Get It. · Technical Writeup

> *"What I cannot create, I do not understand." · Richard Feynman, last blackboard at Caltech, February 15, 1988.*

A developer-oriented walk through the codebase. The goal is that a contributor who has never read the project before can finish this document and confidently navigate the source, change a piece of behaviour, or replace a layer.

Mental model

Get It. is a desktop app for studying a single PDF at a time. Drop a file — a fast, model-free quality gate first confirms the PDF is a digital, text-based document under 150 pages; a scanned, image-dominant, or text-sparse file is rejected up front with a clear message instead of silently feeding empty text to the agents. Once a file clears the gate, two pipelines fire in parallel from upload. The first one ships visualizations inline next to the text so the document is immediately easier to read. The second one builds a concept graph of the same document and scores the student's mastery on four orthogonal axes as they interact with four study tools (chat, flashcards, forced-choice quizzes, Feynman with a curious child). Every interaction lands in one append-only journal on disk. One evaluator agent reads that journal and updates the four-axis scores after every completed session.

```
upload  ─► quality gate (model-free) ─► pdfjs-dist extracts text + glyph bboxes per page
          │
          ├──► visualizer pipeline
          │     ├─ batched concept-detection agent  →  DetectedConcept[] with anchor strings
          │     │   (≤5 pages per call, concurrency 3)   (each concept carries its page)
          │     └─ per-tag visualization-spec agent  →  3d / 2d-anim / formula / graph / 2d-text spec
          │        (lazy: on click by default)           (server-side syntax preflight + client-side
          │                                               runtime repair loop on sandbox crashes)
          │
          └──► knowledge-graph pipeline
                ├─ kg-build agent (one-shot)            →  6–25 concept nodes + typed edges + global note
                │  ◄── full document text                   (bounded by the 150-page upload cap)
                └─ kg-evaluate agent (incremental)      →  per-node {memory, comprehension, structure,
                   ◄── current graph (baseline scores)     application} 0–100, monotone non-decreasing
                   ◄── interactions since the last pass
```

Persistent state is a tree of plain JSON files under one OS-native user-data directory. There is no database, no hosted backend, and no shared key pool. Every model call goes through one **provider router** (`lib/codex.ts`) and runs against the end user's own AI account or API key — OpenAI Codex, Anthropic Claude, Google Gemini, or any OpenAI-compatible endpoint (including a local Ollama model). The user picks the engine in the setup wizard and can switch at any time; the rest of the app is provider-agnostic. The only network call beyond that model traffic is an anonymous open/update ping (a random install id + app version + OS, no document content or PII) used purely for aggregate user counts on the marketing site, and disabled entirely by `GETIT_DISABLE_ANALYTICS=1` .

Code map

The product is a single Next.js 16 application wrapped in a small Electron shell.

```
electron/                   desktop shell + setup wizard + auto-update
  main.js                   single-instance lock, data dir resolution, server spawn,
                            process-tree kill, theme-aware native chrome
  setup.js                  per-engine setup: install/verify CLI, OAuth or API-key wizard
  updater.js                GitHub Releases poll, in-app installer flow
  analytics.js              anonymous open/update ping (opt-out via env)
  wizard/                   wizard HTML/JS for the first-launch + update windows
  codex-bin/<triple>/       bundled Codex CLI per platform/arch (native binary)
  claude-bin/<triple>/      bundled Claude Code CLI per platform/arch (native binary)
  gemini-bin/gemini-cli/    bundled Gemini CLI (pure-JS, platform-independent)
  pi-bin/pi-cli/            bundled Pi/BYOK CLI (pure-JS, platform-independent)
  preload*.js               context-isolated preload bridges

app/                        Next.js App Router pages + API routes
  page.tsx                  upload home
  library/                  catalog of every opened PDF
  viewer/[docId]/           per-document viewer (PDF + right pane)
  api/
    upload/                 pdfjs extraction + quality gate (+ Markdown→PDF on .md) + docId
    analyze-pdf/            legacy single-shot detection (preserved for tests)
    tags/[docId]/           server-owned tag store: GET / POST active-tag / etc.
    jobs/detect/[docId]     POST → kicks the batched concept-detection job for a doc
    jobs/viz/[docId]        POST → kicks per-tag viz-spec generation
    chat/[docId]            POST → chat turn on a provider-tagged thread (start / resume)
    flashcards/[docId]      POST generate / rate / end (triggers scheduleEvaluation)
    quizzes/[docId]         POST generate / answer / end (triggers scheduleEvaluation)
    feynman/[docId]         POST start / explain (triggers scheduleEvaluation)
    kg/[docId]/             build / state / evaluate
    work-context/[docId]    download the journal as JSON
    settings/               GET / POST persisted AppSettings (provider, models, theme, …)
    provider/status         active engine's auth state + account/usage (banner+account poll)
    provider/logout         provider-agnostic sign-out / key-clear + usage reset
    codex/health            process-local provider health mailbox (banner polls this)
    codex/account|logout    legacy Codex-specific account + sign-out (kept for compat)
    pi-proxy/[...path]      BYOK request proxy to the user's OpenAI-compatible endpoint

components/                 React UI (orchestrators + scene renderers + tag UI)
  RightPane/                mode dropdown + the four tool views + KG view
  Visualizer/               3D / 2D / formula / graph / text renderers + sandbox
  PdfViewer.tsx             pdf.js viewer with overlay tag pills
  CodexHealthBanner.tsx     error banner + countdown + re-connect

lib/                        framework-agnostic helpers
  codex.ts                  the provider router: runJson + runJsonInThread, health
                            mailbox, usage capture, universal call-timeout backstop
  provider-types.ts         AIProvider interface, ProviderName, labels, auth-kind map
  providers/                one adapter per engine, all behind the router
    codex-provider.ts       OpenAI Codex via @openai/codex-sdk (native binary)
    claude-provider.ts      Anthropic Claude via the claude CLI (native binary)
    gemini-provider.ts      Google Gemini via the gemini CLI (pure-JS)
    pi-provider.ts          BYOK / local via the pi CLI (pure-JS)
    cli-runner.ts           shared subprocess runner: bundled-binary resolve, stdin
                            prompt, abort/timeout, non-detached for group-kill
  codex-errors.ts           pure error model: classifier + friendly payloads (no SDK dep)
  usage-store.ts            per-provider cumulative token/cost usage at usage.json
  settings-store.ts         persisted AppSettings (provider, per-tier models/efforts,
                            BYOK config, theme) at settings.json
  config.ts                 env defaults for the two viz knobs
  pi-coder.ts               pi CLI launch config (execPath + undici shim) + path resolve
  md-to-pdf.ts              Markdown → text-bearing PDF (marked + pdfkit) for .md upload
  agents/                   per-agent prompt builders (detect, viz)
  kg.ts / kg-runner.ts      KG persistence + build/eval runners + scheduler
  store.ts / tags-store.ts  doc cache + tag store, atomic filesystem persistence
  paths.ts                  the OS-native data-dir resolver (+ docId validation)
  pdf-extract.ts            pdfjs-dist text + bbox extraction + upload quality gate
  schemas.ts schemas-kg.ts  JSON schemas for every agent
```

```
work-context*.ts         journal storage + evaluator summary
viz-runtime.ts           the `new Function` sandbox compiler
```

The agent layer

Every model call funnels through the **provider router** in `lib/codex.ts`. (The file keeps its Codex-era name and public surface — `runJson`, `runJsonInThread`, `CodexError`, `getCodexHealth` — so no downstream route needed an import change when the app went multi-engine.) The stateless workhorse is `runJson(prompt, outputSchema, opts)`; the chat tool additionally uses `runJsonInThread(...)`, which **starts or resumes a thread** so a multi-turn conversation transmits the document once and each follow-up turn carries only the new message. Both paths read `loadSettings().provider`, delegate to the matching adapter, and wrap it in machinery every provider shares:

1. **Delegates to one of four adapters** behind a common `AIProvider` interface (`lib/provider-types.ts`): `CodexProvider` (the `@openai/codex-sdk` client), `ClaudeProvider` and `GeminiProvider` and `PiProvider` (each driving its vendor CLI as a subprocess via the shared `cli-runner`). The router holds one singleton per provider and selects by the active setting; the rest of the app never knows which engine answered.
2. **Pins the model and effort per engine, per tier.** Each tool asks for a `reasoning` tier (`low` for snappy one-shots like detection/viz, `high` / `medium` for chat and evaluation) and the adapter maps that to the user's configured fast/smart model and thinking-effort for *that* engine — Codex defaults `gpt-5.5` (low vs high effort), Claude `sonnet` / `opus` (medium/high), Gemini `gemini-flash-latest` / `gemini-pro-latest`. Pinning the model matters across all engines: left blank, a CLI resolves a default itself — from a user config or one baked into the binary — which can be a model the vendor has since retired for that auth mode, 400-ing everyone without a local override. Codex runs `sandboxMode: "read-only"`, `approvalPolicy: "never"`, `skipGitRepoCheck: true` in a scratch working dir; the CLI engines run in an empty scratch cwd (no project config leaks in) and receive the prompt **over stdin**, not as an argv `-p` — a full-document prompt easily exceeds the OS command-line limit (`E2BIG`) on Linux/Windows.
3. Runs the turn against the supplied JSON Schema (each engine's structured-output mode: the SDK's schema, Claude's `--json-schema`, Gemini's response schema), retries once on parse failure, and returns the typed result.
4. **Captures usage.** After every successful call the router normalises the engine's raw usage into a common `{inputTokens, outputTokens, costUsd}` shape and accumulates it per provider in `usage-store.ts` — so the Account panel can show tokens for API-key engines and the subscription's own limits for account engines.
5. Catches every throw, classifies it into `auth_lost` / `rate_limit` / `binary_missing` / `model_unsupported` / `generic`, and writes the result into a **per-provider health mailbox**. The renderer polls `/api/codex/health` to render a banner whose copy is filled with the *active engine's* label. `model_unsupported` (the pinned model aged out server-side) tells the user to update the app. Rate-limit deadlines are parsed from the message when present, with a conservative fallback cooldown otherwise so the short-circuit below always has a window to act on.
6. **Short-circuits** future calls while a rate-limit window is still active, and bounds **every** call (every tool, every engine) with a hard wall-clock timeout that aborts the subprocess — so a stuck network call or a CLI's own internal backoff can never hang a spinner forever.

The pure error model (the `CodexError` kinds, the classifier, and the friendly per-kind payloads the request/response routes return) lives in a separate `codex-errors.ts` with no SDK import, so it stays unit-testable in isolation; the friendly copy is parameterised by the active provider's label. `codex.ts` re-exports it.

Nine prompts live behind that router, unchanged across engines:

WHERE	WHAT IT RETURNS	SCHEMA
`lib/agents/detect.ts`	`DetectedConcept[]` for a batch of up to 5 pages (each concept tagged with its page)	`detectionSchema` in `lib/schemas.ts`
`lib/agents/viz.ts`	Per-tag visualization spec (one of five renderer types)	`vizSchemaFor(type)` in `lib/schemas.ts`
`lib/kg-runner.ts → BUILD_SYSTEM`	The graph: 6–25 nodes, typed edges, global note	`kgBuildSchema` in `lib/schemas-kg.ts`
`lib/kg-runner.ts → EVALUATE_SYSTEM`	Per-node updates {memory, comprehension, structure, application} + notes	`kgEvaluateSchema`
`app/api/chat/[docId]/route.ts`	One assistant reply	`chatReplySchema`
`app/api/flashcards/[docId]/route.ts`	4–10 Q / A cards	`flashcardsGenerateSchema`
`app/api/quizzes/[docId]/route.ts`	4–8 MCQs with one correct option and three distractors	`quizGenerateSchema`
`app/api/feynman/[docId]/route.ts → CHILD_SYSTEM`	One curious-child prompt	`feynmanChildPromptSchema`
`app/api/feynman/[docId]/route.ts → SUMMARY_SYSTEM`	End-of-session honest summary	`feynmanSummarySchema`

There is no god-prompt and no client-side JSON-Schema validation. Every agent reply arrives as a typed TypeScript object the rest of the code can use without defensive parsing.

The visualizer pipeline

The pipeline that ships time-to-value: tags appear inline the instant detection returns. By default a tag's visualization is generated **lazily** — the first time the student clicks the tag — so model usage stays proportional to what they actually open; this matters on long documents, where a single PDF can carry hundreds of tags. An opt-in `auto-generate` setting flips this back to eager mode, rendering every tag in parallel as soon as detection finds it. Either way, a ready tag is marked with a thin emerald ring so the student can tell at a glance which visualizations already exist.

Server-side jobs. Detection and per-tag viz generation are not renderer loops. Both are first-class **server-side jobs**, singleton-per-doc, idempotent, running inside the Next process. The detection job walks unanalysed pages in batches of up to five pages per model call, runs those batches at concurrency 3, and persists each batch of new tags to `<DATA_DIR>/docs/<docId>/tags.json` as it goes — each detected concept carries the page it belongs to, so one call can tag five pages at once and a 100-page document costs roughly twenty detection calls instead of a hundred. The viz job picks the next tag whose state is `generating: true`, runs the per-type agent at concurrency 4, and persists the spec back to the same file. The viewer is a *consumer*: it polls `GET /api/tags/<docId>` every 1.5 s while any job is in flight, fires `POST /api/jobs/viz/<docId>` on a user click or a sandbox runtime-error report, and only ever updates the *active-tag selection* on the server. The active selection is the lone field the client can write; everything else is server-owned, so a concurrent client navigation cannot overwrite mid-flight detection or generation.

Reopening a doc from the Library weeks later therefore restores the exact tag layout, viz specs, and analysed-pages set without re-detection. Library badges poll the same source so they stay live across the whole catalog with no extra plumbing.

Five renderer types. `lib/agents/viz.ts` routes by `VizType`:

- **`3d`**. The agent emits a JavaScript function body that `components/Visualizer/ThreeDView.tsx` executes with `{ THREE, scene, camera, renderer, controls, group }` in scope. The viewer auto-frames the molecule with a bbox and auto-rotates the group.
- **`2d-anim`**. Same shape, but the function body returns an object with `draw(ctx, width, height, time, dt)` and runs every frame on a Canvas2D context.
- **`formula`**. A headline LaTeX line plus 2–6 derivation steps with one-sentence explanations; rendered with KaTeX.

- `graph`. A `chart_type` (function / points / bars / lines) plus a JSON-string `data_json`; plotted on a Canvas.
- `2d-text`. Title plus caption plus markdown body plus citation list. Used for legal articles, named papers, and authoritative quotations. Web search is enabled only for this type.

The sandbox. `lib/viz-runtime.ts → compileFn` wraps each LLM-emitted function body in an IIFE that shadows the dangerous globals (`window`, `document`, `fetch`, `XMLHttpRequest`, `WebSocket`, `require`, `Function`, `eval`, `globalThis`, `self`, `process`, `navigator`, `location`, `localStorage`, `sessionStorage`) as `undefined` parameters before the inner function runs. Shadowing alone leaves one hole — `({}).constructor.constructor` recovers the real `Function` even when the identifier is shadowed, and through it the real `fetch` off the global scope — so for the synchronous span of the model code the compiler also neutralises `constructor` on every function-kind prototype (`Function`, `AsyncFunction`, `GeneratorFunction`, `AsyncGeneratorFunction`) and restores the exact original descriptors in a `finally`. It does this through `Object.defineProperty` rather than a plain assignment, because `AsyncFunction.prototype.constructor` is spec'd non-writable (`{ writable: false, configurable: true }`) — a direct write throws in strict mode — and wraps each step in try/catch so a stricter engine degrades to plain shadowing instead of crashing the render. The boundary is a defense against LLM mistakes, not against adversarial input: the user is running their own AI account against their own PDFs.

Repair loop. When the sandbox throws inside `ThreeDView`'s `setup_code` or in a `2d-anim` `draw`, the viewer reports the error string back to the server, which hands it to the active engine as repair context (the broken `setup_code` + the captured error message) and asks for a corrected JSON object that compiles and runs end-to-end. The user sees "repairing, attempt N of M" instead of red text. Server-side syntax pre-flight via `new Function(...)` catches truncated bodies before they ever leave the route.

The knowledge-graph pipeline

This is the layer that turns Get It. from a viewer into a measurement instrument. Two agents, one persistence file, one queue.

`kg-build` runs once per document at upload time. The system prompt asks for 6–25 concept nodes the student would actually need to master (not a glossary), typed edges (prerequisite / composition / causal / specialisation / contrast), and a short global note that the viewer prints above the graph. Output is written to `<DATA_DIR>/docs/<docId>/kg.json` with `status: "ready"`. Any failure — an account-level engine error or a one-off — moves the graph to `status: "error"` with the reason, so the KG view drops its spinner and shows a **Retry** button. There is no automatic retry: the build re-runs on demand, and the health banner explains an account-level failure in the meantime.

`kg-evaluate` is the four-axis rubric. Every node carries four 0–100 scores:

AXIS	WHAT IT MEASURES	STRONGEST SIGNAL
memory	recall over time	flashcard ratings (1–4), quiz correctness on definitional questions, recall references in chat
comprehension	understanding in the student's own words	original metaphors in chat, plain-language Feynman explanations, distractor-rejection in quizzes
structure	grasp of how concepts connect	multi-step reasoning that bridges concepts, references to prerequisites, sibling discrimination
application	transfer to new cases	original examples, edge cases, novel problem solving, applied-tier quiz answers

The evaluator sees the current graph with each node's previous scores as a baseline, plus **only the interactions since the last pass** (compacted via `summariseForEvaluator`, which filters the journal by timestamp). Earlier evidence is already encoded in the baseline, so a pass stays cheap no matter how long the journal grows — and because scores only ever rise, nothing is lost by not re-reading the old transcript. Its system prompt enforces three rules: scores are **monotone non-decreasing**, *quantity does not entitle a score*, and concepts with no observable evidence stay at their previous level. The runtime enforces the monotone rule with a clamp on every update (`clampMonotone` in `lib/kg-runner.ts`) so a chatty interaction cannot accidentally erase prior evidence even if the agent disregards its own instruction.

Scheduling. Each evaluator pass is one model turn at medium effort, coalesced through a per-doc queue with at most one in-flight pass and one pending. Flashcards, quizzes, and Feynman fire `scheduleEvaluation(docId)` when a session-worth of evidence lands (a deck closes, a quiz ends, a Feynman session wraps) and return immediately. Chat is chatty by definition, so it is deliberately **not** evaluated per reply: the student chats freely and the client fires a single pass when they leave the Chat tab (`POST /api/kg/[docId]/evaluate`). A pass that finds no new interactions since the last one returns without spending a call. The client polls `/api/kg/[docId]/state` (which exposes the live `evaluating` flag), accelerating to 2.5 s while the agent is working and slowing to 6 s when idle. The badge in the top tab bar reads "Building graph", "Evaluating", "No evaluations yet", or "Synced 12 s ago" depending on what the queue is doing.

An evaluator pass that fails — including on an account-level engine error — simply stops; evaluation is best-effort background scoring, so the next genuine tool interaction schedules a fresh pass and the graph catches up then. We deliberately do **not** auto-retry on a timer (see *Resilience to engine outages* for why that pattern was removed).

The four study tools and the work-context journal

The four tools are deliberately small and deliberately different. Each provides a distinct evidence type.

- **Chat.** Multi-turn, multi-thread, scoped to one document. The first turn opens a thread seeded with the knowledge-graph node list and the full document text; later turns resume that thread (by stored `threadId`) and send only the new message, so the document is transmitted once per conversation instead of re-injected on every reply. The thread records **which provider** created it: resume only when the active engine still matches, otherwise the route transparently restarts the thread with the full prior history as context — so switching engine mid-conversation continues seamlessly instead of resuming a session the new engine can't read. The student can chat freely across as many turns as they like; a single KG re-evaluation runs when they leave the Chat tab.
- **Flashcards.** Open-recall under self-grade. The student picks a topic (or "all"), the engine generates a 4–10 card deck, the student optionally types their answer, reveals, and self-grades 1–4 (Again / Hard / Good / Easy, the FSRS convention). Ratings are recorded per card; closing a deck triggers an evaluator pass.
- **Quizzes.** Forced-choice discrimination. The engine generates a 4–8 question multiple-choice quiz; each item carries one correct option and three plausible distractors picked to expose the confusion a student would actually trip on. The server **shuffles the options** at generation time with `crypto.randomInt`-driven Fisher–Yates so the agent's positional bias (the model tends to put the right answer at index 0) does not leak to the UI. The student picks, gets immediate feedback with a one-sentence explanation, and the quiz ends with a score summary.
- **Feynman.** The agent plays a curious eight-year-old who asks 3 to 4 short, pointed questions. The student is forced into the role of the teacher. After the last turn a separate summary call writes a 3–6-sentence honest read of where the explanation held and where it broke down. The session is bounded so the data stays usable for the evaluator and the student does not drift.

Behind all four sits one artifact: the **work-context JSON**, one file per doc on the server, append-only by convention. Every chat message, every card rating, every quiz answer, every Feynman turn lands here with a UTC timestamp. It is the file the evaluator reads, the file the student can download from the right-pane menu, and by design the only thing the system needs to remember about a study session. Backwards-compatible loading (`loadWorkContext`) back-fills any array that did not exist when the doc's journal was first written, so quizzes added in v1.1.0 work cleanly against pre-quiz journals from v1.0.0; it back-fills new optional fields the same way — per-interaction timestamps (which the incremental evaluator filters on), the chat's `threadId`, and the provider that thread belongs to — so journals written before v1.2.0 evaluate and resume without a migration step.

Persistent state and the types-split pattern

Filesystem-backed under one OS-native data directory per user, resolved once in `lib/paths.ts`.

OS	PATH
macOS	`~/Library/Application Support/get-it/`
Windows	`%APPDATA%\get-it\`
Linux	`~/.local/share/get-it/`

Or whatever the Electron main pinned via the `GETIT_DATA_DIR` environment variable. Layout:

```
docs.json                                  # top-level catalog
docs/<docId>/source.pdf                    # original bytes
docs/<docId>/meta.json                     # { id, filename, uploadedAt, numPages, lastOpenedAt }
docs/<docId>/extracted.json                # cached pdfjs-dist output (text + bboxes per page)
docs/<docId>/tags.json                     # server-owned visualizer tags + viz specs
docs/<docId>/workctx.json                  # the journal: chats / flashcards / quizzes / feynman
docs/<docId>/kg.json                       # the knowledge graph + per-node scores
codex-scratch/                             # the engines' per-call scratch working dir
logs/                                      # embedded server stderr
settings.json                              # provider, per-tier models/efforts, BYOK, theme, viz knobs
usage.json                                 # per-provider cumulative token/cost usage
```

Cheap, recoverable, OS-agnostic, and a clear seam to lift to a hosted backend if we ever want to.

Types-split pattern. Modules are split into pure-TS `*-types.ts` files (no `node:fs` imports) and storage helpers in `*.ts` that do touch the filesystem. Next.js bundles a transitively-imported module into the client when *any* type from it is referenced, including a bare `import type {}`. Splitting types into a node-free file is the only way to keep `lib/kg.ts` and `lib/work-context.ts` server-only without poisoning the browser bundle. The comments in those files say so explicitly.

Settings. `lib/settings-store.ts` owns the persisted `AppSettings` at `<DATA_DIR>/settings.json` — the active provider, per-tier fast/smart model and thinking-effort for each engine, the BYOK URL/key/type, the appearance theme, and the two viz knobs (`lib/config.ts` still supplies their env defaults: `NEXT_PUBLIC_AUTO_GENERATE_VIZ`, off so visualizations generate lazily on click, and `NEXT_PUBLIC_MAX_VIZ_GEN_RETRIES`). The dynamic localhost port the packaged app binds to changes on every launch, so anything cookie- or localStorage-scoped to the origin would forget the user's choice; a plain file in the user-data dir is the only thing that survives a restart. Writes are atomic (temp file + rename), and a change broadcasts a `getit:settings` window event so other pages on the same renderer — including the live theme switch and a mid-session engine switch — react without polling.

Appearance. The theme is `light` (default), `dark`, or `system`. The root layout sets the `dark` class during SSR from the persisted value and a tiny blocking pre-paint script resolves `system` against the OS before first paint (no flash); a `ThemeProvider` then reacts to live settings events and OS-preference changes. The palette itself is one set of CSS custom properties in `app/globals.css`, overridden under `html.dark`; components read the variables, so the same markup themes both ways. Electron's native chrome (the window background and the Windows title-bar overlay) reads the same persisted theme at window creation so it matches from the first frame.

Resilience to engine outages

Every model call funnels through `runJson` in the `lib/codex.ts` router. It classifies failures from any engine into five kinds — through the same regex battery, so a Claude auth error and a Codex auth error land in the same bucket — and writes the latest one into a **per-provider health mailbox**, keyed so switching engine surfaces that engine's own state.

KIND	TRIGGER	UI BEHAVIOUR
`auth_lost`	401 / token revoked / "sign in" message	Banner + "Re-connect" button re-opens the desktop setup wizard
`rate_limit`	429 / "try again in N" / 5-hour / weekly window phrases	Banner with a live countdown to `retryAt` (a fallback cooldown when the message carries no deadline); auto-clears when the window passes
`binary_missing`	the active engine's CLI not found at the resolved path	Banner + button to re-open the setup wizard
`model_unsupported`	"model is not supported" / the pinned model retired server-side	Banner telling the user to download the latest Get It.
`generic`	Anything else	Banner with the raw message

The in-app banner. The renderer polls `/api/codex/health` (fast cadence while there is an active problem, slow cadence otherwise) and renders `components/CodexHealthBanner.tsx`. The countdown updates locally so the banner stays smooth between polls.

Fail fast, retry by hand — never auto-loop. Earlier versions auto-resumed background work on a `setTimeout` keyed off `retryAt`. That had a sharp edge: an account quota message (a ChatGPT or Claude subscription, say) often carries no parseable deadline, so `retryAt` was undefined, the backoff gate (which keyed on it) was never taken, and the viz queue re-picked the same still-`generating` tags and re-hit the wall as fast as calls completed — a tight loop that re-opened the banner the instant the user dismissed it. The fix is two-part: every rate-limit now gets a concrete deadline (parsed, or a fallback cooldown) so the short-circuit always fires, and more fundamentally **no background job auto-retries on a model error**. Instead each surface stops cleanly and offers a manual retry:

- **Viz queue + detection** stop on any account-level error and drop the `generating` spinner from every still-pending tag, returning them to an idle, click-to-retry state. A generic per-concept failure marks only that tag and the queue moves on.
- **KG build** moves to an `error` state with a Retry button; **KG evaluation** just stops and is re-triggered by the next interaction.
- **Request/response tools** (chat / flashcards / quizzes / Feynman) return a friendly `{ kind, message }` instead of an opaque 500, and the view shows it inline next to a Retry control. Chat's send is **atomic** — the user message and the reply are committed together, only on success — so a retry never duplicates the turn; Feynman's "explain" turn is atomic the same way and rolls back its optimistic state on failure.

The one retry loop that stays is the visualizer's **code-repair** loop (a sandbox runtime error feeds the broken code back to the engine for a corrected spec, bounded by `max-repair-attempts`). That is a content-level fix for the model's own output, not a backend-outage retry, and it is unchanged.

`runJson` also short-circuits future calls while a rate-limit window is still active, and caps every call with a hard wall-clock timeout that aborts the underlying subprocess. A chatty UI cannot burn a hundred wasted calls hoping the next one succeeds, and no tool — on any engine — can leave a spinner hanging forever.

Bring-your-own-account as an architectural choice

The decision to drive every agent through the **user's own AI account, over that vendor's official CLI** is the choice that shapes the whole product. It is not cost-cutting and not a missing feature; it is a deliberate boundary — and as of v1.4.0 it spans four engines instead of one.

There is no server-side key for any provider, no shared pool of credits, and no app-side markup on model usage. The Electron shell bundles each engine's CLI; the first-launch wizard runs that engine's own sign-in; every subsequent call runs against the user's account at whatever tier they pay for. The app sees the same auth state the CLI sees — a successful login, a rate-limit window, an expired token — and nothing more.

The four engines and how they authenticate. A small registry (`PROVIDER_AUTH_KIND` in `lib/provider-types.ts`) marks each engine as an *account* or an *API-key* engine, and the wizard, account panel, and usage display all read from it:

ENGINE	CLI (TRANSPORT)	AUTH	USAGE SHOWN
OpenAI **Codex**	`@openai/codex-sdk` + native binary	ChatGPT account (OAuth) or OpenAI key	subscription limits (account)
Anthropic **Claude**	`claude` Code CLI, native binary	Claude Pro/Max (OAuth) or Anthropic key	subscription limits (account)
Google **Gemini**	`gemini` CLI, pure-JS	API key	tokens used
Pi / BYOK	`pi` CLI, pure-JS	any OpenAI-compatible URL + key (incl. local Ollama)	tokens used

For the OAuth engines the wizard drives the CLI's own browser login and polls its auth-status until it reports signed-in; for the key engines it captures the key (and, for BYOK, the endpoint URL and protocol) and verifies it with a real test call before letting the user continue. Switching engine later re-opens the same wizard. BYOK traffic is relayed through `/api/pi-proxy/[...path]`, which also strips `<think>` / `<thought>` reasoning blocks some local models emit.

Three properties follow.

1. **No second subscription, ever.** Other AI-study tools layer a marked-up fee on top of an API key the vendor holds. Get It. cannot do that, because it never holds the key in the first place. A paid tier (ChatGPT Plus, Claude Pro, …)

is the practical floor for sustained sessions; free tiers sign in but their allowance is intentionally small; a local Ollama model is free outright. Whichever it is, it is the access the user already has.

2. **No data resale and no transit-stage intermediary.** Because we never proxy model traffic through our own infrastructure (the BYOK proxy runs inside the local app and talks straight to the user's endpoint), there is none for that traffic to flow through. Work-context journals, knowledge graphs, and per-doc folders all live under the user-data directory on local disk. There is no document upload step and no cloud sync. The single exception is an anonymous open/update ping (a random install id + app version + OS, no document content or PII, opt-out via `GETIT_DISABLE_ANALYTICS=1`) that powers aggregate user counts on the marketing site. "Download your data" is a one-click affordance, but the more honest framing is that there is nothing else *to* download.
3. **The transport is genuinely pluggable.** When the app was Codex-only this was a promise; now it is the architecture. Every engine implements one `AIProvider` interface behind the `runJson` router, so adding a fifth is a new adapter plus a wizard entry, not a change to any tool. The router, the schema enforcement, the health mailbox, the usage store, and the timeout backstop are all written once and shared.

The same property protects the project legally. **Get It. is not affiliated with, endorsed by, or sponsored by OpenAI, Anthropic, or Google**, and is not a derivative work of any closed-source software from any of them; it is an independent application that interoperates with each vendor's publicly released CLI using the end user's own credentials. The student's use of any model through Get It. is governed by that provider's own Terms of Use, Usage Policies, and Privacy Policy. Those documents are authoritative.

Desktop packaging

The Electron shell is the boring kind of shell: it does as little as possible.

`electron/main.js` acquires a single-instance lock, normalises the user-data directory to `get-it` (overriding Electron's default `Application Support/Get It` so the path matches the pure-Next dev default), runs the setup wizard, spawns the Next.js standalone server as a child Node process on a free localhost port, points one Chromium `BrowserWindow` at `http://127.0.0.1:<port>`, and exports the resolved paths of all four bundled engine CLIs as env vars at boot so a mid-session switch never has to re-resolve them. It paints the native window background and (on Windows) the title-bar overlay from the persisted theme so the chrome matches from the first frame. There is no native menu reinvention, no custom IPC for application data, and no second renderer. The UI is the unchanged Next.js app.

We chose Electron over Tauri because we wanted a guaranteed Chromium runtime on every supported OS: Three.js, KaTeX, the `new Function(...)` LLM sandbox, and pdf.js fonts all behave identically on every machine the user can install on.

`electron/setup.js` owns the per-engine setup life-cycle, and **all four engines ship inside the app**. `scripts/electron-prepare.mjs` stages them at build time, two different ways depending on whether the CLI is a native binary or pure JavaScript:

- **Codex and Claude are native binaries.** Each is an npm optional dependency carrying a platform/arch executable. The prepare script fetches the correct platform tarball straight from the npm registry — so a cross-arch build from an Apple Silicon Mac still produces a usable Windows or Linux installer — and stages it under `electron/codex-bin/<triple>/` and `electron/claude-bin/<triple>/`. Versions are pinned (and Claude's is derived from the installed package so the bundle and the SDK never disagree).
- **Gemini and Pi are pure JavaScript.** They ship as their npm package trees under `electron/gemini-bin/` and `electron/pi-bin/` (Pi's `examples/` pruned to keep the bundle lean), platform-independent. Pi is launched through `process.execPath` with `ELECTRON_RUN_AS_NODE=1` and a small `--require` shim that polyfills `webidl.markAsUncloneable` — absent in the Node 20 that Electron 33 embeds, which the Pi/undici stack would otherwise crash on.

At runtime the providers resolve the bundled path first (overridable by a `*_BINARY_PATH` env var), then fall back to a CLI on `PATH` for development. Sign-in is per engine: the wizard drives the OAuth CLIs' own browser login and polls auth-status for the account engines, and captures + live-verifies the key (and BYOK URL/protocol) for the key engines. The wizard gates **Continue** until the chosen engine actually connects, so the main window never opens onto a dead engine. It is a stand-alone `BrowserWindow` loaded from a plain `file:///` page with a minimal context-isolated preload bridge.

Two boot guards worth knowing about.

- The `window-all-closed` handler **does not auto-quit** while a `bootstrapping` flag is true. Without that flag, dismissing the update modal or the wizard (which are both their own `BrowserWindow`) becomes the *last* open window and the implicit auto-quit fires before `whenReady` can reach `createMainWindow()`. The flag flips to false the instant the main window opens.
- `ELECTRON_RUN_AS_NODE=1` is unset at boot. If that env var leaks in, Electron loads as plain Node and `app` is undefined, which manifests as the cryptic `Cannot read properties of undefined (reading 'requestSingleInstanceLock')`. We catch that case and unset before any API touches `app`. (It is then set deliberately, and only for the Pi child, when that engine is launched.)

Nothing outlives the app. Every AI call spawns a CLI subprocess, and a multi-page detection run can have several in flight, so process hygiene is load-bearing. The Next server is spawned `detached` (its own process group on POSIX); the engine CLIs are spawned **non-detached** so they stay inside that group. On quit, `main.js` kills the whole tree with a negative-PID `process.kill(-pid, …)` group signal (SIGTERM then a short-fused SIGKILL), and a standalone `scripts/server-watchdog.cjs` is a dead-man's switch that group-kills the subtree if the parent vanishes uncleanly. An in-flight call's `AbortSignal` (from the universal timeout or a user navigation) kills its child directly. The net effect: closing the window, quitting, or a crash leaves no orphaned `codex` / `claude` / `gemini` / `pi` process behind.

Build and release pipeline

Multi-target builds run from `scripts/build-electron.mjs`. Locally:

```
node scripts/build-electron.mjs --target=mac-arm64   # or mac-x64 / win-x64 / linux-x64 / --all
```

CI: pushing a `v*.*.*` tag to `main` triggers `.github/workflows/release.yml`. The workflow:

1. Rewrites `package.json#version` from the pushed tag so the same number flows into Info.plist / NSIS metadata, into `NEXT_PUBLIC_APP_VERSION` for the in-app version chip, and into the asset filenames.
2. Builds each target on a native runner: macOS Apple Silicon and macOS Intel both run on `macos-latest`, the latter cross-building via `electron-builder --mac --x64` because GitHub's Intel runners (`macos-13`) are being deprecated and queue times are unreliable. Windows builds on `windows-latest`, and Linux x64 builds a portable `.AppImage` on `ubuntu-latest` (electron-builder bundles `appimagetool`; the macOS signing steps are gated to the mac targets, so Linux skips them). The Next standalone server loads no native modules of its own (it is pure JS); the two native engine binaries — Codex and Claude, including their static-musl Linux builds — are fetched per target by `electron-prepare.mjs`, and Gemini/Pi are platform-independent JS, so cross-arch is clean. One Next-side subtlety the Markdown importer forced: `pdfkit` reads its font-metric data from disk at runtime, so it is marked a `serverExternalPackage` (with its AFM data traced into the bundle) to keep `__dirname` pointing at `node_modules` instead of an inlined chunk.
3. Uploads each artefact to a workflow artifact.
4. A final `publish` job collects them and creates the GitHub Release tied to the tag.

macOS builds are **signed with a paid Apple Developer ID Application certificate and notarized by Apple** in the same `scripts/build-electron.mjs` invocation that produces the `.dmg`. The pipeline auto-detects what the host has:

- **`developer-id`** — a "Developer ID Application" identity is in the keychain *and* the App Store Connect API key trio is exported (`APPLE_API_KEY` path to the `.p8`, `APPLE_API_KEY_ID`, `APPLE_API_ISSUER`). `build-electron.mjs` adds `--config.mac.hardenedRuntime=true --config.mac.notarize=true` to electron-builder, which signs every Mach-O with the cert, ships the bundle to Apple's notary service, and staples the returned ticket onto the `.app` *inside* the `.dmg`. Gatekeeper opens the download with no prompt because the stapled ticket is consulted before the network is.
- **`developer-id-no-notary`** — cert is present but no notary credentials. We still sign (and keep Hardened Runtime on so the build is notarizable later) but skip the notary call. Useful for one-off local checks before secrets land in CI.
- **`ad-hoc`** — neither the cert nor the env-var pair is present (or `CSC_IDENTITY_AUTO_DISCOVERY=false` was set explicitly to force the legacy path). The `afterSign` hook (`scripts/electron-after-sign.cjs`) runs `codesign --force --deep --sign -` over the staged `.app` so the Apple Silicon kernel's mandatory-signature check still passes; without that step M-series Macs reject the bundle outright as "damaged" rather than

showing the bypassable Gatekeeper prompt. First launch needs a one-time System Settings → Privacy & Security → Open Anyway dance.

The `afterSign` hook reads `process.env.GETIT_MAC_SIGNING_MODE` to decide which branch ran upstream. In the two `developer-id*` modes the hook only re-verifies the existing signature — overwriting a Developer ID signature with an ad-hoc one would break notarization. In `ad-hoc` mode the hook performs the codesign pass.

CI plumbing lives in `.github/workflows/release.yml`. The macOS matrix jobs decode two secrets into runner-temp files (`MAC_DEVELOPER_ID_CERT_BASE64` → a one-shot keychain via `security create-keychain` + `security import` + `security set-key-partition-list`; `APPLE_API_KEY_BASE64` → a `.p8` at a path exported via `$GITHUB_ENV`) before `build-electron.mjs` runs, then the same detection picks the `developer-id` branch automatically. The keychain and key file are scoped to the runner and disappear with the VM.

Windows builds are not signed: SmartScreen reputation is per-certificate and the project doesn't currently pay for a Windows code-signing cert. The first launch on Windows still shows the SmartScreen prompt; click **More info → Run anyway**.

Auto-update

On boot, before the wizard, `electron/updater.js` calls the GitHub Releases API for `beltromatti/get-it`, semver-compares its tag against `app.getVersion()` (the value the CI step pinned), and picks the asset whose filename matches the running platform and arch. When a newer version exists, a polished `BrowserWindow` shows the release notes and an "Update now" button. Clicking it downloads the asset with a live progress bar, hands the file to `shell.openPath` (Finder mounts the `.dmg`, Windows runs the NSIS installer, Linux surfaces the `.AppImage` to the file manager), and quits so the installer can replace the app on disk.

The user's library, work-context journals, knowledge graphs and settings all live outside the app bundle, so an in-place install never touches them. Network failures, 404s when no release is published yet, and assets missing for the running platform all silently bypass; the rest of startup proceeds unaffected.

Origin and trajectory

Get It. was built in 24 hours at **GDG AI Hack 2026, Milan**, for the **Braynr** challenge. Hackathon team:

- Mattia Beltrami (Politecnico di Milano)
- Matteo Impieri (Politecnico di Milano)
- Filippo Difronzo (Politecnico di Milano)
- Luca Feggi (Università di Padova)

The hackathon submission lived at commit `277ec43` and contained the core architecture this writeup describes: the visualizer pipeline with all five renderer types, the knowledge-graph build agent, the four-axis evaluator, the chat / flashcards / Feynman tools, and the work-context journal. Two design decisions that look obvious in hindsight come straight from the time constraint:

- **`new Function` for the LLM-emitted JS** was the only sandbox we could plausibly ship in 24 hours. We documented it as a defense against LLM mistakes rather than adversarial input; the boundary has held up because the bring-your-own-account model means the user is running their own model calls against their own PDFs.
- **Filesystem-only persistence**. Spinning up a database under a hackathon clock would have eaten the time we needed for the evaluator. The JSON-files-under-a-data-dir layout was a deadline call. It then turned out to be the right call once we added the desktop shell: the same files are now what the auto-update flow preserves across version bumps, and the same files are what the user downloads in a click.

Everything beyond `277ec43` is post-hackathon polish that turned the demo into a shipping product. Roughly chronological:

- **Server-side jobs runner.** Detection and per-tag viz generation moved from renderer loops into singleton-per-doc jobs inside the Next process. The viewer became a poll-and-display consumer. Multi-doc parallel progress and reopen-where-you-left-off both fell out for free.
- **Persistent Library** with `lastOpenedAt`, tag-progress and KG-status badges that poll the same job source as the viewer.
- **Desktop shell**. Electron main, embedded server, free-port spawn, single-instance lock. The renderer is byte-identical to the hackathon Next app.

- **First-launch setup wizard.** Bundled Codex binary, OAuth sign-in capture, re-entry on `auth_lost`.
- **Auto-update.** GitHub Releases poll on boot, in-app installer flow, no data loss across version bumps.
- **Codex error classifier + health mailbox**. The four-category banner with retry-deadline countdown, plus the evaluator queue's automatic resume.
- **Quizzes tool** (v1.1.0). The fourth study surface, with `crypto.randomInt`-driven option shuffle so the agent's positional bias does not leak.
- **Cross-arch CI**. Both macOS targets now build on `macos-latest`; the Intel slice cross-compiles.
- **Bring-your-own-account messaging**. The Notice, the writeup section above, the in-app wizard copy: all aligned so the legal posture and the product positioning are the same sentence.
- **Long-document support** (v1.2.0). The push that makes a 100-page PDF usable without exploding the user's Codex usage, at unchanged output quality. A model-free upload-quality gate rejects scanned / image-dominant / over-long files before any agent runs. Concept detection batches up to five pages per call. Chat moved onto a native Codex thread, so the document is sent once and follow-ups only carry the new message. KG evaluation became incremental (baseline scores plus only the new interactions) and chat now batches a single pass per visit instead of one per reply. Per-call character caps were removed across every prompt so the agents reason over whole sections rather than truncated fragments, and visualization generation defaults to lazy/on-click.
- **Signed and notarized macOS** (v1.2.1). Developer ID signing plus Apple notarization end to end, so a fresh download opens with no Gatekeeper prompt (see *Desktop packaging*).
- **Reliability and reach** (v1.3.0). The model is now pinned explicitly (`gpt-5.5`) so a retired binary default can't 400 users out of every generative feature, with a dedicated `model_unsupported` banner. Every background job stops cleanly on a Codex error and offers a manual retry instead of auto-looping — the fix for a rate-limit retry loop that could re-fire the banner endlessly. **Linux x64** joined macOS and Windows as a first-class AppImage target. An anonymous open/update ping powers real Total / Daily / Weekly / Monthly user counts on the marketing dashboard, cleanly separating genuine installs from in-app updates.
- **Multi-engine, and polish** (this release, v1.4.0). The single Codex transport became a four-engine **provider router** — OpenAI Codex, Anthropic Claude, Google Gemini, and a bring-your-own / local-Ollama path — behind one `AIProvider` interface, with per-engine bundling, per-tier model/effort defaults, a normalised usage store, provider-tagged chat threads that migrate on a mid-session switch, and a universal call-timeout backstop so no tool on any engine can hang. The first-launch wizard became engine-aware (OAuth or API-key, gated on a live verification), and the Account/Settings panels follow suit. **Markdown (`.md`) import** renders to a clean PDF and rejoins the normal pipeline. A **light/dark/system theme** (default light) themes the whole UI and the native window chrome. Subprocess lifecycle was hardened end-to-end (process-group kill + watchdog, no orphans), and the viz sandbox's constructor guard was made correct across all function kinds.
- **Open source, in the open.** The project is Apache-2.0 and actively seeks contributors; a `CONTRIBUTING.md` lays out the vision and a Discord community is where the work is coordinated.

The hackathon clock is no longer a load-bearing constraint, but the product it forced us into has not moved.

What's not here yet

A few choices are deliberately deferred.

Vocal Feynman. The same agent loop and the same end-of-session summary, with a streaming TTS layer over the child voice. The text variant ships today because typed transcripts are strictly better evaluator inputs (no transcription error, no per-token cost, full searchability). The data shape does not change.

A hosted multi-user backend. Out of scope by design. Get It. runs locally against the user's own AI account, against their own PDFs, on their own machine. The Braynr policy band (source-grounded only, local-first, tiered access) we get for free at this scale.

Windows code signing. Windows builds are still unsigned: SmartScreen reputation is per-certificate and Microsoft's path to a Gatekeeper-equivalent zero-warning download (Azure Trusted Signing) requires a paid Azure subscription the project doesn't carry. The macOS notarization story is already in place (see *Desktop packaging* above); Windows is the remaining funding decision, not a missing piece of the architecture.

Notice and license

Get It. is an independent project. It is not affiliated with, endorsed by, sponsored by, or otherwise associated with OpenAI, Anthropic, or Google. The app talks to each model only through that vendor's own official CLI — Codex CLI, Claude Code, the Gemini CLI, or any OpenAI-compatible endpoint the user configures — signed in with the end user's own account or API key. "OpenAI", "ChatGPT", "Codex", "Anthropic", "Claude", "Google", and "Gemini" are trademarks of their respective owners; we use the names only to describe what Get It. interoperates with.

Your use of any model through Get It. is subject to that provider's own terms of use, usage policies, and privacy policy, and to the license of the CLI it ships through. Those documents are authoritative for what each service permits and how data is handled on the provider's side.