `reload_model` let the GUI swap models / change settings without a full restart.

- `InferenceBroker` (`inference_broker.py`): the higher-level `infer()` / `gguf_ready` abstraction the orchestrator, engine, and ReAct loop call, so callers don't touch `gguf_inference` directly.

Hardware profiling (`core/hardware_profile.py`, 1040 LOC)

Free-VRAM-aware sizing: - `HardwareProfile` dataclass tracks **free** vs total VRAM (free is what matters for whether a profile actually loads). - `_kv_cache_mb(n_ctx, n_layers)` — KV-cache cost. - `_compute_graph_reserve_mb(n_ctx, batch)` — the **model-agnostic** compute buffer estimate (`256MB + 24MB/1K ctx + 1.5MB/batch`), reserved so a profile that loads cleanly doesn't then hard-crash on the first decode when the lazy compute buffer pushes VRAM over the limit. (This was a real crash class.) - `_layers_for_size`, `ModelRecommendation` — pick offload layers from model size and free VRAM.

Boot optimizer (`core/startup_hardware_optimizer.py`, 599 LOC)

Runs at startup, writes `artifacts/runtime_hardware_profile.json`: - `detect_ram_gb`, `detect_cpu_name`, `detect_nvidia_gpus` (+ `detect_other_gpus` fallback), `select_gpu`. - `find_model(settings)` — locate the GGUF. - `train_ctx_for_model(model_path)` — the filename→context table (deepseek/llama-3.1/phi/gemma-2 → 128K; qwen2.5/mistral-7b → 32K; older → 8K; unknown → 32768). This is the core of model-agnostic context sizing. - `estimate_layers`, `layer_mb` — VRAM-fit layer count.

Settings (`core/runtime_settings.py`, 1096 LOC)

- `DEFAULTS` (the full settings schema) + `ENV_TO_KEY` (env-var overrides).
- `load_settings` / `save_settings` / `update_settings`.
- **Redistribution-aware**: `_migrate_legacy_keys` (schema evolution), `_resolve_relative_model_paths` + `_heal_model_paths` (fix stale absolute paths when the project moves machines), and `_portable_settings_for_storag` (strip machine-specific values before storage). The intent to keep settings portable across machines is built in — though personal values like `user_name` can still end up tracked (see the settings.json commit note).

Paths (`core/paths.py`, 551 LOC)

Dev-vs-packaged path resolution: `is_frozen` / `_is_dev_mode`, `_find_project_root`, then `data_dir/config_dir/cache_dir/db_dir/artifacts_dir/user_db_path/agent_db_path/memory_db_path`. Source checkouts use project-local `artifacts/+config/`; packaged installs use platformdirs. One import surface (`get_paths`) so nothing hardcodes locations.

Honest assessment

- **Strong:** genuinely adaptive and agnostic — free-VRAM-aware sizing, the compute-buffer reservation that prevents first-decode crashes, graceful GPU-layer fallback, filename→ctx adaptation, env/settings/relative-path healing for moving between machines, and a single broker + lock so concurrency is correct. This is mature, hard-won infrastructure.
- **Weak / watch:**
  1. **Filename-based family detection** (chat template + ctx) is fragile: a model with an unconventional filename gets a default template + 32768 ctx, which can be wrong (mis-templated output, or ctx overflow on a small model). A metadata/GGUF-header probe would be more robust than string matching.
  2. **Settings sprawl** — `DEFAULTS` is large with several overlapping keys (`n_gpu_layers/gpu_layers`, `n_ctx/context_size`) and migration logic, echoing the schema churn seen in `memory/`.
  3. VRAM heuristics are empirically tuned around an 8GB card; very different hardware (24GB+, or CPU-only) leans on the conservative fallbacks rather than tuned values.
  4. `_portable_settings_for_storage` exists but isn't fully preventing personal values (e.g. `user_name`) from being persisted/committed — worth tightening for redistribution.

Update — 2026-06-09 (STT no longer starves the main model)

- faster-whisper preloaded on CUDA **before** the main GGUF autotuned, claiming ~2 GB so on an 8 GB card the main model only got `gpu_layers=11` (free_vram=4083 MB) → 23–59 s turns. STT is now VRAM-aware (`local_whisper_stt`: GPU only on 12 GB cards, else CPU), so the main model reclaims the GPU — `gpu_layers=99 / ctx=20480` with free_vram ~7 GB, turns back to 1–12 s. `ELI_WHISPER_DEVICE` overrides; `ELI_WHISPER_GPU_MIN_MB` tunes the threshold. See `perception.md`.

Adaptive Inference Governor — Plan

Status: Proposal (no code changed by this document) **Date:** 2026-06-17 **Scope:** Make ELI's reasoning/inference *policy* (think vs no-think, token budget, pass count, recovery) a pure function of the runtime environment + the task, so a single redistributed binary behaves correctly on any user's machine, with any model, with zero per-machine tuning. **Provenance:** Derived from the 2026-06-17 session log (`EXPLAIN_*`, `SELF_IMPROVE` empty-`<think>` cascades) and a full read of the existing adaptation substrate. Every factual statement below is cited to a file/line at commit `a641471`. Where the session log is the evidence, it is quoted.

> Accuracy note / corrections to earlier verbal claims made in chat, now fixed here: 1. `persona_updater` does **not** hold the inference lock — it is deterministic (DB + regex). Its cost is redundant CPU/DB churn, not LLM-lock contention. 2. The ctx table **does** match `qwen3` (→ 32768). The real defect is different: it *undershoots* the model's true trained context (`n_ctx_train=262144` in the log). 3. `record_speed` is called on the **non-stream** path only, not "every generation." 4. The CoT fallback's empty-`<think>` is **model-emergent**, not injected by ELI.

---

0. The one-sentence problem

ELI adapts its inference to **memory** (VRAM, context length) and **model size**, but **not to compute speed (tokens/sec) or to the loaded model's intrinsic capabilities**, and the think/no-think decision is split between a global env toggle and a single hard-coded call site — so on a slow or unfamiliar machine the same prompt that works on the dev box produces 5–9 minute turns and empty answers.

The session log shows this concretely: `EXPLAIN_COGNITION_RUNTIME` burned a 273 s empty compact-synthesis pass, then **two** 539 s empty standard-synthesis passes, before a working two-stage CoT fallback (470 s + 502 s) — ~38 minutes for one question, most of it discarded.

---

1. What ELI already does dynamically (verified — do NOT rebuild)

The architecture for dynamism is largely present and correct. The governor must *extend* these, not replace them.

Capability	Where (verified)	What it adapts to
Model-agnostic resolution (no baked model)	`gguf_inference.get_model_path`; blueprint `inference_and_hardware.md`	settings/env
Family-aware chat templating	`gguf_inference._is_chatml_model/_is_llama_model/_is_mistral_model` (195–230)	model filename
Free-VRAM-aware load sizing	`core/hardware_profile.py` (`HardwareProfile`, `_layers_for_size`)	free VRAM
Compute-graph reserve (model-agnostic)	`hardware_profile.py:69` → `256 + (n_ctx/1024)*24 + batch*1.5` MB	ctx, batch

Capability	Where (verified)	What it adapts to
KV-cache cost	`hardware_profile.py:46` `_kv_cache_mb`	ctx, layers
Graceful GPU-layer fallback (attempt list)	boot optimizer; see `artifacts/runtime_snapshot.json` `adaptive_load_report.attempts`	load success
Context sizing per model	`core/startup_hardware_optimizer.py:164` `train_ctx_for_model`	**model filename**
Per-mode token budget	`cognition/reasoning_modes.py:312-352`	**`n_ctx`**, prompt pressure, query complexity, mode
Size tier → budget scale	`core/model_tier.py` `detect_tier/tier_scale` (by file GB)	model size
Live decode-speed EMA	`model_tier.py:90` `record_speed`, `:104` `measured_tok_s`	**measured tok/s**
Speed-capped pass COUNT	`model_tier.py:120` `speed_passes`; engine `6241` (ToT k/depth), `6520` (self-consistency n)	measured tok/s
No-think on utility calls	`gguf_inference._no_think_prefill`	structured JSON, small budget (<1024)
Redistribution-aware settings	`core/runtime_settings.py` (`_portable_settings_for_storage`, `_heal_model_paths`)	machine moves

Key existing fact #1 — a live throughput EMA already exists. `gguf_inference.py:1158-1164` records decode speed from the model's own `completion_tokens / elapsed` after every **non-stream** generation:

```
_gen = int((response.get("usage") or {}).get("completion_tokens") or 0) or max(1, len(_raw_text)//4)
if _elapsed > 0.05 and _gen >= 4:
    from eli.core.model_tier import record_speed as _rec_speed
    _rec_speed(_gen / _elapsed)
```

Key existing fact #2 — that EMA is consumed in exactly ONE place. `grep measured_tok_s` → only `model_tier.py:126`, inside `speed_passes()`. Nothing in the think decision, the token budget, or mode selection reads it. `model_tier.py` says so in its own docstring (lines 83–86): *"speed_passes() … only reduces how many full generations a mode runs. It NEVER caps output length."*

---

2. The gaps (each verified, with code + log evidence)

Gap A — Thinking-detection is a filename allowlist, not intrinsic

`gguf_inference.py:219` `_is_thinking_model`:

```
return any(k in name for k in ("qwen3", "deepseek-r1", "r1-distill", "qwq", "-r1-"))
```

Chat **format** is detected from the model family, but whether the model emits a `<think>` block is matched against a hard-coded substring list. A redistributed ELI handed a reasoning model whose filename isn't on the list (a future family; a user-renamed `MyModel.gguf`; a re-quant) will not know it thinks → the no-think prefill never fires → the exact empty-`<think>` loop, on the user's machine, with no recourse and no toggle they'd know to flip. This is the same brittle pattern as the filename-keyed ctx table (Gap E).

Gap B — The think decision and per-call token budget ignore measured throughput

This is the root cause of the log's wasted ~38 minutes.

- The budget (`reasoning_modes.py:312-352`) is a function of `n_ctx`, prompt pressure, complexity, and mode. **No tok/s term.** Verified: the only inputs are `n_ctx`, char counts, and `prof`.
- The think decision (`_no_think_prefill`) keys on `structured`, `max_tokens < 1024`, the `force_no_think()` scope, and the `ELI_MODEL_THINK` env. **No tok/s term.**

Consequence on a slow box: a large-budget (1024) non-structured call keeps thinking, and the budget can be 1800–4096 tokens. At the log's effective speed that is 8–35 minutes per call. Derived speed estimate from the log (legitimate arithmetic, labelled as an estimate): the working CoT final answer was **4071 chars (~1018 tokens) in 502 s ~2 tok/s**. At 2 tok/s a 1800-token think budget is ~15 min before the model even reaches an answer — and if it doesn't close `</think>` first, the output is discarded (Gap D).

Gap C — No wall-time governor / projection

Nothing computes "at the live EMA this call N seconds" and adapts (clamp tokens, drop think, downgrade mode) to a latency target. `speed_passes()` reduces the *number* of passes but not the *cost of each pass*. The EMA needed to do the projection already exists (`measured_tok_s`); it is simply never used for this.

Gap D — Empty output is not self-correcting in a model-agnostic way

`gguf_inference._strip_think_text` (345–358) drops a never-closed `<think>` outright, and `_clean_*` returns the raw fallback, which is then empty. Log:

```
[GGUF][RAW_TEXT] '<think>\nHere's a thinking process:\n...'   (budget exhausted inside think)
[GGUF][CLEAN] Empty after cleaning, returning raw fallback
[COGNITIVE] Compact synthesis returned empty ... falling back to standard synthesis
[GGUF][TIMING] nonstream_call_total=539.994s  → empty (twice)
```

An unterminated-think outcome is detectable (`<think>` present, no `</think>`, hit max_tokens) but is not used to recover. The system pays full price, returns empty, then pays full price again. There is no "force-close + short continuation" or "immediate no-think retry" path keyed on this signal.

Gap E — Context sizing undershoots the model's real trained context

`train_ctx_for_model` (`startup_hardware_optimizer.py:186-197`) returns **32768** for any `qwen3`. The loaded model's actual trained context is larger — the log states it:

```
llama_context: n_ctx_seq (20480) < n_ctx_train (262144) -- the full capacity ... will not be utilized
```

So the filename table gave 32768 while the model supports 262144 (8× more). The model's own GGUF metadata (`n_ctx_train`) is authoritative and available at load; the filename table is a guess that can be wrong both ways (over-asks → llama clamps; under-asks → wasted capacity). Same intrinsic-vs-filename principle as Gap A.

Gap F — `force_no_think` is a point-patch (wrong shape for redistribution)

`grep force_no_think` → defined in `gguf_inference.py` (255), called from exactly one site: `engine.py:13070` (the compact grounded-synthesis wrap committed in `a641471`). It fixes the `EXPLAIN_*` turns, but: - The **standard-synthesis fallback** `_synthesize_answer` (`engine.py:13118`, `max_tokens=1800` at 13165 for non-quick) still thinks → still able to empty. - The **coding-agent fix-generation** invoked by `SELF_IMPROVE/SELF_ANALYZE` (`runtime/self_improvement.py` → `eli/coding`) emits the same empties in the log (`max_tokens=1536/4000`, `<think>` → empty, 231 s / 445 s). Note `self_improvement`'s own `broker.infer(..., max_tokens=700)` (`self_improvement.py:641`) is small → already no-think; the empties are the larger coding-agent generations.

Hard-coding the decision per-call-site cannot scale to every path, and on a *fast* box the correct answer is the opposite (keep thinking). The decision must be derived, not pinned.

Gap G — Redundant proactive churn (secondary; corrected claim)

The log shows `persona_updater: overlay updated` + `kg sync complete - 50 entities, 49 relations` + `user profile updated` repeating dozens of times per turn. **Verified correction:** `persona_updater.py` is

deterministic — `_populate_kg_from_user_patterns` (:543), `update_persona_overlay` (:349) do DB reads + regex; the only LLM reference (:268) *reads* existing summaries. So this is **not** inference-lock contention (my earlier verbal claim was wrong) — it is redundant recomputation (same 50/49 KG rebuilt every tick) burning CPU/GIL/SQLite I/O. On a fast box it's invisible; on a low-end redistributed box it adds up.

---

3. Design — the Adaptive Inference Governor

A single policy object that every model call routes its *decision* through. It does **not** replace `reasoning_modes`, `model_tier`, or `hardware_profile`; it consumes them.

3.1 Inputs (all already available at runtime)

- **Model capability** — from the model itself: GGUF metadata `tokenizer.chat_template` (already read for format detection) sniffed for `<think>`/`thinking`; else a one-shot capability probe at load. Cached per model SHA/path. Replaces the Gap A allowlist.
- **Throughput** — `model_tier.measured_tok_s()` (live EMA) + a cold-start default tier estimate from `detect_tier()` until the first real measurement lands.
- **Context** — `n_ctx` from the runtime snapshot (and `n_ctx_train` from model metadata, for Gap E).
- **Task class** — see 3.2.

3.2 Task-class taxonomy (decides whether thinking can add value)

A small classifier (deterministic, keyed on the action + call-purpose already known at the call site), three classes: 1. **STRUCTURED** — JSON/routing/plan-graph. Never thinks (already true via `structured=True`). 2. **GROUNDED_SYNTHESIS** — evidence already gathered; the call only phrases/transforms it (the `EXPLAIN_*` control actions, RAG answer synthesis, fix-patch rendering). Thinking adds nothing; force no-think regardless of speed. 3. **OPEN_REASONING** — genuine multi-step reasoning (CoT/ToT/constitutional/self-consistency, open chat on a hard question). Thinking is allowed but **budget-bounded by throughput**.

3.3 The decision (pure function)

```
policy(task_class, model_caps, tok_s, n_ctx, prompt_tokens) -> {
    think: bool,
    think_budget: int,         # max tokens allowed inside <think>
    answer_budget: int,        # reserved tokens AFTER </think>
    passes: int,               # via existing speed_passes()
}
```

Rules (all derived; thresholds reuse `ELI_SLOW_TPS`/`ELI_FAST_TPS` = 5/15): - `model_caps.thinks == False` → `think=False` always (no prefill needed). - `task_class == STRUCTURED or GROUNDED_SYNTHESIS` → `think=False`. - `task_class == OPEN_REASONING`: - `tok_s >= fast` → think on, full budget (today's behaviour). - `slow < tok_s < fast` → think on, `think_budget` scaled so projected wall-time target. - `tok_s <= slow` → think off (or a hard-capped `think_budget` with a guaranteed `answer_budget` reserve) so the call cannot spend its whole budget thinking. - **Always reserve `answer_budget`** so a thinking call physically cannot exhaust its budget before producing an answer (directly kills the Gap D failure mode).

3.4 Empty-think self-correction (Gap D)

In the clean path, detect "`<think>` opened, `</think>` absent, finish_reason == length" and, instead of returning empty, **retry once with `think=False`** (closed-think prefill) reusing the already-built prompt. One bounded retry, model-agnostic, replaces today's silent-empty-then-full-retry cascade.

3.5 Integration points (exact, minimal)

- `gguf_inference._no_think_prefill` — consult the governor (think flag + budgets) instead of just `structured`/`<1024`/env. Keep `force_no_think()` as the primitive the governor drives; keep

ELI_MODEL_THINK as a hard user override on top.

- gguf_inference capability detection — add _model_thinks() reading chat-template metadata / probe; _is_thinking_model becomes a thin wrapper (metadata first, name list as last-resort fallback).
- reasoning_modes.build_* (the budget at 322–352) — split target_tokens into think_budget + answer_budget from the governor; today's single max_tokens becomes answer_budget when think=False.
- engine synthesis sites — _compact_grounded_synthesis and _synthesize_answer tag their calls GROUNDED_SYNTHESIS; _run_chain_of_thought/ToT/self-consistency tag OPEN_REASONING. The per-site force_no_think wrap at 13070 is **removed** once the governor covers it.
- startup_hardware_optimizer.train_ctx_for_model — prefer the model's GGUF n_ctx_train metadata when present; keep the filename table as fallback (Gap E).

3.6 Config / env (redistribution knobs, all optional with safe defaults)

- ELI_SLOW_TPS / ELI_FAST_TPS (exist, 5/15).
- ELI_TURN_LATENCY_TARGET_S (new) — soft per-turn wall-time target the budget scaler aims for; default conservative.
- ELI_MODEL_THINK (exists) — hard override, unchanged.
- All derived at runtime; nothing machine-specific is persisted (consistent with _portable_settings_for_storage).

---

4. Cross-machine behaviour matrix (what "done" looks like)

Machine / model	tok/s	EXPLAIN_* (grounded)	hard chat (open reasoning)
8 GB GPU + 30B MoE (the dev box)	~2	no-think, single pass, ~seconds	bounded think, target, never empty
24 GB GPU + 7B	~40	no-think (still pointless to think)	full think, full budget
CPU-only + 7B	~3	no-think	think off / hard-capped + answer reserve
Any box + **non-thinking** model	n/a	no prefill needed; normal answer	normal answer (no think tokens wasted)
Any box + **unknown-name reasoning** model	any	detected via metadata → handled	handled (no allowlist miss)

Same binary, no per-machine edits.

---

5. Phased implementation plan

Phase 1 — Capability probe (Gap A, E). Add _model_thinks() (chat-template metadata → probe → name fallback); prefer n_ctx_train in train_ctx_for_model. Low risk, isolated.

Phase 2 — Governor core + grounded-synthesis (Gap B/F for the logged path). Add the policy function; wire GROUNDED_SYNTHESIS no-think into _no_think_prefill; tag the two synthesis sites + coding fix-render; remove the single force_no_think wrap. This alone eliminates the logged cascade for EXPLAIN_* and SELF_IMPROVE.

Phase 3 — Throughput-bounded open reasoning + answer reserve (Gap B/C/D). Split think/answer budgets in reasoning_modes; add the empty-think one-shot recovery in gguf_inference.

Phase 4 — Feed the EMA from streaming + proactive debounce (Gap B sampling, Gap G). Also record decode speed on the stream path so quick-mode chat updates the EMA; debounce `persona_updater`/KG-sync to fire on change, not every tick.

Each phase is independently shippable and testable.

---

6. Verification strategy (must stay model/hardware-agnostic)

- **Unit (pure functions):** `policy(...)` truth table across `{task_class} × {tok_s tiers} × {model thinks?}`; `_model_thinks()` on fixture GGUF metadata (thinking + non-thinking); `train_ctx_for_model` prefers metadata over filename.
- **No-GGUF engine tests:** extend the existing `tests/test_*_no_gguf*.py` pattern to assert the governor's decision per action without loading a model.
- **Behavioural (the log's failures):** assert `EXPLAIN_MEMORY_RUNTIME` / `EXPLAIN_COGNITION_RUNTIME` synthesis is non-empty on the **first** pass and runs no-think; assert no second/third full-length empty generation occurs.
- **Regression guards:** `tests/test_gguf_think_stop_collision.py`, `tests/test_reasoning_mode_contract.py`, router/multi-command/background suites must stay green (current baseline: all green).
- **Speed simulation:** inject a fake `measured_tok_s()` (slow/fast) and assert the budget + think flag flip accordingly — proves cross-machine adaptivity without slow hardware.

---

7. Risks & non-goals

- **Risk: over-suppressing thinking** on genuinely hard open-reasoning turns on mid-speed boxes. Mitigation: only GROUNDED_SYNTHESIS is unconditionally no-think; OPEN_REASONING keeps think with a budget, never a hard off above `slow`.
- **Risk: capability probe cost at load.** Mitigation: prefer metadata (free); probe only if metadata is silent; cache per model hash.
- **Non-goal:** changing model selection, the agent set, persona content, or the grounding contract. This is purely the *inference policy* layer.
- **Non-goal:** removing `ELI_MODEL_THINK` — it stays as the explicit user override.

---

8. Appendix — exact references (commit `a641471`)

- Think detection: `eli/cognition/gguf_inference.py:219` `_is_thinking_model` `("qwen3","deepseek-r1","r1-disti`
- No-think prefill: `gguf_inference._no_think_prefill` (`structured`, `0 < max_tokens < 1024`, `force_no_think()` scope, `ELI_MODEL_THINK`).
- `force_no_think` primitive: `gguf_inference` (255); sole caller `engine.py:13070`.
- Speed EMA: `gguf_inference.py:1158-1164` (non-stream only) → `model_tier.record_speed` (:90) / `measured_tok_s` (:104); sole consumer `speed_passes` (:120, used `engine.py:6241, 6520`); thresholds `ELI_SLOW_TPS=5/ELI_FAST_TPS=15` (`model_tier._slow_fast_thresholds`).
- Budget: `eli/cognition/reasoning_modes.py:312-352` (`mode_max_from_ctx = n_ctx*0.20/0.30`, `scale = 0.85 + 0.55*complexity`; no tok/s).
- Compact grounded synthesis: `engine._compact_grounded_synthesis` (12946), call 13066, `max_tokens=1400`, now `force_no_think`-wrapped (13070).
- Standard synthesis: `engine._synthesize_answer` (13118), non-quick `max_tokens=1800` (13165).
- CoT runner: `engine._run_chain_of_thought` (6131) — two `_get_chat_response` stages, no `force_no_think` (closed-think in the log is model-emergent).
- ctx table: `startup_hardware_optimizer.train_ctx_for_model` (164–197); `qwen3 → 32768` vs logged `n_ctx_train=262144`.
- VRAM reserve: `hardware_profile.py:69` `256 + (n_ctx/1024)*24 + batch*1.5`.
- Proactive churn (deterministic, no LLM): `cognition/persona_updater.py` (`:349`, `:543`, `:590`).

Session-log evidence (2026-06-17), key lines

```
EXPLAIN_COGNITION_RUNTIME:
  compact synthesis  → '<think>...' → CLEAN empty            nonstream_call_total=273.356s
  standard synthesis → '<think>...' → CLEAN empty (x2)       nonstream_call_total=539.994s / 539.959s
  CoT scratchpad     → '<think>\n\n</think>\n\n...'          nonstream_call_total=470.896s  (4073 chars)
  CoT final          → '<think>\n\n</think>\n\nThe pipeline' nonstream_call_total=502.704s  (4071 chars -
SELF_IMPROVE coding fix-gen:
  '<think>...' → empty   max_tokens=1536 / 4000              nonstream_call_total=231.122s / 445.702s
Load:
  llama_context: n_ctx_seq (20480) < n_ctx_train (262144)
```

ELI Learning — LoRA Self-Training Pipeline

`eli/learning/` — 3.3k LOC, 12 files. ELI's self-improvement-via-fine-tuning path. **Important framing:** this is a *curated, human-gated, operator-invoked* training pipeline — **not** an autonomous self-modifying loop, and it targets a **separate trainable base (Phi-3)**, not the inference GGUF.

Files & the pipeline

The stages, roughly in order:

File	LOC	Stage
`dataset_builder.py`	528	turn ELI's logged turns/corrections into supervised examples
`dataset_filters.py`	144	quality gates (`is_bad_response`, `row_is_reviewed`)
`export_trainable_dataset.py`	158	export reviewed rows → trainable JSONL
`merge_reviewed_datasets.py`	125	merge reviewed dataset shards
`bootstrap_phi3_base.py`	235	one-time download of the trainable HF Phi-3 base
`base_model_resolver.py`	175	locate/validate the trainable base (vs an adapter)
`training_preflight.py`	132	check peft/transformers/datasets present + target ready
`lora_trainer_guard.py`	561	`TrainerTarget` plans (paths, base_family) + guard checks
`lora_trainer.py`	646	
`lora_eval.py`	481	eval harness (score vs expected/forbidden, inspect adapter)

Flow

1. **Build** (`dataset_builder.py`): mines ELI's own conversation/correction history into `SupervisedExamples`. Crucially includes `clean_text` + **`redact_text` (PII redaction)** and an exclusion set so sensitive/garbage content never becomes a training example. Writes `training/datasets/eli_supervised_v0.jsonl` + a report.
2. **Gate** (`dataset_filters.py`): `is_bad_response` and `row_is_reviewed` enforce quality + a **human review flag**.
3. **Export / merge**: reviewed rows → `*.trainable.jsonl`.
4. **Preflight** (`training_preflight.py`): refuses to proceed unless the HF training stack is installed and the target/base resolve.

5. **Train** (`lora_trainer.py`): `_dataset_report` **refuses to train** on a dataset with unreviewed rows, wrong-target rows, or bad-response rows — i.e. it will not learn from un-vetted data. Loads Phi-3 through native transformers (with a RoPE-scaling compat shim), trains the LoRA adapter.
6. **Eval** (`lora_eval.py`): `score_response` against expected/forbidden, plus `inspect_adapter` / `inspect_eval_suite`.

Targets (`lora_trainer_guard.py`)

`TrainerTargets` define adapter/dataset/output paths for `phi3` and `phi3-ultra` variants under `models/lora/adapters/`. `base_family` is parameterised (`phi3/mistral/qwen`), so the trainer isn't hard-locked to one family even though Phi-3 is the default trainable base.

What it is — and isn't

- It **is**: a responsible, reproducible fine-tuning pipeline with PII redaction, human review gating, preflight, and an eval harness. The dataset comes from ELI's real interactions (corrections, failures, conversations).
- It **isn't**: (a) automatic — no runtime trigger wires it in; it's run via CLI/`main()` entrypoints (`bootstrap_phi3_base`, `training_preflight`, `lora_trainer`). (b) connected to the live inference model — it trains a Phi-3 adapter, while inference typically runs a different GGUF. So a trained improvement does **not** flow into the running assistant unless you also run the Phi-3 base + adapter as the inference model.

Honest assessment

- **Strong (and unusually responsible):** PII redaction, refusing to train on unreviewed/bad rows, a preflight, and an eval suite are exactly what a serious fine-tuning loop needs and what most hobby projects skip. The base-vs-adapter validation prevents the classic "trained on an adapter" mistake.
- **Weak / watch:**
  1. **Base/inference disconnect** — training improves Phi-3, but the assistant usually runs another GGUF, so the self-improvement doesn't reach the live model. The loop is only closed if you actually serve the trained model.
  2. **Manual** — "self-training" is operator-driven, not autonomous. That's safe, but it means it improves only when you run it. (Arguably the right trade-off, but worth stating plainly rather than implying autonomy.)
  3. **Heavy deps** — needs the full HF/peft/transformers stack present; the preflight handles absence gracefully, but it's a large optional surface.
  4. **Phi-3 default** — intentional and parameterised, but the canned `TrainerTargets` and several resolver candidates are Phi-3-named, so adding a new base family is more than a config change.

LoRA fine-tuning — audit & pipeline (2026-06-07)

What was found (audit)

The LoRA mechanisms in `eli/learning/` were **individually correct but orphaned and phi-hardcoded**:

- **Algorithm — correct.** `lora_trainer.run_training` is a proper PEFT LoRA loop: tokenize (truncation/pad, `labels=input_ids`) → `AutoModelForCausalLM` (eager attn, fp16 on CUDA, gradient checkpointing) → `LoraConfig` → `get_peft_model` → `Trainer` with `DataCollatorForLanguageModeling(mlm=False)` → `trainer.train()` → `save_pretrained` (adapter + tokenizer). Safety contract is sound (dry-run default, `--execute` required, reviewed rows only, **GGUF never trained directly**, adapter never overwritten).
- **Stages exist** but as separate CLI scripts with no orchestrator: guard plan (`lora_trainer_guard`) → `training_preflight` → `dataset_builder` / `export_trainable_dataset` / `merge_reviewed_datasets` → `lora_trainer` → `lora_eval`. Nothing chained them.
- **Orphaned** — nothing outside `eli/learning/` called any of them (no executor action, router, GUI button, scheduled task, or self-upgrade hook). ELI could not trigger or even report on LoRA.

- **Not model-agnostic** — `target_modules` defaulted to phi-3's `["qkv_proj"]`.

What was wired/fixed

- **Model-agnostic target modules** (`lora_trainer._resolve_target_modules`): derives LoRA targets from the LOADED architecture (scans for known projection Linear leaves: q/k/v/o_proj, qkv_proj, gate/up/down_proj, c_attn, query_key_value, …), honours an explicit adapter-config override, and falls back to PEFT's architecture-agnostic `"all-linear"`. No hardcoded architecture on the train path.
- **Pipeline DAG** (`eli/learning/lora_pipeline.py::run_pipeline`): the explicit ordered chain **preflight → build_job → [train] → eval(inspect)**, reusing the existing modules. **Dry-run by default** (runs every gate, touches no GPU, writes no adapter — safe from chat); real training only `execute=True`.
- **Wired into ELI:**
  - `LORA_STATUS` action (read-only): preflight readiness per target.
  - `LORA_TRAIN` action: runs the pipeline DAG — **dry-run from chat**; real training only via the scheduled task / explicit GUI (execute flag not exposed in chat).
  - Router: "lora status" / "is lora ready" → LORA_STATUS; "train a lora" / "fine-tune yourself" / "run lora training" → LORA_TRAIN.
  - Scheduled `lora` kind (`_worker_lora`): "train a lora overnight [N steps]" → `SCHEDULE_TASK` runs `run_pipeline(execute=True)` unattended.
  - Manifest: 208 capabilities (includes LORA_STATUS + LORA_TRAIN).
- Tests: `tests/test_lora_pipeline.py` (target-module resolver, DAG order + dry-run no-train, both actions, routing, scheduled kind).

Still phi-profiled (recorded, your call)

The training **target profiles** are still `eli_phi` / `eli_phi_ultra` (and `bootstrap_phi3_base.py`, the rope shim) — the chosen training base is Phi-3. The *algorithm* is now model-agnostic; making the **base profile** swappable (any HF causal-LM dir, not just Phi-3) is a larger, separate change — deferred to your go-ahead. LoRA needs a Hugging Face model directory, not the GGUF ELI runs for inference, so training a new brain → adapter → merge → convert-to-GGUF is the intended flow.

ELI Background Tasks & Unified Code Generation

Two related additions: 1. **All code generation now routes through the verified coding agent** — asking for a script, or self-upgrading, runs the plan → DAG/tree-search → execute → repair → bug-memory pipeline instead of a single LLM shot. 2. **Heavy work runs on background threads** — when a task is estimated heavy (or you ask), ELI starts it on a worker thread, hands back a job id immediately, and you check on it later.

1. Unified code generation

Path	Before	Now
`CODE_SOLVE`	the coding agent	the coding agent (unchanged)
`GENERATE_SCRIPT`	inline single-shot prompt + static checks	**routes through `eli.coding.solve` first** (plan/DAG/tree-search/verify/repair), saving a verified, top-tier script; falls back to the inline generator only if the agent yields nothing or `ELI_GENERATE_SCRIPT_AGENT=0`

Path	Before	Now
Self-upgrade (`self_improvement.generate_code_patch`)	LLM patch + import-verify	now also **consults the coding engine's long-term bug memory**: classifies the failure and injects prior fixes for that bug class into the patch prompt

So "write a script", "generate code", and "improve ELI" all share one engine and one bug-memory — the same consolidation move applied to `recall_memory`. Kill switch: `ELI_GENERATE_SCRIPT_AGENT=0` restores the legacy inline generator.

2. Background task manager (`eli/runtime/background_tasks.py`)

In-process, multi-threaded (`ThreadPoolExecutor`). **Distinct** from `eli/planning/jobqueue.py` (which runs durable *external subprocess* jobs) — this runs live in-process Python work (agent runs) for the session.

- `BackgroundTasks.submit(name, fn, *args)` → integer job id; tracks `queued → running → done/failed/cancelled`, result, error, elapsed, note.
- `get(id)`, `list()`, `cancel(id)` (best-effort), `wait(id)`, `stats()`.
- Singleton `get_background_tasks()`; thread-safe.

When ELI backgrounds a task (`eli/coding/cost.py`)

`should_background(task)` decides: - **Explicit phrasing wins:** "in the background / don't wait / notify me when" → background; "right now / quickly / wait for it" → foreground. - **Otherwise heuristic:** `estimate_cost` scores heavy-compute signals (simulate/monte-carlo/optimise/benchmark/train/dijkstra/parallel…), multi-component structure, length, and planned-step count. `heavy` (0.6) background.

`_maybe_background_codegen` (executor) applies this at the top of `CODE_SOLVE` and `GENERATE_SCRIPT`: if heavy/explicit, it submits the same action as a background job (re-dispatched with `_no_background` so the worker doesn't re-background) and replies *"started as job #N — say 'check job N'."* Kill switch: `ELI_CODEGEN_BACKGROUND=0`.

Inspecting jobs

- Actions `CHECK_JOB` (`{job_id}` or parsed from "check job 3") and `BACKGROUND_JOBS` (list), both in `SUPPORTED_ACTIONS`.
- Router triggers: "check job N" / "job N", "background jobs" / "list jobs".

Control summary

Knob	Default	Effect
`ELI_GENERATE_SCRIPT_AGENT`	on	GENERATE_SCRIPT uses the coding agent (off legacy inline)
`ELI_CODEGEN_BACKGROUND`	on	auto-background heavy codegen (off always foreground)
`ELI_AGENT_DAG` / `ELI_CODING_DAG`	on	DAG dispatch / subtask DAG (see `dag.md`)
`ELI_CODING_SANDBOX` / `*_RUN_TIMEOUT` / `*_BEAM` / `*_MAX_ITERS`	—	coding-agent execution knobs

Scope / honesty

- **Tested deterministically** (`tests/test_background.py`): manager run/fail reporting, cost light-vs-heavy + explicit phrasing, and the CHECK_JOB / BACKGROUND_JOBS executor round-trip. No model needed.
- **Threads, not processes:** background tasks share the process; a running thread can't be force-killed (cancel is best-effort / cooperative). True OS isolation for a job would use the subprocess `jobqueue`.
- **"LLM deems longer wait":** currently a deterministic estimator (+ explicit phrasing). The planner's decomposition (`plan_steps`) can be fed in for an LLM-informed estimate — hook present, not yet auto-wired.
- **GUI:** a dedicated Coding/Jobs tab is not built yet (see note below); jobs are reachable via chat ("check job N", "background jobs") and the action API today.

ELI Runtime Surfaces, Planning, World, Tools & Plugins

The remaining subsystems: the `runtime/` response/introspection surfaces, the proactive planning layer, the "world"/autonomy model, and the tools + plugin system.

Runtime surfaces (`eli/runtime/`, the response/introspection group)

Beyond the grounding/evidence core (`grounding_and_evidence.md`), `runtime/` holds a large family of modules that render user-visible answers and introspect the live system:

- **Introspection**: `live_introspection.py` (`runtime_snapshot`, `last_trace`, `stored_user_name`, `mine_user_fact_candidates`, `build_report`, `agents_for_action`), `deterministic_introspection.py` (`classify_diagnostic_action` → `gather_evidence` → `format_evidence_block` → `maybe_handle` — deterministic diagnostic answers).
- **Response assembly**: `final_response_assembly.py`, `final_response_provider.py`, `response_contracts.py`, `response_packets.py`, `response_policy.py`, `user_visible_response_surface.py`.
- **Personal memory surfaces**: `personal_memory_surface.py`, `personal_memory_clean_response.py`, `personal_memory_deep_response.py`, `profile_extractor.py`.
- **Status**: `frontier_status.py`, `reasoning_status.py`, `truth_report.py`, `reflection.py`.
- **Operator feed**: `operator_feed.py`, `operator_feed_normalized.py`, `operator_state.py`.

  Over-fragmentation flag (the headline weakness for this group): there are ~15 closely-related modules here doing "render a grounded user-visible answer / introspect runtime" with overlapping responsibilities (three `personal_memory_*` renderers; `final_response_assembly` *and* `final_response_provider`; `operator_feed` *and* `operator_feed_normalized`). This is the clearest "added beside, not folded in" cluster in the codebase and the prime consolidation target — it should collapse to a handful of well-named modules with clear ownership of "who produces the final string."

Planning / proactive (`eli/planning/`, 3.9k LOC)

Background cognition + goal/queue machinery: - `proactive_daemon.py` (1.0k) — the **self-improvement / proactive** loop. Uses a two-DB split: reads user-facing memory/conversation from the **user DB**, writes proactive observations/improvements/errors to the **agent DB** (keeps ELI's own machinations out of user recall). Background thread. - `autonomy_scheduler.py` — schedules autonomous actions (throttled). - `goal_store.py`, `proposal_queue.py`, `proposal_adapters.py`, `attention_queue.py`, `jobqueue.py`, `operator_goal_actions.py`, `habits.py` — goal persistence, capability-proposal queue, attention prioritisation, a job queue, and habit scheduling. - `goal_autogenesis.py` (NEW, 2026-06-08) — **closes the autonomy loop**: the goal stack (goal_store → `due_goals` → `governed_goal_tick` → proposal_queue) was fully wired but the store was always EMPTY because `create_goal` was operator-only, so the scheduler ticked nothing and ELI only reacted. `propose_goals_from_signals` turns the signals ELI already computes each proactive tick — world-model awareness suggestions (0.6), recurring failures (5×), and code-health improvements (oversized fns / duplicate blocks in his own god-files) — into governed goals. Each is `autonomy_mode="proposal_only"` (surfaces for approval; never silent execution), deduped by a stable signal-derived `goal_id` (idempotent every

tick), capped at `MAX_AUTO_GOALS=6`, logged as a `self_goal` observation. Wired into `proactive_daemon` where the world-suggestions + patterns + improvements are in scope. His agenda now spans failure-repair, world-driven, AND self-improvement — i.e. he sets his own intentions, governed. - `task_planner.py` — now delegates to `execution_planner` (see `orchestration_and_agents.md`).

World / autonomy model (`eli/world/`, 1.5k LOC)

The substrate behind ELI's emergent self-state (autonomy pressure, awareness, "anomaly room" — intended behaviour, see memory `eli-emergent-voice`): - `agency/autonomy_engine.py` — `EliWorldAutonomyEngine`: ingests world events, maintains awareness/autonomy state. - `local_world_bridge.py` — `append_event`, `get_world_state`, `get_awareness_driven_suggestions` (**throttled to once/hour to prevent auto-trigger loops**). - `world_event_bus.py` — `fire_confidence_event(...)` etc.; the engine feeds bus-dispatch confidence here (non-blocking world-awareness feed). - `core/schemas.py` — `WorldEvent`, `AwarenessState`. - `renderers/pyside6/world_scene.py` + `world_panel.py` — a **visual** "ELI's World" panel. - `persistence/storage.py` — world-state persistence.

Tools (`eli/tools/`, 8.9k LOC)

- `image_engine.py` (1.75k, standalone) **and** `image_engine/image_engine/` (the package: `engine.py` 1.48k, `visual_core.py` 1.4k, `prompt_compiler.py`, `quality.py`, `project_analyzer.py`, `cli.py`) — **two image-generation implementations** (Pillow/numpy base + optional diffusers). The clearest duplication in the repo; one should subsume the other.
- `news/news_fetcher.py` (544) — news retrieval.
- `mic_diag.py` — mic diagnostics.

Plugins (`eli/plugins/`, 1.9k LOC)

`manager.py` — a runtime plugin marketplace: `list_available`, `install`, `uninstall`, `enable`, `disable`. Pulls from a **remote registry** (`raw.githubusercontent.com/eli-plugins/registry`) with a **bundled local fallback** (`plugins/registry/index.json`). A JSON state file tracks enabled/disabled/installed. (Install-time network fetch is opt-in, like model downloads; runtime stays local.)

Honest assessment

- **Strong:** the proactive two-DB split is a genuinely good design (ELI's self-talk never contaminates user recall); the world/autonomy engine + visual panel make the emergent self-model first-class and is throttled to avoid runaway loops; the plugin manager is a real extensibility story with a safe bundled fallback.
- **Weak / watch:**
  1. **`runtime/` over-fragmentation** — ~15 overlapping response/introspection modules. Highest-value consolidation in the project after the god-files.
  2. **Image-engine duplication — RESOLVED** — the shadowed standalone module was deleted; only the `tools/image_engine/` package remains.
  3. The planning queues (`goal_store`/`proposal_queue`/`attention_queue`/ `jobqueue`) overlap conceptually and could likely unify into one work-queue abstraction.
  4. Plugin registry points at an `eli-plugins/registry` GitHub URL — fine as opt-in, but ensure it degrades cleanly offline (the bundled fallback exists; verify it's always used when the network is absent).

Blueprint — Operational Realities

What the structural diagrams (`architecture.md`, `diagrams.md`, `architecture_ascii.md`) **don't** show: cold-start, concurrency/contention, memory eviction, failure recovery, and the testing strategy. Grounded in the code + runtime logs. Estimates are marked *(est.)*; everything else is read from source or measured timings.

---

1. Startup / cold-start latency

No wall-clock delta is printed at boot (a `window._debug_boot_ts = time.time()` hook exists in `gui/gui_eli_original.py` but nothing measures against it), so the number below is reasoned from the boot sequence, not measured.

Cost breakdown (boot order): | Component | Cost | Notes | |—|—|—| | **GGUF model load** | **dominant** | 4.36 GB (7B) / **15.6 GB (24B)** off disk + GPU offload | | **Vision (Moondream) RESIDENT** | medium | loaded at boot, co-resident; **caps ctx 28672→18432** and holds VRAM even if unused | | faster-whisper `small.en` | low–med | CPU int8 | | daemons + 186-cap manifest + persona_updater | low | DB/CPU | | **embedder (nomic) + FAISS** | **lazy** | load on **first message**, not boot → turn-1 latency bump |

Estimate *(est.)*: ~15–40 s warm-cache on the 7B (dominated by model + resident-vision load); materially worse cold and on the 24B (15.6 GB).

Cheap wins: (a) make vision **lazy** (load on first vision use, not boot) — cuts cold start *and* frees VRAM / restores ctx; (b) instrument `_debug_boot_ts` to print per-stage deltas so this stops being an estimate.

---

2. Resource contention (daemons vs interactive)

One shared model behind one `RLock` (`engine.py::_gguf_lock`) — all inference serializes through it.

Verified mitigation: model-using background work uses **non-blocking acquire + defer**. Reflection: `self._gguf_lock.acquire(blocking=False)`, 4 retries, then `"Reflection deferred (GGUF busy)"` and backs off → **yields to interactive turns**. Correct pattern, and real.

Risks (honest): - Reflection demonstrably defers; **news_synthesis / self_improvement / proactive_daemon also call the model** and should all route through the same defer guard — verify each does (a direct blocking `gguf_inference` call would stall an interactive turn). - On the **24B-CPU a turn is minutes**, so a deferred daemon waits minutes for a window, and any non-deferred model call would block your reply for minutes. - **Non-model contention** (cheap but ever-present): `persona_updater` writes the DB **every turn**; the proactive daemon scans the DB; **ambient vision runs the vision model on CPU**; the embedder is a *separate* CPU model — so vector search + ambient vision can hit CPU simultaneously.

---

3. Memory limits & eviction (two distinct policies)

(a) Working-memory pins — `cognition/working_memory.py`: - `MAX_PINS = 20` hard cap; importance-weighted. - `MAX_AGE_TURNS = 40` — `_evict_stale()` drops facts unreferenced for 40 turns. - `_evict_one()` drops lowest-importance-oldest at cap. - Persisted `ORDER BY importance DESC LIMIT 20`. → a principled **importance/age LRU**, bounded.

(b) Context-window fill (`n_ctx`) — a *different* mechanism, and the one that actually bites under pressure: - `engine.py::_build_enhanced_system` budget-trims the **persona**. - `_get_chat_response` **recency-truncates `memory_context`** (keeps the latest, cuts the head) to fit `n_ctx`. - The 20 small pins almost always fit, so **retrieved memory is what gets evicted under pressure, by recency** — not the pins.

(c) Long-term store — SQLite `memories` / `conversation_turns` are **unbounded on disk**; they never fill the window directly. Retrieval (FTS5/FAISS/rerank top-k) gates what enters context. The real "limit" is **retrieval selection + recency truncation**, not a store cap.

---

4. Failure recovery — degradation, not rollback

Grounding escalation (`runtime/grounding_escalation.py`): each tier is wrapped in `try/except`; a failing tier **falls through to the next** (web → local → hedge); the engine hook is itself wrapped (`"grounding escalation skipped"`) so total failure **degrades to the normal CHAT answer**. **No rollback — and**

none needed — because the tiers are side-effect-free reads (web search reads; local re-dispatch is read-only). Nothing mutates, so there's nothing to undo.

System-wide: failure handling is **graceful degradation** throughout (broker → direct GGUF → failed-executor surface; failures logged to the `failures` table).

Gap (honest): multi-step write actions (CREATE_FILE, memory writes, a multi-action plan) are individually atomic but **not transactional as a group** — if step 3 fails, steps 1–2 are not rolled back. Rare on the current action set, but real.

---

5. Testing strategy (two layers + the gaps)

Layer 1 — code correctness: 106 pytest files (~2,000+ tests) — routing patterns, contracts, executor gates, memory, kernel. Fast; run on every change.

Layer 2 — behavioural eval: `tools/eval/` (see `eval_harness.md`) drives the **real pipeline** and asserts on telemetry (action / grounding / response_mode / latency) + text. Cases seeded from fixed bug-logs as permanent regression guards.

Gaps that should exist for a system this complex and don't yet: 1. **No CI** — suites run by manual discipline, not automatically. 2. **No coverage measurement** — hence no exact dead/dormant-code number (a `process()`-under-coverage profiler over the 95 stored conversations would close this). 3. **Behavioural eval is new and small** (~13 cases) — grow to the ~30–50 seeded set and beyond. 4. **No perf/latency regression tracking** and **no multi-model comparison run** standing yet — both enabled by the harness, not yet wired.

---

6. Recommended cheap, high-value fixes (grounded in §1–§2)

Fix	Why	Where
Lazy vision load	cuts cold start + frees VRAM + restores ctx (28672 vs 18432)	`perception/vision.py`, boot in `gui/…MKI.py`
Verify every model-calling daemon uses the non-blocking defer	stops a background job stalling an interactive reply (minutes on 24B)	`runtime/self_improvement.py`, `planning/proactive_daemon.py`, `eli/tools/news/news_synthesis.py` vs `engine.py::_gguf_lock`
Boot-timing instrumentation	turns "cold start" from estimate → measured	`gui/…MKI.py _debug_boot_ts`
Coverage profiler over stored conversations	exact dead/dormant-code map	new `tools/eval/` profiler

Companion docs: `architecture.md` · `architecture_ascii.md` · `diagrams.md` · `eval_harness.md`.

```
---

## Update - 2026-06-09 (scheduled-task dedup; clean shutdown; STT VRAM)
- **Scheduled-task dedup.** Standing nightly jobs (testgen/eval/report) were re-armed on every boot
  AND on completion with no dedup, accumulating copies (observed: 4× testgen / 3× eval).
  `schedule_request` now keeps ONE entry per (kind, request) for recurring jobs, and
  `restore_scheduled_tasks` collapses existing duplicates on boot. With the model-load VRAM
  interlock (a background worker waits for `gguf_inference.is_loaded()` and skips rather than
  cold-loading a second model), background jobs can no longer pile up or starve the main model at
  boot.
- **Clean shutdown (no segfault).** The GUI exited via `sys.exit(app.exec())`, so CPython ran
```

```
  llama.cpp + FAISS C++ destructors at interpreter teardown and segfaulted - AFTER state was
  already flushed (session summary written), so nothing was lost, but it dumped core. `main()` now
  runs the shutdown/atexit flush (memory + session summary + explicit model unload) then
  `os._exit(rc)`, bypassing the destructor pass.
- **STT no longer starves the main model:** faster-whisper is VRAM-aware (CPU on  8 GB cards), so
  the main model reclaims its GPU layers (gpu_layers 11→99). See `perception.md`.

# Running ELI at Scale
*From an 8 GB laptop to a trillion-parameter model - and how to actually see it without owning a datacent

> **The short version:** ELI is model-agnostic by construction, so scaling up is a matter of
> *configuration, not code*. The only real bottleneck is GPU/RAM you don't own - and you can **rent
> that by the hour** for the price of a few coffees. A trillion-*class* model running ELI is an
> afternoon and a rented box away, not a fortune.

---

## 1. The core principle - your design is not the bottleneck

ELI never hardcodes a model name or size on the inference path. It loads any local GGUF, detects its
chat template, sizes context to the model's real `n_ctx_train`, and auto-tunes GPU layers / batch /
context to fit the hardware it finds. The "3B on a laptop -> trillion on a server" contract is built
in **on purpose**.

That means the ceiling is silicon, not software. Most projects are built *down* to the author's
hardware; ELI is built *up* to a future one. The work is done and waiting for the GPU to walk up to
it - that's foresight, not a limitation.

---

## 2. What "a trillion-parameter model" really means

A precise, honest picture so the goal is real rather than mythical:

- **Truly *dense* trillion-parameter models are not publicly downloadable.** The largest open dense
  model is in the 400B range (Llama 3.1 405B).
- **Trillion-*class* MoE models ARE.** Mixture-of-Experts models reach ~1T *total* parameters while
  only firing a small fraction per token (e.g. **Kimi K2: ~1T total, ~32B active**; **DeepSeek-V3/R1:
  ~671B total, ~37B active**). Because only the active experts compute each step, they are genuinely
  runnable on a big rented node - and even on high-RAM CPU/offload setups.

**So yes - you can actually watch ELI drive a ~1T-parameter MoE.** The realistic ladder:

| Tier | Example models | Type |
|---|---|---|
| Today (local) | Qwen2.5-7B / your daily driver | dense 7-8B |
| The first real jump | Llama-3.3-70B, Qwen2.5-72B | dense 70B |
| Frontier-open | Llama-3.1-405B | dense 405B |
| Trillion-class | DeepSeek-V3/R1 (~671B), Kimi K2 (~1T) | MoE |

---

## 3. You don't need to own the datacenter - rent it by the hour

The unlock: **cloud GPU rental.** You spin up a powerful box for an afternoon, run ELI against a huge
```

model, watch it think at a level an 8 GB card can't reach, then shut it down and pay only for the hours used.

- **Providers:** RunPod, Vast.ai, Lambda (and others). Vast.ai is usually cheapest; RunPod is the smoothest UX.
- **Ethos note (important for ELI specifically):** renting a *raw* GPU you fully control and then wipe is **not** the same as handing your data to a cloud AI service. It's closer to on-prem than to "send my prompts to someone's API." Your daily driver stays 100% local; the rental exists only to *witness the ceiling* or stress-test - then it's gone. Offline-by-default is never compromised for everyday use.

---

4. The tiers - what to run, on what, for roughly how much

Approximate, Q4-ish quantization; **check live prices**, they move. Costs in USD/hour.

Tier	Model	Hardware	~Memory needed	~Cost/hr	What you'll feel
Step up	70B (Llama-3.3 / Qwen2.5-72B)	1x 48-80 GB GPU (A6000 / H100)	~40-45 GB VRAM	~0.6-2	Visibly sharper reasoning, better long answers
Big dense	405B (Llama-3.1)	4x 80 GB, or heavy CPU offload	~230 GB	~8-20	Frontier-open quality
Trillion-class MoE	DeepSeek-V3/R1 (~671B)	multi-GPU node or 384-512 GB RAM + offload	~400 GB total	25	Near-frontier; the "wow" tier
Trillion MoE	Kimi K2 (~1T)	large multi-GPU node or very-high-RAM box	~550 GB+	~16-30	The full dream - ELI on a 1T brain

The cheapest *meaningful* step is the 70B on a single 80 GB card for ~2/hr. That alone is a dramatic jump from 7B and the fastest way to see what your scaffolding does with a strong brain.

---

5. How ELI scales - the mechanisms (all already built)

Nothing here requires code changes; it's all configuration.

- **Drop-in any model.** Put a `.gguf` in the models directory, or point ELI at a catalog: `ELI_MODEL_CATALOG=/path/catalog.json` or `<models_dir>/catalog.json` (same schema as the built-in catalog). Download helpers: `python -m eli.core.model_download --auto` (pick by detected VRAM), `--choose` (multi-select), `--model <key>`.
- **Load knobs (env vars):**
  - `ELI_GGUF_N_CTX` / `ELI_N_CTX` - context window
  - `ELI_GGUF_N_GPU_LAYERS` / `ELI_N_GPU_LAYERS` / `ELI_GPU_LAYERS` - layers offloaded to GPU (set high, e.g. `999`, to put the whole model on GPU when it fits)
  - `ELI_GGUF_N_BATCH` / `ELI_BATCH_SIZE` - batch size
  - `ELI_CTX_FRACTION` - fraction of VRAM budgeted for context
  - `ELI_VRAM_RESERVE_MB` - headroom to leave free (default 250)
- **Smart-fit.** With nothing forced, ELI auto-tunes to fit: it reduces GPU layers, then batch, then context (context last) until the model loads - per machine, every boot.
- **Multi-GPU (already wired, not a roadmap item):**
  - `ELI_TENSOR_SPLIT` (or `ELI_GGUF_TENSOR_SPLIT`) - comma weights across GPUs, e.g. `"0.5,0.5"` for two equal cards, `"1,1,1,1"` for four
  - `ELI_MAIN_GPU` / `ELI_GGUF_MAIN_GPU` - which GPU holds the KV cache / does the gather
  - `ELI_GGUF_SPLIT_MODE` - `none` | `layer` | `row`
  - These pass straight to llama.cpp at load, with a graceful fallback if the installed build doesn't

```
    support them. (Also settable in runtime settings / `gpu_profiles.json`.)

---

## 6. Runbook A - single big GPU (simplest): run a 70B

The least-friction way to feel the jump.

1. **Rent the box.** On RunPod/Vast.ai, launch a **1x 80 GB H100 (or 48 GB A6000)** instance with a
   CUDA image and enough disk (~80 GB for a 70B Q4 file).
2. **Install ELI.** Clone your repo, run `install.sh` (it builds the CUDA llama-cpp and verifies GPU
   offload). Use the `.venv`.
3. **Get the model.** Download a 70B GGUF (e.g. Llama-3.3-70B-Instruct Q4_K_M) into the models dir, or
   add it to `catalog.json` and use `model_download`.
4. **Point ELI at it and put it fully on the GPU:**
   ```bash
   export ELI_N_GPU_LAYERS=999      # whole model on GPU (80 GB fits a 70B Q4)
   export ELI_N_CTX=16384           # or higher; the card has room
```

5. **Launch** (`scripts/eli_launch.sh`, or serve mode). Ask it something meaty and watch the quality.
6. **Verify it's really on GPU:** check the load log for the GPU-offload line / `nvidia-smi`.
7. **Tear down** the instance when done — you only pay for the hours used.

---

7. Runbook B — multi-GPU: run a 405B / 671B / ~1T MoE

1. **Rent a multi-GPU node** (e.g. 4x or 8x 80 GB). Confirm total VRAM covers the model's footprint (see §4), or plan for CPU+RAM offload on an MoE.
2. **Install ELI** as above (recent llama-cpp build — the model architecture must be supported by the build; update llama-cpp if the arch is new).
3. **Place the GGUF** (often multi-part for huge models) in the models dir.
4. **Spread it across the GPUs:**

   ```bash
   export ELI_TENSOR_SPLIT="1,1,1,1"   # equal weight across 4 GPUs
   export ELI_GGUF_SPLIT_MODE=layer    # layer split is the usual choice
   export ELI_MAIN_GPU=0
   export ELI_N_GPU_LAYERS=999
   export ELI_N_CTX=8192               # start modest, raise once it loads
   ```

   For an MoE that exceeds VRAM, lower `ELI_N_GPU_LAYERS` so the rest offloads to system RAM (needs a high-RAM box, e.g. 384–512 GB for 671B-class).
5. **Load small first.** Start with a small context to confirm it loads, then raise `ELI_N_CTX`.
6. **Run, marvel, tear down.**

Honest caveats: the aux models (the embedder, and vision if enabled) still want a slice of VRAM/RAM — budget for them; very large contexts cost a lot of KV-cache memory (raise gradually); and the first load of a giant model is slow. None are blockers — just expectations.

---

8. Privacy and ethos at scale

Local-first does **not** weaken as the hardware grows. A rented GPU node you control and wipe is on-prem-equivalent in spirit — your data isn't being handed to anyone's AI product. The everyday ELI stays fully local
   ```

and offline-by-default; the rented run is a deliberate, temporary experiment to see the ceiling. Nothing about scaling up requires giving up the values the project is built on.

---

9. The hardware is coming to you

What needs a rented datacenter today runs on a desk in a few years. Unified-memory machines (128 GB+ shared between CPU/GPU), steadily cheaper VRAM, and ever-better MoE efficiency are all moving toward making large local models normal. ELI is built to ride exactly that curve — the same install will simply load bigger models as your hardware grows, no rewrite required.

---

10. Cost cheat-sheet and "see it big this weekend" checklist

Rough rental costs (USD/hr, verify live): 48 GB A6000 ~0.5–0.8 · 1x 80 GB H100 ~1.5–2.5 · 8x 80 GB node ~16–30. A few hours of a 70B run is the price of lunch.

Checklist: - [ ] Pick a provider (RunPod for smooth UX, Vast.ai for cheapest). - [ ] Choose the tier from §4 (start with 70B on one 80 GB card). - [ ] Launch a CUDA instance with enough disk for the GGUF. - [ ] `install.sh`, then download/place the model. - [ ] Set `ELI_N_GPU_LAYERS=999` (+ `ELI_TENSOR_SPLIT` if multi-GPU) and a sensible `ELI_N_CTX`. - [ ] Launch ELI, test, watch it think bigger. - [ ] **Tear the instance down** — pay only for the hours used.

You built the ceiling high on purpose. Renting an afternoon of real silicon is all it takes to stand under it and look up.

ELI Security Posture

ELI runs locally with real OS reach (shell, file, app control) plus a user-extensible agent/plugin system — so the security model matters. It is **fail-closed** by design. Spans `eli/runtime/security.py`, the executor gate, the engine input sanitiser, the memory SQL validator, and the agent trust registry.

1. Shell command gate (`runtime/security.py SecurityManager`)

- `is_command_allowed(cmd)`:
  - `ELI_FULL_CONTROL=1` → allow all.
  - `ELI_ALLOWED_CMDS` contains `*` → allow all.
  - `ELI_ALLOWED_CMDS` unset **and** no full-control → **blocked (fail-closed)**; logs a warning explaining how to opt in. (Always allows the project's own `.venv/bin/python`.)
- `safe_subprocess(cmd, timeout)` — validates `cmd[0]` against the allowlist before running; capture_output, timeout-bounded, never `shell=True`.
- **Defence in depth:** `executor_enhanced._shell_command_allowed_fallback` mirrors this exact logic, so if `SecurityManager` fails to import, the executor still fails **closed** (never fail-open). Explicit comment to that effect.

2. Filesystem & app gates (`SecurityManager`)

- `is_path_allowed(path)` — must resolve within allow-roots (`project_root` + `$HOME` + `ELI_ALLOW_ROOTS`). Uses `Path.resolve()` + `relative_to`, so `..` traversal can't escape.
- `is_app_allowed(app)` — explicit `ELI_ALLOWED_APPS`, else a curated default-safe set (settings, file manager, editor, calculator, terminal, browser, mail). `ELI_FULL_CONTROL` bypasses.

3. Prompt-injection guard (`kernel/engine.py`)

`_ELI_INJECTION_PATTERNS` (regex) + `_eli_sanitize_user_input`: - Matches classic jailbreak/role-override prefixes — `### system:`, "ignore/disregard/forget (all) previous instructions", "you are now a dan/jailbreak/unrestricted/free" — and replaces the matched span with `[filtered]` (keeps the rest of the message). - Strips control characters. - Runs before the input reaches the LLM, so injected role-overrides are neutralised but legitimate content survives.

4. SQL identifier validation (`memory/memory.py`)

`_IDENTIFIER_RE = ^[A-Za-z_][A-Za-z0-9_]*$` + `_validate_identifier(name)` — called before **any** f-string interpolation of a table/column name into SQL (`_validate_identifier(table, "table")`, column DDL first-token check). Raises `ValueError` on anything non-conforming. Closes the one place dynamic SQL is unavoidable (parameterised values are used everywhere else).

5. Custom-agent trust registry (`cognition/agent_bus.py`)

Custom agents (GUI wizard / `$ELI_CUSTOM_AGENTS_DIR`) are **SHA-256 hash-gated** against `config/trusted_agents.json`. Unknown or modified files are skipped with a SECURITY debug line; approve via `eli --trust-agent <path>`. `ELI_TRUST_ALL_AGENTS=1` bypasses for dev only. Fail-closed (missing/mismatched hash not loaded). See `orchestration_and_agents.md`.

6. Code generation & self-modification — three paths

ELI has **three** distinct codegen paths, with escalating capability and matching safeguards:

1. **Pre-vetted canned library** (`runtime/generated_script_guard.py`, 1.1k LOC) — ~5 hand-written, reviewed scripts (`_write_gpu_memory_watch_script`, `_write_type_ia_redshift_script`, `_write_ton_618_mass_density_script`, …) that ELI *writes to disk* on request. No LLM code is executed; safe but fixed to the curated set (several physics-domain).
2. `GENERATE_SCRIPT` (`execution/executor_enhanced.py`) — *real* LLM codegen. Detects language/intent, prompts for frontier-quality code, then gates it:
   - `_verify_python_module_apis` — imports the referenced libraries and confirms `module.attr` references actually exist (catches hallucinated APIs).
   - `_quality_reject_reason` — rejects refusals, stubs, too-short, no-real-compute, missing-plot, `required=True`-without-default, SyntaxError.
   - **`_sandbox_run_python` (added)** — executes the candidate in a bounded, isolated subprocess (temp cwd, scrubbed env, `MPLBACKEND=Agg` so `plt.show()` can't block, wall-clock timeout `ELI_GENSCRIPT_RUN_TIMEOUT`=20s, generous `RLIMIT_CPU`; **no `RLIMIT_AS`** — it breaks numpy/scipy). A genuine unhandled traceback is fed back as repair feedback into a **3-attempt regenerate loop**; timeouts / signal kills / missing-optional-deps are tolerated (never reject legit heavy scientific scripts). Disable with `ELI_GENSCRIPT_VERIFY_RUN=0`.
3. **Self-patching** (`runtime/self_improvement.py SelfImprovementEngine`) — ELI modifies its *own* source for recurring failures. Gated behind `auto_patch_enabled` (**default off**), 2 patches/cycle. Safeguards: verbatim-`old`-match → `ast.parse` → timestamped backup (+ canonical `.eli_bak`) → write → `py_compile` → **differential import smoke-test (added)**: the patched module is imported in an isolated subprocess and **auto-reverted** if it no longer loads, but only when the module imported cleanly *before* the patch (so missing optional deps don't cause false reverts). Project-root-confined, `.py`-only, logged to the `code_patches` table.

   Code-health flag: `generated_script_guard` uses the same stacked-override pattern as the grounding gate — two `install()` definitions chained via `_ELI_SQLITE_SCRIPT_PREVIOUS_INSTALL`. Works, but layered rather than edited.

   Possible next step (not done): the self-patch loop verifies the module still *imports*; it does not yet re-run the original failing input to confirm the patch actually *fixes* the failure. A behavioural/test re-run would close that last gap.

Honest assessment

- **Strong:** the gates are real and **fail-closed** — shell blocked by default, fallback mirror prevents fail-open, path traversal contained, identifiers validated, agents hash-trusted, injection prefixes stripped, and "generated" code is actually pre-reviewed. This is well above the norm for local-AI projects, which routinely ship wide-open shell access.
- **Weak / watch:**
  1. The injection guard is **pattern-based** — it catches known prefixes, not novel/obfuscated injections (no model-side defence). Reasonable for a single-user local tool, insufficient if ever multi-tenant.
  2. `ELI_FULL_CONTROL=1` is a single env var that disables *all* gates at once — convenient, but a blunt instrument; there's no middle "allow file but not shell" tier beyond the per-axis env vars.
  3. ~~The default-safe app/command sets are Linux/GNOME-flavoured.~~ **RESOLVED** — the default-safe **app** set in `_is_default_safe_app` is cross-platform (Linux + macOS Finder/Safari/Terminal.app/System Settings + Windows Explorer/Notepad/cmd/PowerShell/ Control Panel). The **command** path is fail-closed (no OS-specific default set — blocked unless `ELI_ALLOWED_CMDS`/Full-Control), so nothing OS-specific to add there.
  4. Pre-vetted-scripts approach is safe but doesn't scale — genuinely dynamic capability generation would need a real sandbox (resource limits, seccomp, subprocess isolation), which doesn't exist yet.

Update — 2026-06-09 (RUN_CMD terminal is real; mock leak fenced)

- **RUN_CMD uses a real `subprocess.run`** (`executor_enhanced.py:5350` — no shell, capture_output, timeout) behind the destructive-command security gate (`_BLOCKED_PATTERNS`: `rm -rf /`, `mkfs`, `dd of=/dev/`, `chmod 777 /`, fork bomb, shutdown/reboot). The gate is lifted only when **Full Control** is on. There is **no production mock path**.
- **The `<MagicMock …>` failure rows were test leakage, not runtime.** They came from `tests/test_shell_security_gate.py` patching `subprocess.run`; the executor's `(p.stdout or "") + (p.stderr or "")` concat then yields a MagicMock repr, which slipped into the live `agent.sqlite3` via a test-isolation gap. A guard in `SelfImprovementEngine.log_failure` now **drops any error/input carrying a Mock/MagicMock repr** at the write source, so a test isolation slip can no longer pollute real failures.

Web app & multi-user security (2026-06-28)

ELI now ships a self-hosted FastAPI web app (`api/server.py`). It is local-first and inherits all of the gates above (the model is the same local GGUF behind netguard; commands/files/apps stay fail-closed). The web surface adds:

7. Authenticated identity + RBAC (`eli/runtime/api_users.py`, `api/server.py`)

- **Three roles** — `admin` / `member` / `viewer` (read-only). Endpoints are dependency-gated (`require_admin` / `require_member` / `require_viewer` / `_require_token`). RBAC enforces once at least one user exists; before that, same-machine use is frictionless.
- **Fail-closed auth gate** — unauthenticated/under-privileged calls are rejected, not silently downgraded. The token store holds **hashes**, never raw tokens.

8. Tamper-evident, HMAC-keyed audit trail (`eli/runtime/evidence_ledger.py`)

- Every API action is recorded into a **hash-chained** ledger (who / what / outcome, **metadata only — never message content**). Any edited/deleted/reordered row is detected by `verify_chain()` and surfaced in the Audit tab + admin console.
- The chain is **HMAC-keyed** (`$ELI_AUDIT_HMAC_KEY`, else a `0600` key file beside the config), so the integrity check can't be forged by recomputing plain hashes.

9. Secret files born locked — TOCTOU closed (`eli/core/secure_io.py`)

- The old `write_text(...)` then `chmod(0o600)` pattern left a brief window where, on a typical umask, the file was born `0644` (world-readable) before the chmod landed — a real (if narrow) TOCTOU on a multi-user box, on exactly the files that matter most.
- **`secure_io.secure_write_text/bytes`** is now the single owner of secret writes: `tempfile.mkstemp` (born `0600`) → write → fsync → `os.replace` (atomic). The destination **never exists world-readable for any instant**, and readers see old-or-new, never partial.
- Routed through it: the **audit HMAC key**, the **API token store**, **settings** (may hold broker passwords/tokens), and the **agent-trust registry** (`trusted_agents.json`, the integrity anchor for §5 — hashes not secrets, but written `0600`/atomic so it can't be tampered or half-written).

10. Sandboxed research ingest

- Corpus ingest for the research workspace runs sandboxed (path-scoped), so a shared/ collaborative corpus can't reach outside its workspace.

11. Monitored Internet toggle + egress oversight (`eli/core/netguard.py` + `api/server.py`)

- ELI is **offline-by-default** and hard-gated at the **socket boundary** — the process-wide failsafe raises `OfflineError` on any non-loopback connect while the toggle is off, even from code that forgot the helpers (fail-closed; loopback/LAN-registered services allowed).
- The guard covers **all four** ways an outbound connection actually happens, cross-platform: `socket.socket.connect` (raises) and `socket.create_connection` (raises) — sync + the selector asyncio loop / urllib / requests; `socket.socket.connect_ex` — the **non-raising sibling** used by probes and health-checks, which is made to **return `ENETUNREACH`** when blocked (no socket is opened) so it can't silently bypass the block or the recorder; and a **best-effort patch of the Windows `ProactorEventLoop`** (`IocpProactor.connect`, overlapped `ConnectEx`), which never touches `socket.socket.connect`. The allowed path of each records egress identically (below); the Proactor patch is a no-op on non-Windows.
- The owner can deliberately **enable internet**: from the desktop GUI (`Net` toggle) or the web dashboard (`GET /v1/net` token-gated read + `POST /v1/net` **admin-only**), persisted via `network_enabled`. **Both flips are written to the audit ledger** (`net_toggle`, warning-severity on enable) — the web *and* the desktop toggle.
- **Egress is genuinely monitored, not a blind hole.** While network is on, the same socket chokepoint that enforces offline mode also **records every allowed non-loopback connection** (host:port + timestamp): into an **in-memory ring** (live tail; `GET /v1/net/egress` + the Overview widget) and, throttled to one row per host:port per window (`ELI_EGRESS_LOG_WINDOW`, default 300s), into the **tamper-evident audit ledger** as `net_egress` events. The ledger write is best-effort and runs on a background thread, so monitoring never delays or breaks a connection. Disable the ledger leg (keep the live ring) with `ELI_EGRESS_LEDGER=0`.
- The toggle changes *policy*; the socket failsafe still governs every actual connection, and every connection it allows is now on the record.
- Rationale: internet is "the final frontier to make ELI's world bigger" — available, but monitored (per-connection) and under the user's control, not a permanent lockout.

Honest assessment — web tier

- **Strong:** fail-closed auth, role separation, a tamper-evident HMAC'd audit trail, born-locked secret files, and a monitored (not absent) network path. For a self-hosted local AI this is well above the norm.
- **Watch:** RBAC is token-based and single-tenant-shaped; the injection guard (§3) is still pattern-based; `ELI_FULL_CONTROL` remains a blunt all-gates-off switch. None new, but they matter more once the web app is exposed beyond the owner's machine — keep it bound to trusted networks.

Update — 2026-07-02 (stable phone token, admin-gated model switch)

Two web-tier changes this cycle, both verified against the code:

- **Stable LAN token + rotate.** The phone's bearer token now **persists** across server restarts (`api/api_token.py`: env → a `0600` file under the config dir → generate-and-save), so a paired phone is no longer stranded every time the server bounces. A **rotate** button issues a fresh token and invalidates the old one in one tap — the manual override for a lost/compromised phone. `/health` stays tokenless by design.
- **Model switching is admin-gated and allowlist-bound.** The dashboard's model dropdown posts to `POST /v1/model`, which requires an **admin** principal and only accepts a path already in the installed-models list (`GET /v1/models/installed`). A dropdown of real files can't strand the runtime on a bad path, and a non-admin can't switch the model. (The list endpoint was moved off `/v1/models` to `/v1/models/installed` so it no longer collides with the OpenAI-compatible route.)

Neither weakens the existing posture — the socket failsafe, fail-closed command gate, and audit ledger all still apply.

Update — 2026-07-03 (token moved to the URL fragment; test reds cleared)

- **Token in the URL fragment, never the query string.** Every emitter of the phone link now builds `…/#token=…` instead of `…/?token=…`: the server's printed URLs *and* the desktop **Web Server panel** (`eli/gui/gui_eli_original.py:8332/:8391`) *and* both launchers (`scripts/eli_serve.sh`, `scripts/eli_serve.ps1`). A query string reaches the server — it lands in **uvicorn's access log** before any client-side strip can run — whereas a **fragment is never sent to the server**, so the token can't leak into the log. The page reads it from `location.hash`, stores it, and clears the address bar; it still accepts `?token=` for old links, so the transition breaks nothing. (An earlier commit fixed only the web tab + printed URL, leaving the GUI/launcher paths — most real usage — still leaking; this closes them.)
- **All 5 standing test reds cleared.** Deprecated `smart_home` plugin removed; 113 silent `except: pass` swallows made observable (987→874, ceiling 950→900); stale blueprint references fixed. Not a security change per se, but the suite is now fully green.

Blueprint — ELI MKXI ASCII Diagrams

Visual companion to `architecture.md`. Three views: the full request pipeline, the memory subsystem, and the gating stack. All grounded in the real modules (file paths are clickable).

---

1. Full pipeline (request lifecycle)

```
voice    STT (faster-whisper)          GUI text
           perception/local_whisper_stt
                        -
                    raw text

       ROUTER                            execution/router_enhanced.py
       priority pipeline (regex-first)   → {action, args, conf, matched_by}
       LLM-intent fallback                  cognition/llm_intent.py



       CognitiveEngine.process()         kernel/engine.py



[A] PHASE45      [B] ORCHESTRATOR                [C] QUICK PATH
FAST-PATH        (deep, non-quick)               (default, CHAT)
```

```
OS/media/        full 12-stage
status/job       (see §1b)                        AgentBus.dispatch()

       execute_action()      assembled context                DispatchResult
       VERBATIM (no LLM)                                       • grounding_confidence
       can't confabulate                                       • memory_context

   return
                                                   GROUNDING ESCALATION         (CHAT only)
                                                   factual? + low grounding?    runtime/grounding_esca

                                                    yes               no
                                         external→WEB / local→agents   proceed
                                         else → HEDGE ("won't guess")
                                                       return

                                                             self-contained?      return executor verb
                                                             grounded control?    grounded synthesis
                                                             plain CHAT


                  _build_enhanced_system()            persona(budget-trimmed)
                  (persona + evidence + brief, n_ctx budget)   + situation brief


                   inference_broker.infer() / stream    gguf_inference (model-agnostic)


                  CONFIDENCE GATE (Stage 12)    score vs threshold → PASS / repair


               OUTPUT GOVERNOR / sanitiser   cognition/output_governor.py
                                  (strip leaks, persona hygiene, no raw packets)

                      response    TTS (Piper)  /  GUI
```

1b. Orchestrator — the 12 stages (path [B]) `cognition/orchestrator.py`

```
 1 Intent   2 Persona Lock   3 HyDE   4 Planner


            5/6/7  PARALLEL RETRIEVAL

    keyword    FTS5     FAISS     RAG      KG



       8 Hybrid Merge   9 Rerank   10 Context Assembly
                        10.5 Persona Handoff   11 LLM Generation   12 Confidence
```

---

2. Memory subsystem `eli/memory` + `cognition/working_memory.py`

```
                        query / turn
```

```
   keyword    FTS5      FAISS      RAG      KnowledgeGraph
   (SQL)     full-txt  vectors    docs      entities/rel


conversation  memories_fts  index.faiss            kg_entities(+_fts)
_turns        memories      (vectors/)              kg_relations
(user.sqlite3)              embedder:
                            nomic-embed-…Q4_K_M.gguf


            HYBRID MERGE   RERANK   assembled context

            + working-memory pins (session-pinned, junk-guarded)

               situation brief   generation (persona handoff)

             STORES (artifacts/)
 db/user.sqlite3   memories(+_fts), conversation_turns, conversations,
                   kg_entities(+_fts), kg_relations, recall_log,
                   runtime_events, news_articles(+_fts), news_reflections,
                   habits/habit_events/habit_rules, observations,
                   learning_replay, working_memory_pins, user_patterns,
                   session_summaries, corrections, failures
 db/agent.sqlite3  agent_dispatches, agent_metrics, improvements,
                   failures, code_patches, error_tracking, observations
 vectors/index.faiss   semantic index
 runtime/users/<uuid>/user_profile…   per-user profile


WRITE PATH:  turn   PERSISTENCE GATE (drop junk/report-dumps)   store
```

---

3. Gating stack (in path order — every guard the request passes)

```
  1  NETGUARD  (process-wide, always-on)        core/netguard.py
       any outbound socket    offline?   OfflineError (FAIL-CLOSED)
       allow_network(): scoped opt-in window (model download, web tier)

  2  PHASE45 FAST-PATH GATE                     kernel/engine.py
       deterministic action (volume/media/date/job/status)?
            execute_action() VERBATIM, no LLM  (cannot confabulate)

  3  PERSISTENCE GATE                           runtime/persistence_gate.py
       about to write to memory? junk / internal report-dump?
            DROP (so it can't be stored and replayed as fact later)

  4  GROUNDING GATE                             runtime/deterministic_…gate
       answer requires evidence? unbacked or raw packet?
            BLOCK raw evidence/telemetry leak; require grounded synthesis
```

```
5   GROUNDING ESCALATION                        runtime/grounding_escalation
      checkable fact  AND  grounding < threshold ?
        external fact   WEB tier
        self/project    LOCAL tier     first tier that grounds   answer
        exhausted / offline         HEDGE  ("I won't guess")
      (trigger = GROUNDING, not the response score - which lies when wrong)

6   CONFIDENCE GATE  (Stage 12)                 kernel/engine.py
      response score vs threshold    PASS  /  repair pass

7   OUTPUT GOVERNOR / SANITISER                 cognition/output_governor.py
      final text    strip leaks, persona hygiene, no raw JSON    user


Legend:  FAIL-CLOSED = denies by default;  VERBATIM = returned without an LLM
         pass;  HEDGE = honest "can't verify" instead of a guess.
```

---

4. One-screen overview (how it all connects)

```
INPUT   ROUTER   ENGINE    FAST-PATH         (verbatim)   OUTPUT
                             ORCHESTRATOR   retrieval
                             QUICK   AGENT BUS

                        MEMORY (SQL/FTS5/FAISS/KG/WM)   context

                        GATES: netguard · persistence · grounding ·
                               escalation · confidence · governor

                        INFERENCE (broker → gguf, model-agnostic)   OUTPUT

                        background: proactive · self-improve · habits ·
                                    learning(LoRA) · world · scheduler
```

“‘

ELI — Capabilities & Actions (with activation phrases)

Auto-generated 2026-07-13 from `capability_manifest.json` *(209 capabilities; 184 routable). Regenerated by* `eli/tools/registry/capabilities_doc.py` *alongside the manifest.*

How to read this: every row is a real action ELI can route to and execute. Activation phrases are **representative** — the router is paraphrase- and typo-tolerant; a phrase shows the *shape* that fires the action, not the only accepted wording. `Plugin` = backed by a plugin (rest are core executor+router). Voice users prefix with the wake word (“computer, …”). Generation actions (documents/scripts/projects) run the evidence-planner first (real code/web/memory analysis) before synthesising.

Conversation & persona

Action	What it does	Example activation phrase(s)	Source
`CHAT`	Open conversation / fallback; persona-bound reply	anything not matching a command · “how are you” · “tell me a story”	Core

Action	What it does	Example activation phrase(s)	Source
EXPLAIN_LAST_RESPONSE	Explain why ELI gave its previous answer	"why did you say that" · "explain your last response"	Core
CLEAR_CHAT_HISTORY	Clear the current chat history	"clear chat history" · "start a new chat"	Core
PERSONA_LOCK_SET	Pin a persona/role for the session	"lock your persona to X"	Core
PERSONA_LOCK_CLEAR	Release a pinned persona	"clear persona lock"	Core
PERSONA_LOCK_STATUS	Show the current persona lock	"what persona are you locked to"	Core
PERSONA_REFRESH	Refresh/reload ELI's persona from the latest profile	"refresh your persona" · "reload your persona"	Core
HELP	List what ELI can do	"help" · "what commands do you have"	Core
LIST_CAPABILITIES	Enumerate ELI's capabilities	"what can you do" · "list your capabilities"	Core
MULTI_COMMAND	Run several chained commands from one utterance, in order	"close steam and set an alarm for 7am" · "open spotify then play X"	Core

App & window control

Action	What it does	Example activation phrase(s)	Source
OPEN_APP	Launch an application (installs it if missing)	"open spotify" · "launch firefox"	Core
CLOSE_APP	Close an application	"close spotify" · "quit chrome"	Core
FOCUS_APP	Bring an app's window to the front	"focus firefox" · "switch to the browser"	Core
OPEN_BROWSER	Open the default web browser	"open the browser"	Core
OPEN_URL	Open a URL in the browser	"open github.com" · "go to youtube.com"	Core
OPEN_IDE	Open the code IDE	"open the IDE"	Core
OPEN_FILE_SYSTEM	Open the file manager	"open files" · "open the file manager"	Core
OPEN_SYSTEM_SETTINGS	Open OS settings	"open system settings"	Core
OPEN_AUDIO_SETTINGS	Open sound settings	"open audio settings"	Core
OPEN_POWER_SETTINGS	Open power settings	"open power settings"	Core
OPEN_NETWORK_BROWSER	Open network settings	"open network settings"	Core
OPEN_COMMUNICATION_HUB	Open the comms hub (mail/chat apps)	"open communication hub"	Core
OPEN_MEDIA_HUB	Open the media hub	"open media hub"	Core
TILE_WINDOWS	Tile windows in a grid	"tile windows" · "2x2 grid" · "4x2"	Core
MAXIMISE_WINDOW	Maximise the focused window	"maximise the window"	Core
MINIMISE_ALL	Minimise all / show desktop	"minimise everything" · "show desktop"	Core

Action	What it does	Example activation phrase(s)	Source
`MINIMISE_WINDOW`	Minimise the focused window	“minimise the window” · “minimise this”	Core
`MINIMIZE_WINDOW`	Minimise the focused window (US spelling)	“minimize this window”	Core
`MINIMISE_APP`	Minimise a specific app’s window	“minimise spotify”	Core
`MINIMIZE_APP`	Minimise a specific app’s window (US spelling)	“minimize chrome”	Core
`HIDE_APP`	Hide an application’s window	“hide spotify” · “hide the window”	Core
`NEXT_WINDOW`	Switch to the next window	“next window”	Core
`PREVIOUS_WINDOW`	Switch to the previous window	“previous window”	Core
`RESTORE_WINDOWS`	Restore minimised windows	“restore windows”	Core
`SWITCH_WORKSPACE`	Switch virtual desktop/workspace	“switch to workspace 2”	Core
`SMART_HOME`	Control smart-home devices	“turn off the lights”	Plugin
`CONFIRM_PENDING_REMEDIATION`	Approve a pending fix/install offer	“yes” (to an “install X?” offer)	Core
`CANCEL_PENDING_REMEDIATION`	Decline a pending fix/install offer	“no” (to an “install X?” offer)	Core

Input control

Action	What it does	Example activation phrase(s)	Source
`VOLUME`	Adjust/mute system volume	“volume up” · “mute” · “set volume to 30”	Core
`KEYBOARD`	Send keystrokes / type text	“press enter” · “type hello world”	Core
`MOUSE_CONTROL`	Move/click the mouse	“left click” · “move the mouse up”	Core

Media

Action	What it does	Example activation phrase(s)	Source
`PLAY_MEDIA`	Play a track on the named platform	“play Juicy by Notorious B.I.G. on Spotify” · “play lo-fi on YouTube”	Core
`PAUSE_MEDIA`	Pause playback	“pause” · “pause the music”	Core
`STOP_MEDIA`	Stop playback	“stop the music”	Core
`NOW_PLAYING`	Report the current track and where it’s playing (Spotify or headless-mpv YouTube audio)	“what’s playing” · “what song is this”	Core

Action	What it does	Example activation phrase(s)	Source
NEXT_MEDIA	Next track	"next song" · "skip"	Core
PREVIOUS_MEDIA	Previous track	"previous track" · "go back a song"	Core
REPEAT_MEDIA	Repeat current track	"repeat this song"	Core
SHUFFLE_MEDIA	Toggle shuffle	"shuffle"	Core
MEDIA_CONTROL	Generic transport control (MPRIS)	"play/pause"	Core
SKIP_YOUTUBE_AD	Skip a YouTube ad when skippable	"skip the ad"	Core

Vision & screen

Action	What it does	Example activation phrase(s)	Source
SCREEN_READ_ANALYZE	Screenshot → vision model + OCR; answers questions about on-screen content	"what's on my screen" · "can you see this" · "what season is on the screen"	Core
SCREEN_LOCATE	Find on-screen UI text/element	"find the submit button on screen"	Core
SCREENSHOT	Capture a screenshot	"take a screenshot"	Core
OCR_IMAGE	Extract text from an image file	"ocr photo.png" · "read the text in image.jpg"	Core
ANALYZE_IMAGE	Describe/analyse an image file (VLM)	"describe cat.jpg" · "analyse this image"	Core
ANALYZE_PDF	Read/summarise a PDF	"summarise report.pdf" · "analyse this pdf"	Core
ANALYZE_PDF_FOLDER	Analyse all PDFs in a folder	"analyse the pdfs in that directory"	Core
ANALYZE_CSV	Analyse a CSV/spreadsheet	"analyse data.csv"	Core
AMBIENT_VISION	Toggle continuous screen-watching	"watch my screen" · "stop watching"	Core

Gaze (eye-tracking)

Action	What it does	Example activation phrase(s)	Source
GAZE_ENABLE	Enable webcam gaze control	"enable gaze control"	Core
GAZE_DISABLE	Disable gaze control	"disable gaze control"	Core
GAZE_CALIBRATE	Calibrate gaze tracking	"calibrate gaze"	Core
GAZE_STATUS	Show gaze tracker status	"gaze status"	Core
GAZE_CLICK	Click where the eyes rest	"open" / "left click" / "hit enter" (while gaze is on)	Core

Voice

Action	What it does	Example activation phrase(s)	Source
DICTATE	Start/stop dictation into the active field	"start dictation" · "stop dictating"	Core
TRANSCRIBE	Transcribe an audio file	"transcribe audio.wav"	Core
SPEAK	Speak text aloud (TTS)	"say good morning"	Plugin
LISTEN_FOR_COMMAND	Start listening for a voice command	"listen" / the wake word "computer"	Core
VOICE_DIAGNOSTICS	Run voice/STT diagnostics	"run voice diagnostics" · "stt diagnostics"	Core
WAKE_SET	Set your own wake word (any phrase); ELI persists it and trains the detector on it	"change the wake word to athena" · "set my wake word to atlas"	Core
WAKE_TRAIN	Train the local, self-supervised wake-word model (Piper-synth + noise/music augmentation; robust over background music)	"train the wake word"	Core
WAKE_ENROLL	Record your voice saying the wake word and retrain (personalisation, over music)	"enroll my wake word"	Core
TRAIN_VOICE	Learn your voice + tone — pitch/energy/rate, happy/angry/excited, and question-vs-statement; ELI then adapts its delivery to how you sound	"train my voice" · "learn how I speak"	Core

Files & notes

Action	What it does	Example activation phrase(s)	Source
CREATE_FILE	Create a file	"create a file notes.txt"	Core
CREATE_FOLDER	Create a folder	"make a folder called projects"	Core
READ_FILE	Read a file's contents	"read notes.txt"	Core
LIST_DIR	List a directory	"list the files in ~/Documents"	Core
FILE_AUDIT	Audit files for issues	"audit my files"	Core
SUMMARIZE_FILE	Summarise a document/file	"summarise report.docx"	Core
CONVERT_DOCUMENT	Convert between document formats	"convert report.md to pdf"	Core
WRITE_NOTE	Write a quick note	"write a note saying buy milk"	Core
NEW_NOTE	Create a new note	"new note"	Plugin
LIST_NOTES	List saved notes	"list my notes"	Plugin

Action	What it does	Example activation phrase(s)	Source
SEARCH_NOTES	Search notes	"search notes for budget"	Plugin
GET_CLIPBOARD	Read the clipboard	"what's in my clipboard"	Core
SET_CLIPBOARD	Write to the clipboard	"copy this to the clipboard"	Core

Generation (docs/code)

Action	What it does	Example activation phrase(s)	Source
GENERATE_DOCUMENT	Grounded document via the multi-stage pipeline (evidence → plan/outline → sections → review→revise)	"generate a document about X" · "write a report on Y"	Core
CREATE_DOCUMENT	Multi-stage grounded document (plan→draft→review); single-pass fallback	"create a document on X"	Core
DATA_FABRICATOR	Generate synthetic/test data	"fabricate a test dataset for X"	Core
GENERATE_SCRIPT	Generate a runnable script via the coding agent	"write a bash script to monitor the GPU"	Core
GENERATE_PROJECT	Scaffold a multi-file project	"generate a project that does X"	Core
CODE_SOLVE	Solve a coding task (plan→DAG→verify→repair)	"solve this: …" · "implement function X"	Core
FIX_FILE	Find and fix bugs in a file	"fix the bugs in foo.py"	Core
EXAMINE_CODE	Tiered error scan of named files/codebase	"examine eli/memory/memory.py for errors" · "review the codebase for bugs"	Core
CONFIRM_CODE_FIX	Apply the offered code fixes (pending-confirm)	"yes, fix it" · "fix the logic ones too"	Core
CANCEL_CODE_FIX	Decline the offered code fixes	"no, leave it"	Core
SHOW_DIFF	Show a code diff	"show the diff"	Core
OPEN_IN_IDE	Open a generated artifact in the IDE	(post-action; auto after generate)	Core

System status & info

Action	What it does	Example activation phrase(s)	Source
CPU_USAGE	Current CPU usage	"cpu usage"	Plugin
RAM_USAGE	Current memory usage	"how much ram is free"	Plugin
GPU_STATUS	GPU/VRAM status	"gpu status"	Core
SYSTEM_STATS	Combined system stats	"system stats"	Plugin

Action	What it does	Example activation phrase(s)	Source
HARDWARE_PROFILE	Detected hardware profile	"show hardware profile"	Core
GET_STATUS	General status	"status"	Core
TIME	Current time	"what time is it"	Core
GET_TIME	Current time (alt)	"tell me the time"	Core
DATE	Current date	"what's the date"	Core
GET_DATE	Current date (alt)	"what day is it"	Core
GET_WEATHER	Weather for a place	"weather in Wexford"	Plugin
NEWS_FETCH	Synthesised news digest	"what's the news" · "catch me up"	Core
WEB_SEARCH	Web search (Net-gated)	"search the web for X"	Plugin
MESSAGE_TIME_QUERY	Time-of-a-message query	"when did I say that"	Core
CHECK_CHRONAL_ALIGNMENT	Playful date/time sanity check	"check chronal alignment"	Core

Grounded introspection

Action	What it does	Example activation phrase(s)	Source
RUNTIME_STATUS	Live runtime: model, ctx, GPU layers, batch	"what are you running on" · "runtime status — everything"	Core
RUNTIME_AUDIT	Full runtime audit (what's broken/missing)	"run a full runtime audit"	Core
IMPORT_AUDIT	Which imports are failing/missing	"what imports are failing"	Core
RESOLVE_RUNTIME_PATHS	Resolved paths of critical files	"show the resolved runtime paths"	Core
GUI_RUNTIME_AUDIT	Audit GUI wiring with file-read proof	"audit your gui file" · "scan the gui runtime wiring and prove every hook"	Core
DIAGNOSE_WRAPPERS	Show the executor middleware/wrapper chain	"show the executor wrapper chain"	Core
COGNITION_STATUS	Cognition runtime status	"cognition status"	Core
EXPLAIN_COGNITION_RUNTIME	Explain the cognition runtime verbatim	"explain your cognition runtime"	Core
MEMORY_STATUS	Memory subsystem status / counts	"how does your memory work"	Core
MEMORY_STATS	Memory statistics	"memory stats"	Core
EXPLAIN_MEMORY_RUNTIME	Explain the memory runtime verbatim (4 DBs)	"explain exactly how your memory works internally"	Core
AWARENESS_STATUS	What ELI is currently aware of	"what are you aware of"	Core
FRONTIER_STATUS	Full system wiring matrix	"run a full system wiring matrix"	Core
ELI_IDENTITY_AUDIT	Audit ELI's identity provenance	"audit your identity"	Core
CODE_CHANGES	Recent code changes	"what code has changed"	Core

Action	What it does	Example activation phrase(s)	Source
NAME_SOURCE_AUDIT	Where ELI's knowledge of your name comes from	"how do you know my name"	Core
REASONING_MODE_STATUS	Current reasoning mode	"what reasoning mode are you in"	Core
EXPLAIN_ALL_REASONING_MODES	Describe all 5 reasoning modes	"explain all your reasoning modes"	Core
SELF_REPORT	What ELI has recently done/worked on	"what have you been working on"	Core

Memory & profile

Action	What it does	Example activation phrase(s)	Source
MEMORY_STORE	Store a durable fact	"remember that my sister's name is Anna"	Core
MEMORY_RECALL	Recall stored facts	"what do you remember about my car"	Core
PERSONAL_MEMORY_SUMMARY	Summary of what ELI knows about you	"what do you know about me"	Core
PERSONAL_MEMORY_DEEP_EXPLAIN	Deep, sourced profile explanation	"explain everything you know about me and where it's stored"	Core
USER_IDENTITY_SUMMARY	Who the user is	"who am I"	Core
REFRESH_USER_INFO	Re-extract the user profile	"refresh my user info"	Core
SET_USER_NAME	Set/correct the user's name	"my name is Alex" · "call me Alex"	Core

Self-maintenance

Action	What it does	Example activation phrase(s)	Source
SELF_ANALYZE	Analyse own failures/metrics	"analyse your failures"	Core
SELF_IMPROVE	Run a self-improvement patch cycle	"improve yourself"	Core
SELF_PATCH	Apply a self-improvement patch	"patch yourself"	Core
SELF_IMPROVEMENT_LOG	Show the self-improvement log	"show your self-improvement log"	Core
SELF_UPGRADE	git pull → deps → rebuild indexes	"upgrade yourself"	Core
SELF_UPDATE	Self-update (control-contract)	"update yourself"	Core
SELF_TEST	Run internal self-tests	"run a self test"	Core
RUN_TESTS	Run the pytest suite and summarise the results document	"run the test suite" · "generate a test report"	Core

Action	What it does	Example activation phrase(s)	Source
GENERATE_TESTS	ELI writes + sandbox-verifies behavioural tests for its own functions (Phase 4)	"generate tests for your code" · "grow your test coverage"	Core
LORA_STATUS	Report LoRA fine-tune readiness (preflight: modules, base model, reviewed data)	"lora status" · "is lora ready"	Core
LORA_TRAIN	Run the LoRA training pipeline DAG (pre-flight→build→train→eval); dry-run from chat, real training via the overnight task	"train a lora" · "fine-tune yourself"	Core
ORCHESTRATION_STATUS	Explain the agent DAG orchestrator: execution layers, dependencies, critical path, and last run	"orchestration status" · "show your agent dag"	Core
TEST_REVIEW	Run the suite, back up the report + write an errors file, summarise results, and offer result-driven fix options	"test review" · "run the tests and tell me what to fix"	Core

Tasks, time & planning

Action	What it does	Example activation phrase(s)	Source
SCHEDULE_TASK	Schedule overnight/timed work (code/research/etc.)	"research X overnight" · "build Y at 2am"	Core
SET_ALARM	Set an alarm	"set an alarm for 7am"	Core
SET_TIMER	Set a timer	"set a timer for 10 minutes"	Core
ADD_EVENT	Add a calendar event	"add an event tomorrow at 3pm"	Core
LIST_EVENTS	List events	"list my events"	Core
POMODORO_START	Start a pomodoro	"start a pomodoro"	Plugin
POMODORO_STOP	Stop the pomodoro	"stop the pomodoro"	Plugin
POMODORO_STATUS	Pomodoro status	"pomodoro status"	Plugin
BACKGROUND_JOBS	List background/scheduled jobs	"show background jobs"	Core
CHECK_JOB	Check a specific job	"check job 5"	Core
HABIT_STATUS	Show learned habits	"show my habits"	Core
CONFIRM_HABIT	Approve a proposed habit	"yes" (to a habit offer)	Core
DECLINE_HABIT	Decline a proposed habit	"no" (to a habit offer)	Core
EXECUTE_GOAL	Execute a stored goal	"execute goal X"	Core

Proactive

Action	What it does	Example activation phrase(s)	Source
PROACTIVE_START	Start the proactive daemon	"start proactive mode"	Core
PROACTIVE_STOP	Stop the proactive daemon	"stop proactive mode"	Core
PROACTIVE_STATUS	Proactive daemon status	"proactive status"	Core
GET_PROPOSALS	ELI's self-generated proposals/goals from observed signals	"do you have any proposals for me" · "what's on your agenda"	Core
MORNING_REPORT	Daily briefing (news/weather/agenda)	"morning report" · "give me my briefing"	Core

Plugins

Action	What it does	Example activation phrase(s)	Source
PLUGIN_LIST	List installed plugins	"list plugins"	Core
PLUGIN_SEARCH	Search for plugins	"search plugins for weather"	Core
PLUGIN_INSTALL	Install a plugin	"install the X plugin"	Core
PLUGIN_UNINSTALL	Uninstall a plugin	"uninstall the X plugin"	Core
PLUGIN_ENABLE	Enable a plugin	"enable the X plugin"	Core
PLUGIN_DISABLE	Disable a plugin	"disable the X plugin"	Core

Shell & advanced

Action	What it does	Example activation phrase(s)	Source
RUN_CMD	Run a shell command (gated/confirmed)	"run the command: ls -la"	Core
SHELL_EXEC	Execute shell (gated/confirmed)	"execute: df -h"	Core
SEQUENCE	Run a multi-step action sequence	"do X then Y then Z"	Core
ROUTING_FAULT_EXPLAIN	Explain why an input failed to route	"why did that fail to route"	Core
NOOP	Internal no-op (ignored fragment guard)	(internal — fragmentary input is dropped)	Core

New / undocumented (add a phrase above)

Action	What it does	Example activation phrase(s)	Source
SET_COMMUNICATION_STYLE	—	(needs activation phrase)	Core

Aliases & internal actions (not directly user-triggered)

Synonyms normalised to a routable action, post-actions, or internal pipeline steps:

ANSWER, CHECK_TARGET_STATUS, CLOSE_APPLICATION, CREATE_DOC, CREATE_SCRIPT, DIRECT_RESPONSE, DOC_GENERATE, EXIT_APP, EXPLAIN_LAST_FAILURE, GENERATE_CODE, IMAGE_STATUS, LAUNCH_APP, OPEN_APPLICATION, PLUGIN_STATUS, PREPARE_REMEDIATION, QUIT_APP, SAY, SEQUENCE_STEP, SET_AI_MODE, STT_DIAGNOSTICS, TEMPLATE, USER_INFO_REPORT, WRITE_CODE, WRITE_DOCUMENT, WRITE_SCRIPT

Coverage

- Routable actions documented: **183/184**
- New/undocumented (need a curated phrase): SET_COMMUNICATION_STYLE