SNUG PROTOCOL

The Snug Protocol: An Open Protocol for Agent-Backed Personal Software

A specification of the wire protocol, security model, portable data format, and connected-application surface for sandboxed micro-applications that use a host's language model as a runtime capability, reach the user's own services under human-frozen ceilings — and that their users own outright, as a single file.

AUTHOR

Jeetu Maker

SPECIFICATION · EDITION

1.0 · edition 3

DATE

August 2026

Abstract. Large language models have collapsed the cost of authoring software to the point where an application built for a single person is economically trivial. What has not existed is the structure that makes such software safe to run, durable enough to keep, and portable enough to own: an isolation model, a data format, a credential doctrine, and a wire contract that any vendor can implement. This paper specifies the Snug Protocol, which supplies that structure in three coupled layers. The first is a versioned postMessage frame protocol between a sandboxed application and its host, together with a tagged chat envelope by which the host relays an application's request to its own agent endpoint; this makes the host's model a runtime capability the application invokes rather than a code generator that produced it. The second is a portable user database — one SQLite file holding an individual's applications, their versions, their data as native tables, their conversations, their connections, and their settings — which travels intact between hosts, model providers, and devices, and which the user may seal under passphrase encryption they alone hold. The third is a connected-application surface under a single doctrine — the model proposes, the human approves, the host enforces — in which an application reaches the user's own services only through a host-side executor whose host ceiling is frozen at the moment of human approval, and in which even a device that has no API at all is reachable through a local helper that is governed as a capability rather than trusted as a host. We present the protocol's normative rules, an explicit threat model in which application code is treated as hostile at all times, and the hard constraints from which the security properties follow. We situate the design against the Model Context Protocol, vendor application platforms, and the local-first literature, and state its present limitations candidly.

Contents

1 Introduction 3
The market of one, and what it still lacks

2 Design goals and non-goals 4
What the protocol commits to, and what it deliberately refuses

3 Architecture: the three actors 5
Model provider, host provider, and the user who owns the file

4 The wire protocol 7
Thirteen frames, seven rules, and the chat envelope

5 Security model 12
Threat model, hard constraints, and residual risk

6 The portable user database 15
One file: hub tables, native per-app data, naming, encryption

7 Connected applications 20
Requirements, grants, frozen ceilings, and the executor

8 Linked-device connections 24
Governing a provider that has no API: the helper surface

9 Runtime contracts and the data surface 26
Turns cheap enough for any brain; data that answers to its owner

10 Conformance 27
What an implementation must do to claim the name

11 Related work 29
MCP, vendor application platforms, local-first software

12 Limitations and future work 30
What is unsolved, stated plainly

13 Conclusion 31

— Normative references 31

A Appendix: message schema summary 32

1 Introduction

For most of the history of commercial software, an application existed because someone judged that enough people wanted it. That judgement was an economic filter, and it was ruthless: if your need was too particular — too specific to your household, your condition, your one strange workflow — no product was built, because the addressable market was one person.

Language models dissolved that filter. The marginal cost of producing a small, working application has fallen far enough that building one for a single user is now unremarkable. Anyone who has asked an assistant for a bespoke tracker, a calculator shaped to their own rules, or a game invented by their child has produced software that no company would ever have shipped.

What did not arrive with that capability is the surrounding structure. Generated applications land as artifacts inside a vendor's product: they run under that vendor's rules, their data lives in that vendor's storage, and they are portable only in the sense that text is portable. The moment such an application needs to persist state, hold a credential, or outlive the conversation that produced it, four questions become urgent, and none of them is a modelling question:

- **Isolation.** The code was written by a language model, possibly under the influence of text the user did not author. On what terms may it run?
- **Custody.** Where does its data live, who can read it, and what happens to it when the user leaves?
- **Continuity.** If the application is genuinely the user's, it must survive a change of host, a change of model provider, and the disappearance of either.
- **Connection.** The application is most useful when it can reach the user's own services — their music, their lights, their finances, even their messages. On whose authority, within what wall, and with credentials held by whom?

These are protocol questions. They are answered once, in public, by a contract that multiple implementations can satisfy — or they are answered privately and differently by every vendor, which is the same as not being answered at all.

The Snug Protocol is a proposed answer. It rests on one architectural observation: an application does not need to *contain* intelligence if it can *call* it. A chess application built on Snug ships no chess engine. It sends the board position across a boundary, and the host's agent replies with a move. The application supplies the interface, the rules, and the state; the model supplies judgement, on demand, as a runtime capability. We refer to this throughout as the body/mind split, and it is a relationship at runtime, not a fact about how the code was produced.

That single decision reorganises everything else. If the model lives outside the application, the application needs no network access, and can therefore be sandboxed far more tightly than a conventional web page. If it needs no network access, it cannot exfiltrate a credential, so credentials can be kept strictly on the host side of the boundary — which is also what makes it safe to let the application *request* the network as a mediated capability, one whose every reachable host a human approved and the host froze. And if the application is a small self-contained program with its data held separately, then the application and its data can be written into a single portable file that the user carries between hosts.

The protocol has three layers, specified in this paper. Section 4 covers the wire protocol: thirteen versioned postMessage frames and a tagged chat envelope. Section 6 covers the portable user database, a storage and host-behaviour layer that leaves the wire protocol untouched — including the file's naming rules and its optional passphrase protection. Sections 7 through 9 cover the connected-application surface: how an application declares what it needs, how a human grant freezes what it may reach, how the one host-side executor enforces that wall on every request, and how the same governance extends to a provider that has no API at all. Section 5, which sits before them, is in our view the load-bearing part: the threat model and the hard constraints that every other layer exists to preserve.

> **CONVENTIONS**
>
> The key words **MUST**, **MUST NOT**, **SHOULD**, and **MAY** are used in the sense of RFC 2119. The wire protocol's core is normative and published since spec v0.1; the storage layer, the connected-application surface, runtime contracts, and linked-device connections are normative as of specification 1.0. One surface — standing approvals — is explicitly PROVISIONAL and so marked where it appears. Where this paper and the specification differ, the specification governs.

2 Design goals and non-goals

A protocol is defined as much by what it declines to do as by what it specifies. We state both here so that later sections can be read as consequences rather than choices.

2.1 Goals

G1 The agent is a capability, not a dependency

An application MAY call the host's model at runtime; it MUST NOT be required to. An arcade game whose rules are entirely deterministic should compute locally, pay no latency, and run with no model configured at all. The host advertises what it can do and degrades gracefully when an application asks for nothing.

G2 Hostile-by-default application code

Application code is authored by a language model, which may have been influenced by content the user did not write. The protocol therefore assumes the application is hostile and derives its guarantees from isolation, not from the good behaviour of generated code.

G3 The user owns the artifact

Everything an individual accumulates — applications, versions, data, conversations, connections, settings — is one file in a standard, inspectable format they can export at any time and open with ordinary tools, and may protect with a passphrase only they hold.

G4 No required backend

A conforming host is deliverable as static files. A hosted service may add convenience — synchronisation, a subscription path to a model — but application execution MUST NOT depend on it.

G5 Additive evolution

Unknown message types and unknown fields are ignored rather than rejected, so that a newer participant can talk to an older one without negotiation beyond a version check. The protocol grew from nine

frames to thirteen without breaking a single deployed application, which is this rule doing its job.

G6 **Implementable by more than its author**

Every published message is JSON Schema, exported byte-identically from the reference implementation. The schemas, not the prose, are the contract.

G7 **The model proposes; the human approves; the host enforces**

Anything that widens what an application can reach — a provider, a host, a scope — is proposed by a model or a manifest, reviewed verbatim by a human, and then *frozen* by the host at the moment of approval. Enforcement is structural and unconditional: there is no strictness flag anywhere in the protocol, because a security property that can be disabled by configuration will be disabled in some deployment, and that deployment is the one that gets attacked.

2.2 Non-goals

- **A marketplace.** The protocol says nothing about discovery, distribution, ratings, or payment. It specifies how an application talks to a host, and what a user's file contains.
- **A general plugin system for agents.** Extending what a model can *do* — tools, resources, external systems — is well served by the Model Context Protocol, and Snug does not duplicate it. Snug addresses the inverse direction: giving a user-authored application access to the model, and governing what that application may reach. Section 11 develops the distinction.
- **A rendering or component standard.** An application is an ordinary self-contained HTML document. The protocol constrains what crosses the boundary, not what the application draws.
- **Cryptographic custody.** The protocol does not claim that a host is incapable of reading a user's secrets — only that a conforming host never receives them. The optional at-rest encryption of §6.6 changes where the file is readable, not who holds custody; the distinction is drawn precisely in §5.4, and we regard overstating it as a security defect in its own right.

3 Architecture: the three actors

Snug models three parties, deliberately kept separable, as shown in Figure 1. The separation is the portability story: any one of them can be replaced without disturbing the other two.

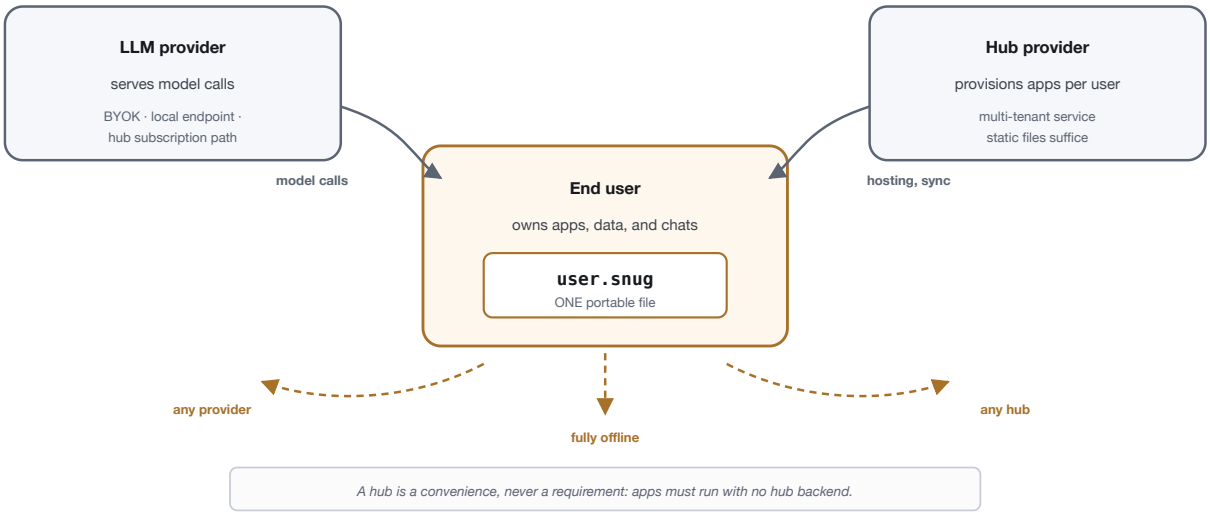

Figure 1. The three actors. The user's file is the centre of gravity: it holds everything the user has accumulated and can move to any model provider, any host, or none. A host provider is a convenience — the protocol requires that applications run without one.

The **model provider** serves inference. It is reached in one of three ways — directly from the browser with a key the user supplies, directly from the browser against a local endpoint on the user's own machine, or through a host's subscription path — and §6.5 treats all three as a single settings choice rather than a migration.

The **host provider** operates the surface where applications are built and run. It provisions applications per user and may offer synchronisation, caching, and a subscription path to a model. What it may not do is make itself necessary: a conforming host delivers application execution from static assets, and its server components, where they exist, are strictly optional. A host may be a browser page, or a desktop shell wrapping one — the reference implementation ships both, and the desktop host is what makes devices on the user's own network, and local helper processes, reachable at all.

The **end user** owns the artifact. Not an account, not rows in a vendor's database — one file, in a format with a thirty-year record of stability and universal tool support.

Figure 2 shows how these actors are realised inside a running host. Three boundaries in that diagram carry the security properties, and all three are examined in §5: the seam between the host page and the sandboxed frame; the point at which an application-originated request is validated before it can have any effect; and the one seat — the connected-fetch executor — through which every credentialed call to the outside world must pass.

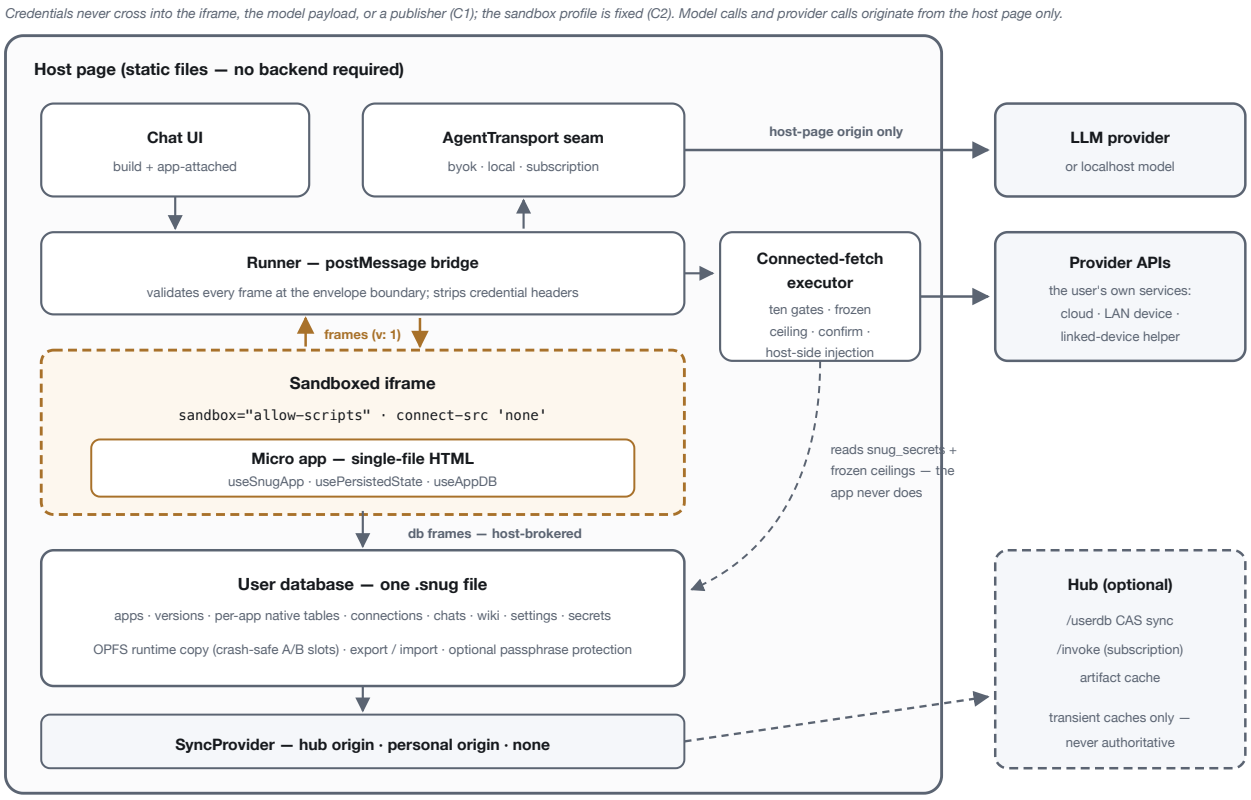

Figure 2. System architecture. The application is confined to a sandboxed frame with no network egress; every capability it has — model access, storage, connectivity — is mediated by the host through versioned frames. Model calls and provider calls originate from the host page only. The user database is authoritative in every mode; host-side stores are transient caches.

3.1 The runtime relationship

The consequence of G1 and G2 together is that the application is a body and the agent is its mind, but the body is fully capable on its own. Two applications illustrate the poles. A chess opponent calls the model on every move: the judgement it needs — what is a good move — is not expressible as a rule the application could apply itself. An arcade game calls the model never: collision detection, scoring, and win conditions are deterministic, and routing them through a model would add latency, cost, and a dependency on a configured key to a program that should run offline.

Both are conforming Snug applications. Deciding which turns need the model is an authoring decision, and the protocol records nothing about it — the property is emergent from the application's code, and the host already advertises its capabilities and degrades when they go unused.

4 The wire protocol

NORMATIVE · CORE SINCE V0.1

The wire protocol comprises two coupled contracts. *Frames* are postMessage messages exchanged between the application frame and the host runner. The *chat envelope* is the tagged message a host sends to its own agent endpoint on behalf of an application, together with the reply contract the agent must satisfy.

4.1 Frames

Thirteen frame types are defined. Each carries `v: 1`: the wire protocol has been additively extended, never broken, since v0.1. All thirteen are published as JSON Schemas exported byte-identically from the reference implementation — the nine core frames since v0.1, the net and open-url pairs since spec 1.0's consolidation.

TYPE	DIRECTION	PURPOSE
`snug:app-announce`	app → host	Self-describing metadata on mount: `appId`, `displayName`, `description`, `iconEmoji`, `iconColor`. The host acknowledges with `snug:host-ready`.
`snug:host-ready`	host → app	Sent on frame load and again as an announce acknowledgement (idempotent): `instanceId`, `protocolVersions`, `capabilities` (`streaming`, `db`, `auth`; optional additive `net` and `openUrl`), `theme`, optional `locale`.
`snug:app-message`	app → host	A request for the agent: `requestId`, `instanceId`, `appId`, `action`, and optional structured `payload`, `state`, and `responseSchema`.
`snug:app-cancel`	app → host	Abort an in-flight `requestId`.
`snug:app-response`	host → app	Streaming, final, or error, per R3. Three shapes: cumulative `text`; a final `data` payload; or an `error`.
`snug:db-request`	app → host	Host-brokered per-application storage. Operations: `exec`, `export`, `import`, `kvGet`, `kvSet`.
`snug:db-response`	host → app	Result rows and columns, an exported byte payload, or an error.
`snug:net-request`	app → host	The governed network capability (§7): `url`, `method`, optional `headers` and `body`. Strict: a body on GET/HEAD, or any credential-shaped header, rejects the whole frame. Carries **no** `appId` — the network binding is host-assigned.
`snug:net-response`	host → app	`status`, whitelist-filtered `headers`, scrubbed `body`, optional `truncated` — or an envelope error.
`snug:open-url-request`	app → host	A request that the *host* open an https URL in the user's real browser, after its own confirm dialog showing the full URL, on a user gesture. No target, no window features, no navigation primitive; userinfo-bearing URLs are refused at the schema.
`snug:open-url-result`	host → app	`opened`, `declined`, or `refused` (with an optional reason).
`snug:host-event`	host → app	Open additive channel: `theme-change`, `visibility`, `connection-event`, and further namespaced events. Unknown events are ignored. Subject to R7: hints, never content.
`snug:app-event`	app → host	The reverse channel, e.g. `resize` carrying a height.

Table 1. The thirteen frame types of specification 1.0. Published schemas are input shapes: validators MUST accept unknown fields (R2). The net and open-url frames are the stated exception — strict, because their fields become real-world effects.

Storage is brokered rather than direct for a reason that recurs throughout the design. An application frame runs with a null origin (§5.3), and browser storage APIs are unavailable to it. One of the production systems that preceded this protocol taught applications to use `localStorage`, which worked only because its sand-

box was weakened enough to permit it — precisely the weakening this specification forbids. Routing storage through frames preserves the sandbox and gives the host a mediation point. The network capability follows the identical logic one layer up: the frame's content security policy blocks every egress path, so the only network an application can have is one the host performs on its behalf — which is what makes §7's governance enforceable rather than advisory.

4.2 The application lifecycle

Figure 3 traces a complete request, including the two paths by which a request may end without a normal answer.

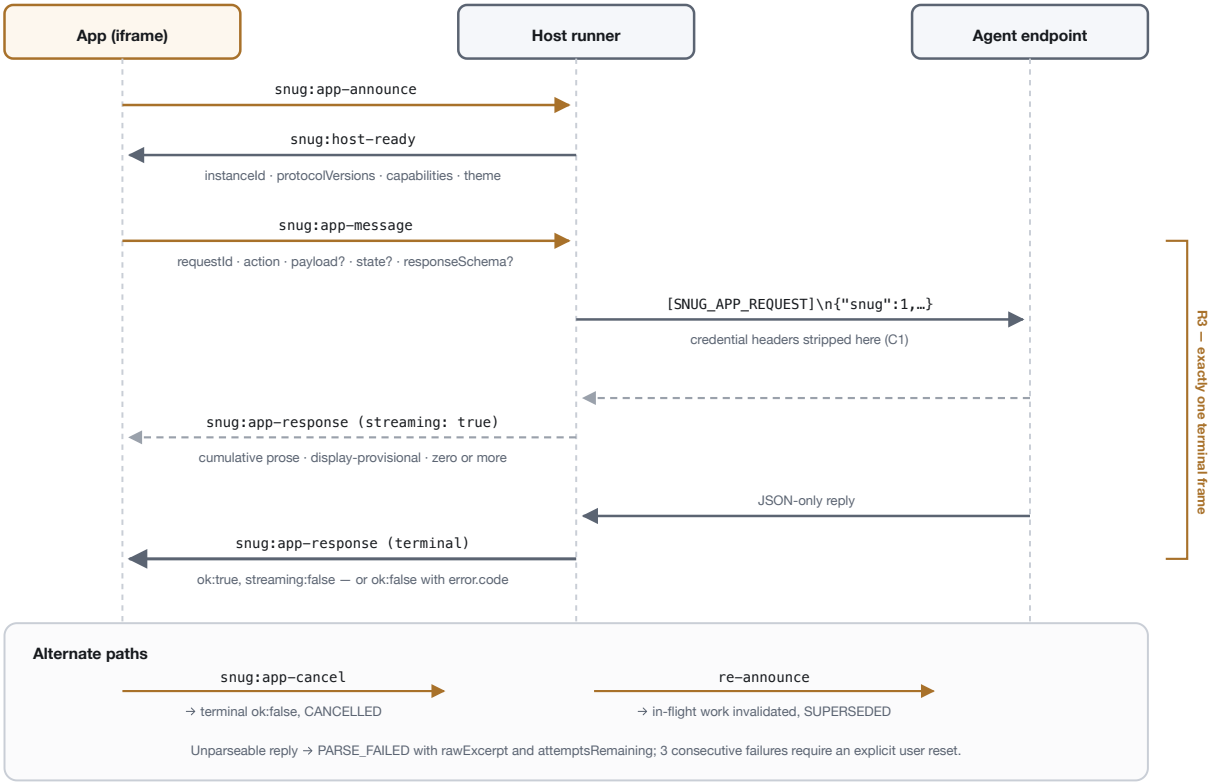

Figure 3. The frame lifecycle. The application announces itself and receives an instance identity; requests then flow outward through the host, which strips credential headers before relaying an envelope to the agent endpoint. Streaming frames are provisional; exactly one terminal frame is authoritative (R3).

4.3 Normative rules

Seven rules constitute the normative core. We give each rule and, briefly, the failure it exists to prevent — since a rule whose motivation is undocumented tends to be implemented approximately.

R1 Versioning

Every frame carries `v: 1`; the chat envelope carries `snug: 1`. Unsupported versions are rejected with `UNSUPPORTED_VERSION`. Parse failures surface a `requestId` recovered from the raw frame when it carried a plausible string identifier; hosts answer `UNSUPPORTED_VERSION` or `MALFORMED` on the wire only in that case, and never otherwise. Answering an unidentifiable frame would mean emitting a re-

ply that no request can be correlated with — an invitation to a spurious response being attributed to an unrelated request. `snug:host-ready.protocolVersions` advertises what the host supports.

R2 Additivity

A frame with a valid `v` but an unrecognised `snug:*` type MUST be silently ignored, and unknown fields on known frames MUST be ignored. The `snug:` type prefix and the event namespaces are reserved. This is what allows a host and an SDK to be upgraded independently. The strict net and open-url frames are the deliberate exception: their fields become network requests and browser navigations, so an unknown key there is a rejection rather than a passthrough.

R3 Terminal frame

Every accepted `requestId` receives exactly one terminal `snug:app-response`, either successful and non-streaming or an error. `streaming: true` frames are cumulative prose and are display-provisional; the terminal frame is authoritative. Hosts MAY suppress streaming for schema-constrained requests. Without this rule an application cannot know when to release a pending interaction, and a dropped final frame is indistinguishable from a slow one.

R4 Identity

Hosts route by `event.source`. A sandboxed frame has a null origin, so `targetOrigin` is necessarily `'*'` and origin comparison is unavailable as an identity mechanism — the message source object is the identity. The host mints `instanceId` and delivers it in `snug:host-ready`; applications echo it in every request. A fresh `snug:app-announce` from the same frame invalidates in-flight work with `SUPERSEDED`. `appId` is display metadata and **not a security principal**: it is self-asserted by the application and MUST NOT be used to make an access decision. `requestId` MUST be unique per instance.

R5 Error codes

`error.code` is an open string. The known codes are `PARSE_FAILED`, `THREAD_CONFLICT`, `NETWORK_ERROR`, `RESET_FAILED`, `CANCELLED`, `SUPERSEDED`, `UNSUPPORTED_VERSION`, `CONSENT_REQUIRED` (reserved), `AUTH_REQUIRED` (reserved), and `HOST_ERROR`. `MALFORMED` is defined by R1 as a wire answer to an unparseable frame and is not part of this list. The network capability carries its own registry of `NET_`-prefixed codes — approval, ceiling, SSRF, confirmation, redirect, and size refusals each have a distinct name, because an application that cannot tell "the user said no" from "the host is broken" will retry the wrong one. Receivers treat unknown codes according to the `retryable` flag and render them as `HOST_ERROR`.

R6 Limits

Frames are capped at 256 KiB, with two larger size classes: `db-request` and `db-response` at 8 MiB, so that a base64-encoded 5 MiB artifact round-trips through the storage bridge; and `net-request` / `net-response` at 1 MiB + 64 KiB — a 1 MiB response body plus envelope margin, with the response read capped *while reading* and an overflow becoming a terminal size error rather than a silent drop. Artifacts are capped at 5 MiB, `rawExcerpt` at 200 chars, and announce strings at displayName 80 and description 400. The parse-failure budget is 3 consecutive failures, after which the host requires an explicit user reset. Thread-conflict backoff is 100 / 250 / 500 ms.

R7 Push hints

A host-initiated push (`snug:host-event`) carries **references, never content** — a doorbell, not a delivery. The application answers a hint with its own governed reads, and the host rebuilds every field of a hint before forwarding it. Two frame-layer facts force this shape: host-event frames ride the ordinary

256 KiB class and an oversized frame is dropped *silently*, and host-event frames carry no `instanceId`, so a stale sender is indistinguishable from a live one. With hints, a stale or dropped event costs one redundant refetch and can never inject state.

> **WHY A PARSE-FAILURE BUDGET**
>
> A model that returns malformed output once will often do so repeatedly for the same input. Unbounded retries turn a formatting failure into a billing incident and a denial of service against the user's own quota. Three strikes, then a human decision.

4.4 The chat envelope

When an application requests the agent, the host does not expose the model endpoint to it. The host constructs an envelope and sends it to its own agent endpoint, using the same path its ordinary chat traffic takes. The wire form is a tag line followed by a JSON object:

```
// The host sends this to its own agent endpoint — never the application.
[SNUG_APP_REQUEST]
{
  "snug": 1,
  "appId": "portfolio-tracker",
  "instanceId": "c0a8-…",     // host-minted; the routing identity
  "requestId": "9f1c-…",      // unique per instance
  "action": "suggest_rebalance",
  "payload":   { /* structured, application-defined */ },
  "state":     { /* the application's own view of context */ },
  "responseSchema": { /* optional: constrain the reply shape */ }
}
```

Detection requires both the tag prefix and the `snug: 1` marker. Requiring two independent signals prevents ordinary chat content that happens to quote the tag from being processed as an application request.

Servers SHOULD skip thread history for application requests: the envelope is self-contained by way of `state`, and replaying an unrelated conversation into an application's turn both costs tokens and leaks the user's chat into a context the application can observe through the reply. Servers MUST apply the credential-header strip described in §5.2.

4.5 The agent reply contract

The agent replies with only a JSON object; a human-readable `message` field is recommended. Because models emit fenced code blocks and surrounding commentary with some regularity, hosts parse with graduated tolerance — a raw parse first, then a fenced block, then balanced-object extraction — and reject null, array, and scalar replies. A failure at every stage becomes a `PARSE_FAILED` frame carrying `rawExcerpt` (capped at 200 chars) and `attemptsRemaining`, so the application can surface something honest rather than hanging.

Tolerant parsing here is not indiscipline. The alternative — failing the turn on a stray code fence — converts a cosmetic model artifact into a broken application, and every implementation would end up writing the same three fallbacks privately anyway. Specifying them makes behaviour uniform across hosts.

5 Security model NORMATIVE

This section states what the protocol defends against, the constraints from which its guarantees derive, and — with equal prominence — what it does not defend against. The reference implementation maintains a public threat model organised the same way, in which every enforced invariant names the file that enforces it and the test that fails if it regresses; a claim that cannot name both is listed as a residual instead.

5.1 Threat model

The first adversary is **the application itself**, and Figure 4 shows exactly where that adversary is confined. This is not a hypothetical posture. Application code is authored by a language model from a prompt that may incorporate untrusted text: a web page the model consulted, a document the user pasted, a message from a third party. Prompt injection is a live technique with no complete defence at the model layer, so the protocol assumes injection sometimes succeeds and constrains what a successful injection can accomplish. The connected-application surface adds three more adversaries with real capabilities: a hostile or compromised *provider* on a host the user approved, a hostile *connection requirement* authored by a steered model, and a hostile *file* arriving through import or sync.

THREAT	MECHANISM	RESIDUAL
Application exfiltrates a credential	Credentials never enter the frame; `connect-src 'none'` removes every egress path the application could use; the governed network path strips credential-shaped headers at the schema and injects host-side only	None by construction, provided C1 and C2 hold
Application reads another application's data	Runtime isolation is physical: an application executes against a materialised database containing only its own objects (§6.3)	Host implementation defect
Application reads host-namespace tables	Reserved prefixes are refused; unvalidated names are never interpolated, and a violation fails the write closed	Host implementation defect
Application escalates by asserting another identity	Routing is by message source, not by any self-asserted field; `appId` is explicitly not a principal (R4)	None at the protocol layer
Application reaches a host nobody approved	The frozen ceiling (§7): exact-hostname membership computed at approval, checked on every request; SSRF literals refused even inside a ceiling; redirects never followed	Any path on an approved host is reachable (§5.4)
Injected instructions cause a privileged call	The application has no privileged call available: capabilities are host-mediated, credential injection is host-bound and always strict, and every mutating call stops at a human confirmation naming host, method, and URL	Model-layer manipulation of answer *content* (§5.4)

THREAT	MECHANISM	RESIDUAL
A hostile requirement impersonates a known provider	Registry-borrow ban on two triggers (name and host-intersection); provider names confined to printable ASCII; walkthroughs rendered as plain text; verbatim human review of every seat (§7.2)	Pure-ASCII lookalike names (§5.4)
A hostile imported file carries grants or contracts	Imported connection rows demote to declared and cannot serve traffic; imported runtime contracts are dropped unless byte-identical to known ones; endpoint settings require re-confirmation	Import can still carry data a user then trusts (§5.4)
Application exhausts the user's model quota	Parse-failure budget of 3; cancellation; host-side rate control	An application may still make many valid requests
Malformed frames drive the host into an inconsistent state	Schema validation at the boundary; exactly one terminal frame per accepted request (R3); bounded backoff	Host implementation defect

Table 2. Threats and the mechanism that answers each. Every mechanism is structural; none depends on the application behaving correctly.

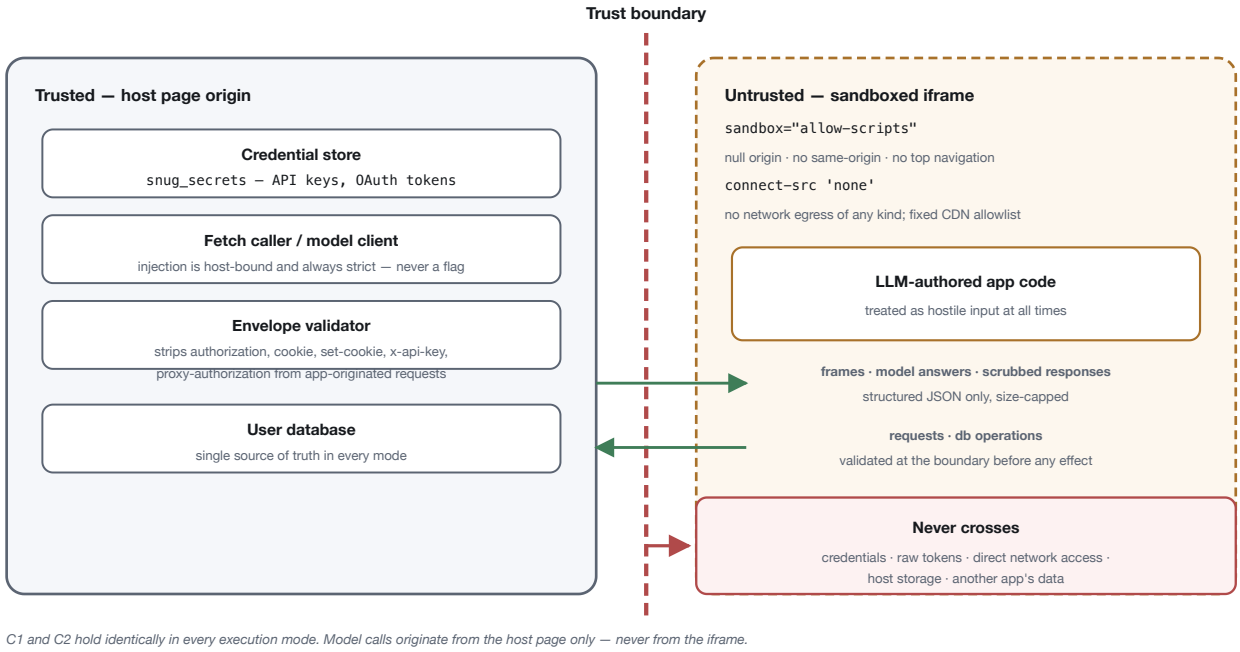

Figure 4. The trust boundary. Structured, size-capped JSON crosses in both directions and is validated before it can have any effect. Credentials, raw tokens, direct network access, host storage, and other applications' data never cross at all.

5.2 C1 — the token boundary

The first hard constraint: **credentials never enter the application frame, never reach the model, and never reach a publisher.**

Concretely, hosts MUST strip `authorization`, `cookie`, `set-cookie`, `x-api-key`, and `proxy-authorization` from any application-originated request at the envelope boundary — and the net frame schema refuses those header names outright, so a credential-carrying request is malformed before any handler runs. Where the host injects a credential on an application's behalf (§7), injection is host-bound and **always strict — never a configurable flag**.

The emphasis is earned. An audit of the two production systems this protocol generalises from found a strictness control that defaulted to off, which is to say a documented feature that silently converted a prompt injection into credential exfiltration. A security property that can be disabled by configuration is a security property that will be disabled in some deployment, and the deployment where it is disabled is the one that gets attacked. The protocol therefore does not offer the choice.

5.3 C2 — sandbox integrity

The second hard constraint: **application frames run with `sandbox="allow-scripts"` only, with `connect-src` blocked.** These are never weakened, including for demonstrations. Hosts that permit any script source at all SHOULD confine it to a fixed allowlist, as the reference implementation does; the allowlist itself is a host policy rather than a protocol requirement.

Three properties follow. First, the frame is assigned a **null origin**: `allow-same-origin` is absent, so the application cannot reach host storage, host cookies, or the host DOM — and, as noted in R4, this is also why identity must be established by message source. Second, `connect-src 'none'` removes fetch, XHR, WebSocket, and EventSource, so there is no channel by which data could leave even if the application obtained something worth sending. Third, because ordinary storage is unavailable to a null origin, persistence must be brokered by the host, which is what makes per-application isolation enforceable rather than advisory.

> **A NOTE ON DEMONSTRATION BUILDS**
>
> The most common way a sandbox weakens is a temporary exception that is never removed. The reference implementation asserts the sandbox profile in its test suite, workspace-wide: a frame carrying `allow-same-origin` anywhere in the tree fails the build. We recommend the practice to any implementer — a constraint that is only documented is a constraint that erodes.

5.4 Residual risks

We state these plainly, because a threat model that lists only solved problems is marketing.

Secrets are at rest in the user's file — by default in plaintext, optionally sealed. Credentials live in the `snug_secrets` table inside the user's own file. They are stripped from host-bound synchronisation and from default exports, and the file is vacuumed so that freed pages retain nothing recoverable. They are nonetheless present in the local file — a deliberate **trade-off** in favour of portability: cross-device continuity means the credentials travel with the file to storage the user chooses. The user may additionally protect the whole file with a passphrase (§6.6), which changes what an attacker holding the *bytes* holds — but the claim that supports is exactly "*the file can be encrypted with a passphrase only the user holds*", and no more. It is not zero-knowledge, it is not end-to-end encryption, and it defends neither a running host page that has already unlocked the file nor a machine on which the passphrase is typed under observation. No key-management or host-blind claim is made, and none should be made until a key-provider layer ships.

The honest formulation, and the only one we endorse: *your keys never reach our servers; your file, including keys, goes only to storage you choose — and you may seal it with a passphrase only you hold.* That is a statement about host custody plus at-rest encryption. It is not the stronger claim that keys never leave the local file, which would be false the moment a user connects a personal synchronisation origin — a legitimate and intended use.

The model can be manipulated in what it says. C1 and C2 bound what an injected instruction can *do*: it cannot reach a credential, open a socket, or touch another application's data — and on the connected surface it cannot widen a ceiling or bypass a confirmation. It can still influence the *content* of an answer the application then renders, and it can propose actions in the hope a hurried user approves them. Applications that act on model output should treat that output as untrusted input; hosts should prefer `responseSchema` so that replies are shape-constrained; and the confirmation dialog naming host, method, and URL is, by design, the wall that holds when everything upstream of it has been steered.

The scrub is honest about its class. Response scrubbing matches the values the host injected, raw and percent-encoded. A provider that re-encodes a credential — base64, hex, split across fields — defeats it by design; the frozen ceiling, which bounds who can receive the secret at all, was always the primary wall. The same honesty applies to the pseudonymisation backstop of §8: it is anti-default and anti-naive, not anti-adversarial, and the specification requires that statement to travel with the feature.

Any path on an approved host is reachable. The ceiling is host-granular: it does not bound ports, paths, methods, or query strings. A debug or echo endpoint on an approved host is reachable, which is precisely the precondition the re-encoding residual above depends on. Approving a host is approving the host.

Pure-ASCII lookalike provider names are accepted. The confusable guard stops homoglyph and registry-evasion attacks; it cannot reject `5potify` without rejecting legitimate names. Lookalikes are carried by the borrow ban's host-intersection trigger and by provenance disclosure in review, and a conforming host must not present the charset guard as more than it is.

Host implementation defects remain possible. Several mechanisms above depend on the host implementing them correctly: name validation, materialisation, header stripping, gate ordering. The protocol can specify these and the schemas can constrain the wire, but neither can compel a host to be correct. This is the ordinary condition of protocols, and it is why §10 states conformance in terms that are testable — and why the reference implementation's threat model pins every claim to a named test.

Availability is not addressed. A misbehaving application can make many individually valid requests. Rate control is left to hosts, which are better placed to know their own cost model.

6 The portable user database

> **SCOPE**
>
> This section describes the storage and host-behaviour layer of specification 1.0; the wire protocol of §4 is unchanged by it. Storage schema version 6 is described; the source of truth for constants and DDL is the reference implementation, locked by a snapshot test.

If applications are the user's, their data must be too — and in a form that outlives any particular host. Snug's answer is a single SQLite file per user, `user.snug`, which is the canonical artifact.

`PRAGMA user_version` carries the schema version — currently 6, migrated forward-only, with a file stamped newer than the implementation understands refused rather than overwritten; total size is bounded by `MAX_USERDB_BYTES`, 64 MiB. Figure 5 shows what that file contains.

SQLite is chosen for unglamorous reasons: it is a stable, universally supported, single-file format with three decades of tooling. A user who exports their file can open it in any SQLite browser and read their own data without our software. We consider that property, rather than any feature, the substance of ownership.

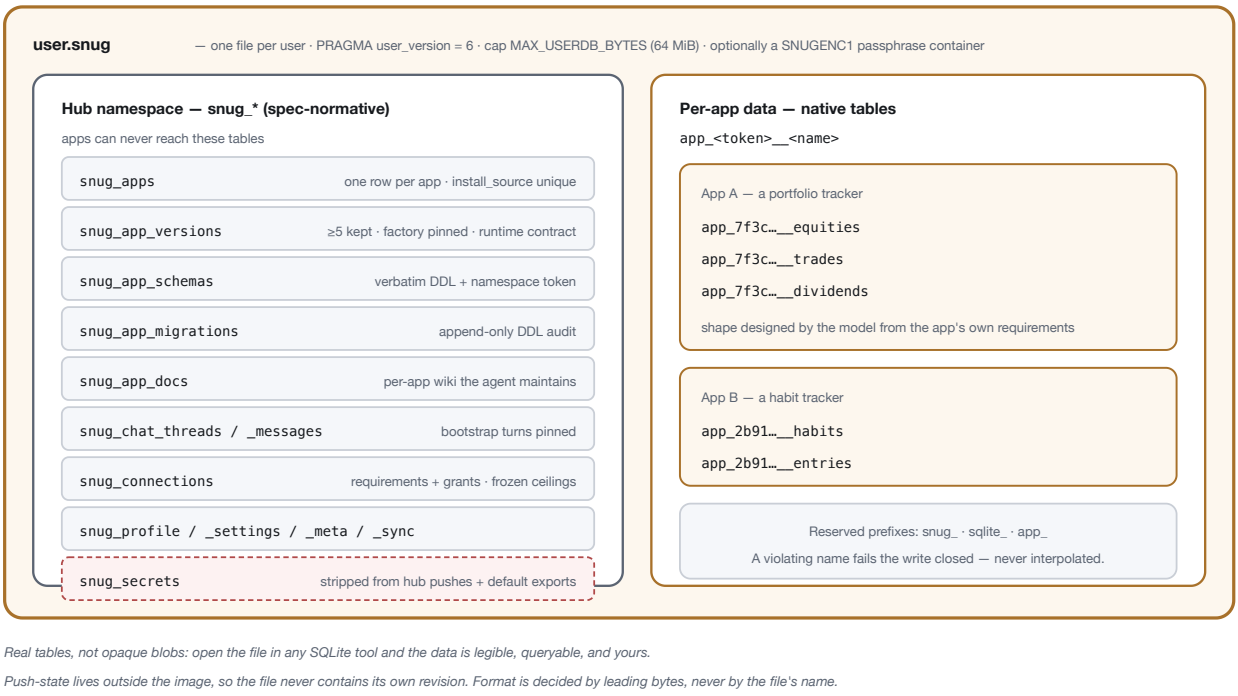

Figure 5. The user database. Host-namespace tables carry applications, their retained versions, their schemas and migration history, their documentation, their connections with frozen ceilings, and the user's conversations and settings. Application data sits alongside as real, legible tables under an injective namespace token.

6.1 Host-namespace tables

Tables prefixed `snug_` are the host namespace and are unreachable from applications. They hold the file's identity and creation metadata (`snug_meta`), the display profile (`snug_profile`), execution settings (`snug_settings`), credentials (`snug_secrets`, §5.4), one row per application (`snug_apps`), complete HTML per version together with the version's runtime contract (`snug_app_versions`, §9), the schema registry (`snug_app_schemas`), an append-only DDL audit (`snug_app_migrations`), a per-application documentation wiki (`snug_app_docs`), conversation history (`snug_chat_threads` and `snug_chat_messages`), synchronisation configuration (`snug_sync`) — and, since storage v4, the connection requirements and grants of §7 (`snug_connections`), so that what an application may reach travels with the user in the same file as everything else the user owns.

Two version-retention details are worth drawing out, because they encode a view of what an application is. Hosts retain at least `VERSIONS_RETAINED` — five — unpinned versions per application, pruning oldest first; but the *factory* version, the first build or the state at installation, is pinned and never pruned. Reverting is not deletion: it copies the target forward as a new version. The user can therefore always return to a working application, and the history of how it got there is not destroyed by the act of going back.

The documentation wiki deserves a note. Each application may accumulate markdown documents — advisory slugs include `vision`, `requirements`, `plan`, `lessons`, `memory`, and `next-tasks` — maintained by the agent as the application evolves, and optionally seeded by the starter that installed it (seeding never overwrites an existing document). An application is thus not a frozen artifact but a living one, carrying its own rationale forward so that a later modification is informed by earlier intent.

6.2 Per-application data as native tables

Each application's data lives as **real tables** in the same file, named `app_<token>__<name>`. The token is produced by a normative, total, and injective function of the host-assigned namespace: a UUID-shaped namespace becomes 32 lowercase hex characters with dashes stripped, and anything else becomes `'x'` followed by the hex encoding of its UTF-8 bytes. Injectivity is what makes namespaces non-colliding; the `'x'` prefix sits outside the hex alphabet, so the two ranges cannot overlap.

The naming rules are normative. Reserved prefixes — `snug_`, `sqlite_`, and `app_`, case-insensitively — are refused, blocking an application from forging another application's at-rest name. Object names must match `^[A-Za-z][A-Za-z0-9_]{0,40}$`. The single exemption is the driver-internal `snug_kv` table, held at rest as `app_<token>__snug_kv`, which backs the key-value storage operations of §4.1. A conforming host **refuses to persist** any runtime whose object names violate the rule, failing closed with prior state retained; an unvalidated name is never interpolated into SQL.

These schemas are designed by the model from the application's own requirements. A portfolio tracker receives tables for equities, trades, and dividends, because that is what a portfolio tracker is about. This is a departure from storing application data as an opaque blob: the shape is legible at rest, so both the host and the agent can reason about the application's data when modifying it later — a property §9 turns into a user-facing capability.

6.3 Materialisation and isolation

Storing every application's tables in one file raises the obvious question: what stops an application from reading tables that are not its own? The answer, shown in Figure 6, is that it never executes against that file.

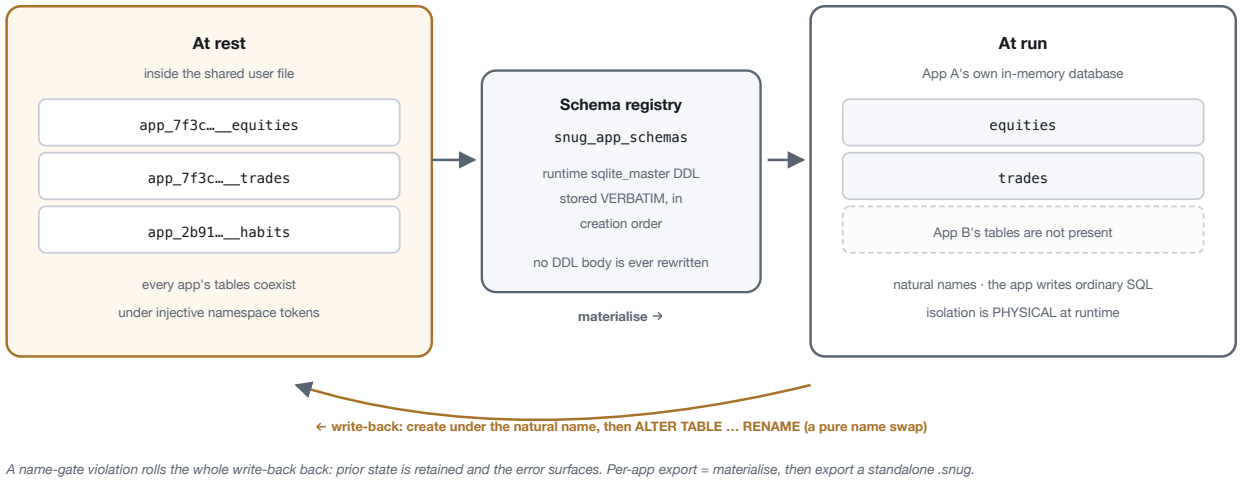

Figure 6. Materialisation. The registry stores each application's DDL verbatim; at load, those objects are replayed into a fresh in-memory database containing nothing else. The application executes ordinary SQL against natural table names, and isolation at runtime is physical rather than advisory.

At load, the host materialises the application's objects into a database belonging to that application alone, under their natural names. Application SQL executes there. Host-namespace tables and other applications' tables are not merely protected — they are absent from the database being queried. Isolation is physical at runtime, which is a stronger property than validation, because it does not depend on catching every hostile query.

Two implementation commitments make this trustworthy. **No DDL body is ever rewritten.** The registry stores the runtime schema verbatim — tables, indexes, triggers, and views, in creation order — and replay reproduces it exactly. At-rest names are produced by `ALTER TABLE … RENAME`, a pure name swap, rather than by editing statement text. String-rewriting SQL was considered and rejected: statement parsing is fragile and injection-prone, and the embedded engine exposes no authorizer with which to validate the result.

Per-application export follows directly: materialise, then export. The user receives a standalone `.snug` file with natural table names — their application's data, in a form with no dependency on Snug at all.

6.4 Host obligations

A conforming host must satisfy four obligations, which together are what make the file portable in practice rather than in principle.

1. **Never require its backend for execution.** The client is static files; application reads and writes hit the browser's copy of the user database.
2. **Offer export and import.** One-click download and upload of the canonical file. Default export strips secrets and vacuums; including them is an explicit opt-in. On import, the file's endpoint settings are treated as executable configuration and require user re-confirmation before any agent turn runs — an imported file is untrusted input like any other, a rule §7 and §9 extend to connections and contracts.
3. **Optionally act as a synchronisation origin**, using conditional requests for concurrency: retrieval returns bytes with a revision tag; a write carries the expected revision and is rejected on mismatch, with a missing precondition rejected outright. Cookie authentication requires a double-submit CSRF token, and

cross-origin policy is fail-closed. First login provisions the user record only — a host never creates an empty database image that could overwrite local state; the client pushes up.

4. **Support pluggable origins** through an information/pull/push contract, so that a user may keep their file in storage they control. Conflicts are resolved by compare-and-swap on the revision token; divergence is surfaced to the user, and last-writer-wins occurs only on explicit user action.

The third obligation's final clause is small and load-bearing. A host that provisions an empty file on first login will eventually overwrite somebody's real data with emptiness, because a fresh login on a new device is indistinguishable from a first-ever login. Pushing up rather than provisioning down makes the failure impossible.

6.5 Execution modes

Three modes are defined, summarised in Figure 7, and the user chooses between them in settings.

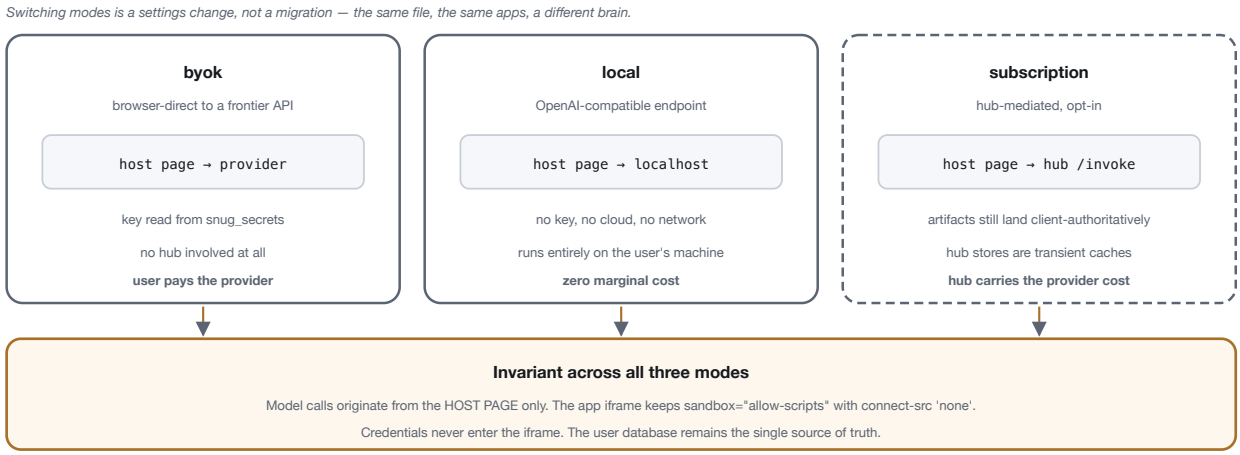

Figure 7. Execution modes. The three differ in who pays and where inference happens, and in nothing else that matters: the same file, the same applications, the same security constraints. Local mode runs with no key, no cloud, and no network dependency at all.

In every mode the user database remains authoritative. In subscription mode a host may cache artifacts and conversation history server-side, but the client fetches artifact content and writes it into the user database itself; host-side stores are transient caches. This is what keeps mode-switching a settings change rather than a migration — and what ensures that a host's disappearance costs the user a service, not their software.

6.6 File naming: the extension is a convention, not a format claim

The canonical user file is `user.snug`, and the artifact a host offers for download is `snug-user.snug`. A conforming implementation determines a file's format from its **leading bytes**, never from its name: `SQLite format 3` means a plain database; `SNUGENC1` means the protected container of §6.6. Implementations SHOULD accept the historical `.sqlite` extension on input — users hold exports and backups made before the rename — and MUST NOT reject a file on its extension alone. A host that finds a pre-existing `user.sqlite` and no `user.snug` reads it and adopts the canonical name on its next write, *without renaming, copying, or deleting the original*: renaming a file an implementation looks for is a data-loss operation unless every derived name — sync sidecars, quarantine copies, remote paths — moves with it.

6.7 Protected files: the SNUGENC1 container

A user MAY protect their file with a passphrase. A protected file is not a SQLite database; it is a container carrying one: the whole database encrypted under AES-256-GCM with a random 32-byte file key, and that file key independently wrapped by keys derived — PBKDF2-HMAC-SHA256, 600,000 iterations, with the count carried in the header so it can be raised without orphaning old files — from each of the container's secrets. Key wrapping rather than direct encryption is what makes a passphrase change cheap, and what lets a second slot exist at all.

Five of the container's rules are worth restating here because each one encodes a way users actually lose data. A conforming implementation MUST create a **recovery key slot** alongside the passphrase — at least 128 bits of real entropy — because a single point of loss is not an acceptable shape for a user's only copy of their data. It MUST bind the header as authenticated data and use fresh random nonces for every encryption operation. It MUST distinguish *locked* — a healthy container that no supplied secret opened — from *corrupt*, because reporting damage as a wrong passphrase sends a user hunting for a secret that was never the problem, and reporting a wrong passphrase as damage invites them to destroy a healthy file. A locked file is never quarantined, rewritten, or replaced. And the container MUST be self-opening — everything except the secret travels inside it — so that any device holding the file and the secret can open it, which is what keeps the portability promise true for protected files.

What protection does not change is custody (§5.4): the key was always the user's and still is. And one property is stated to the user before it is enabled, not after: a protected file whose passphrase and recovery key are both lost is unrecoverable. There is no escrow and no reset. That is the property, not a gap in it.

7 Connected applications

A personal application becomes most valuable at the moment it can reach the services its owner already uses — and that is also the moment it becomes dangerous, because reaching a service means holding a credential, and the application is hostile by assumption (G2). Most platforms resolve this tension with a connector catalogue: a fixed set of integrations, each hand-built, each reviewed, each maintained by the vendor. Snug takes the opposite position: **the integration engineer is the model, the reviewer is the user, and the enforcement is the host's** — so any provider is reachable, including ones no catalogue would ever list, without a single credential entering code that was written by a language model minutes ago.

This section specifies that surface. It is the largest part of the specification, and every rule in it is a consequence of one doctrine: *the model proposes, the human approves, the host enforces* (G7).

7.1 Requirements and grants

The design splits "connection" into two halves with different writers, different lifetimes, and different trust, shown in Figure 8. A **requirement** describes what the application needs — the provider, the auth kind, the credential fields to collect, the registration walkthrough, endpoints and scopes, the header or query template that places the credential, and the hosts it declares it will call. It is credential-free, and it is written at *authoring* moments: an app build, an auth-touching edit, a starter install. A **grant** records what the user allowed: the approval, its timestamp, and — decisively — the **frozen host ceiling**, the exact set of hostnames the connection may ever reach, computed at the moment of approval from the requirement the human ac-

tually reviewed. Credential values live in neither; they are held in the user's file under a namespace nothing but the host-side executor reads.

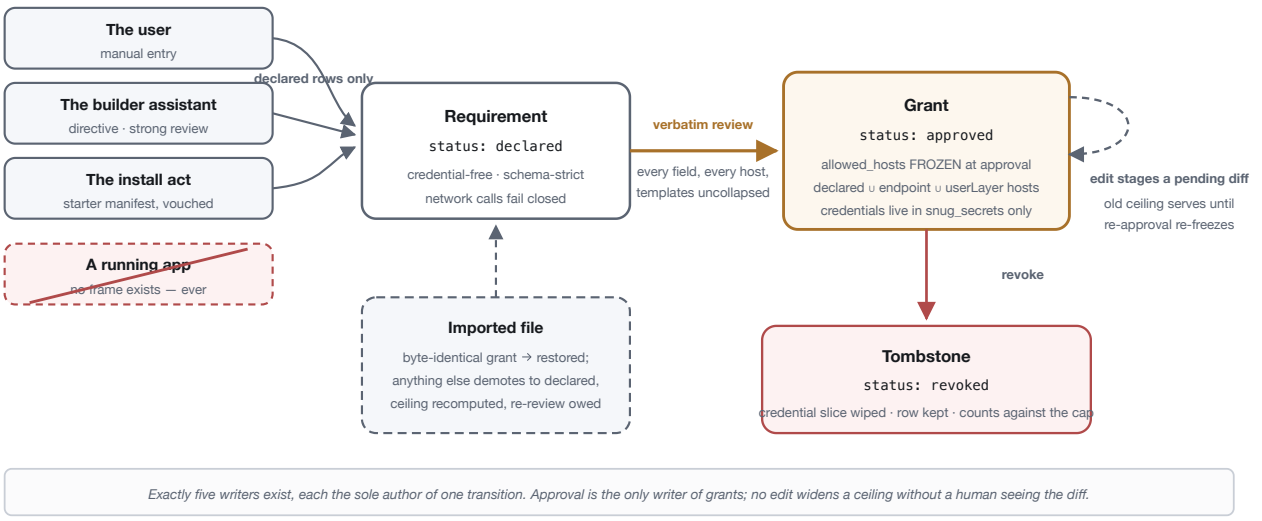

Figure 8. The connection lifecycle. Three proposers may write credential-free requirements; a running application is structurally not among them. Only the user's approval writes a grant and freezes the ceiling; edits stage a diff against the grant rather than touching it; revocation wipes the credential slice and leaves a visible tombstone. Imported rows demote to declared unless byte-identical to a grant the device already held.

A running application may never propose a connection. There is no frame, no SDK call, and no announce field through which application code can ask for a credential grant — the capability is structurally absent from the wire, not policy-refused at review. Exactly three proposers exist: the user, by hand; the builder assistant, through a reviewed directive in the build conversation; and the install act, through a starter's manifest whose vouching requires the installed bytes to match the shipped starter. All three may write *declared* rows only. Approval is the only writer of grants, and an edit against an approved grant stages a pending diff — the old ceiling keeps serving until the user re-approves the change field by field. There is no path by which an edit widens a ceiling without a human seeing the diff.

Requirements are strict at every level — an unknown key anywhere is a rejection, never a passthrough, so a seat added in a future revision cannot ride in unreviewed on a channel that predates it. Seven auth kinds are defined: `api_key`, `bearer_token`, `basic_auth`, `oauth2_client_creds`, `oauth2_auth_code`, `linked_device` (§8), and `none` — the keyless provider, a first-class kind rather than an absent row, because a public API still needs a host ceiling and a keyless connection with no grant still fails closed. Devices on the user's own network, whose address no registry can pin, are declared through a `lanHost` seat — a declaration that an address *will be collected*, never an address — and the collected address must be a private-range literal of the declared class, alone, or the requirement is refused: a public host wearing LAN clothes is exactly the shape the rule exists to stop.

7.2 Templates, and why the credential can be placed but never read

A requirement may carry a header template or a query template that describes where the credential goes — `Authorization: Bearer {{api_key}}`, a signed timestamp header, a `?appid=` query parameter. Template values may reference only declared field keys, five pinned request tokens, five pinned helpers

(timestamp, base64, two HMAC forms, and one provider-scoped JWT signer whose arguments must both be declared fields), or quoted literals — and the grammar is flat: no nesting, no user-defined functions, and a quoted argument is a literal in the engine, not only in the linter, so a template that passed review because its quotes made it look inert can never resolve those quotes into a live credential. An unknown token is a review-time rejection, never a silent fallback-to-literal: one transposed character in a signing template would otherwise sign the literal string `"api_secrt"` and produce a plausible signature the provider rejects — at signing time, where nobody is looking.

Helpers are a closed enumeration, and additions are reviewed spec-level changes rather than configuration, because an unused helper is reachable signing surface. Signing schemes that cannot be expressed by the enum are added as provider-scoped helpers, never as a general-purpose escape hatch.

7.3 The registry, and what a host may vouch for

A host may keep a registry of pinned providers — known names, known hosts, known endpoints, known scopes, and pairing ceremonies. The registry's load-bearing rule is the **borrow ban**: a requirement that names a registry provider, *or* whose declared hosts intersect a registry entry's, has the registry's pinned values substituted for its own — discarded, not merged. Both triggers are required, because name-matching alone is evaded by renaming and host-matching alone is evaded by trading on a brand while declaring no overlapping host; and the ban is kind-agnostic, so an API-key requirement naming a known OAuth provider cannot borrow its legitimacy while pointing the credential somewhere else. A borrow is disclosed to the user: "these values came from the host's registry, not from the app" is a materially different provenance claim.

Registry pins encode a principle this paper wants on the record: **prefer a scope the token cannot exceed over a rule the application promises to follow.** The reference implementation's mail integration, for example, pins scopes that permit moving a message to trash and structurally exclude the scope that permits permanent deletion — so irreversible destruction is impossible *regardless of application code*, which is a stronger statement than any review of application behaviour could make. Pinned scopes are entry-level identity, never per-flow options; and a pin change against an approved grant always re-consents, never silently promotes.

For providers whose credential is obtained by ceremony rather than typed — a button pressed on a bridge, a QR code scanned by a phone, a one-time setup token pasted from a portal — the ceremony lives in the *registry entry*, never on the persisted requirement: a requirement seat carrying claim mechanics would be a channel through which a prompt-injected declaration aims an uncredentialed request. Every pairing family carries a mandatory *verify* probe — prove the counterparty is live and speaking the expected protocol before any credential is written — and the resulting proof markers are recorded per transport family, so a stale proof from one family can never vouch for another.

7.4 The executor

Everything above describes what may be *declared* and *approved*. What may be *done* is decided in exactly one place: a host-side executor that is the only code in the system that both reads credential values and calls fetch. Every application-originated network request passes its ordered gates, shown in Figure 9 — shape validation, host-assigned identity binding, the grant check, the frozen ceiling, the SSRF guard that refuses private-address literals even when a ceiling contains them, the human confirmation for every mutating

method, credential-header stripping, host-side injection, a fetch that never follows redirects, and a size-capped, scrubbed read.

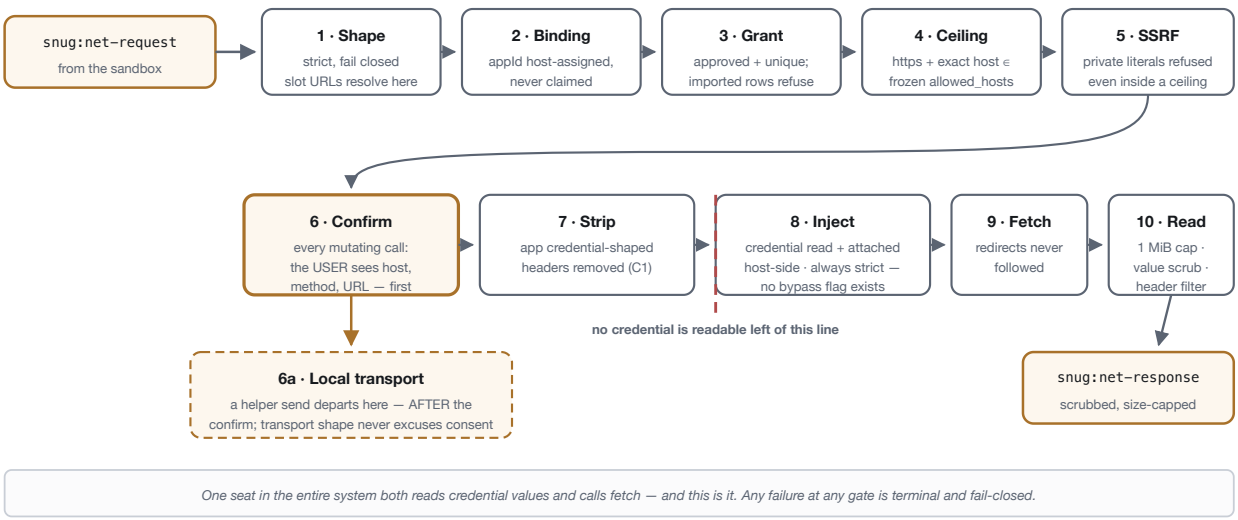

Figure 9. The executor's gate sequence. The confirmation gate precedes any credential read — a denied request resolves no secret and performs no fetch. A send to a local helper transport (§8) departs at gate 6a, after the confirmation and carrying the strip, injection, and scrub obligations with it; transport shape never excuses consent.

Two orderings in that pipeline are load-bearing enough to state as rules. **The confirmation precedes any credential read**: a denied mutating request not only performs no fetch, it never resolves a secret at all. And **local transports depart after the confirmation, never before**: the reference implementation's audit found a helper send that had been placed above the confirm gate under a comment asserting the gate had answered — every send reached the helper with credentials injected and no confirmation, and the suite stayed green because every test hardcoded a granting gate. The general lesson, now a conformance item: speech and spending in the user's name are gated by the confirmation, and nothing about a request's transport ever excuses it.

Applications address their connections by *slot*, not by host: `snug-connection://<slot>/path` resolves through the grant to the connection's single approved host, runs the full gate pipeline on the resolved URL, and never discloses the resolved address to the application — the user sees the real host in the confirmation dialog; the application sees only its own slot. Beyond keeping addresses out of hostile code, the indirection solves a lifecycle problem: installed applications never receive rebuilds, so a hostname baked into shipped HTML is a liability the slot removes.

7.5 The provider lane, and requests the model composes

Because every wall above is host-enforced, a host can safely let the *model* compose provider requests on the user's behalf — "what's on my calendar", asked in chat, becoming a governed GET. The lane that does this receives connection *facts* (slot, provider name, scope summary) and never credentials; its requests execute through the same executor, the same ceiling, the same confirmation gates as application code; and a confirmation card rendered inline in chat is presentation, not authority — its resolution becomes an ordinary user message, and the only approval seat remains the executor's gate. The model is a request author, never a

grant author: zero ceiling matches means a refusal and a connect offer, exactly as it does for application code.

One forward-looking surface is specified in provisional form: a **standing approval**, by which a user explicitly arms a narrow, frozen scope — one connection, one target thread, one trigger class, rate-capped, quiet-houred, kill-switched, journaled — so that a specific class of send may proceed unattended. The specification pins the shape (a separate gate consulted before the session gate, deriving its target from two independent request sources that must agree, falling through to the ordinary confirmation for anything outside its frozen scope) precisely so implementers do not invent looser versions; the arming channel itself is deliberately unspecified until it has been built and audited. Armed is a recorded answer, not a bypass.

8 Linked-device connections

Some of the services people most want their personal software to reach have no API to call: they authenticate a *device session*, not a request. There is no key to paste and no OAuth redirect to run — a long-lived stateful client holds session key material and speaks a proprietary transport. A personal-messaging account is the canonical case. Most platforms simply cannot express this; the ones that try tend to do it by trusting a local companion process and punching a hole in whatever governance they had. The specification defines the surface — shown in Figure 10 — without weakening a single wall.

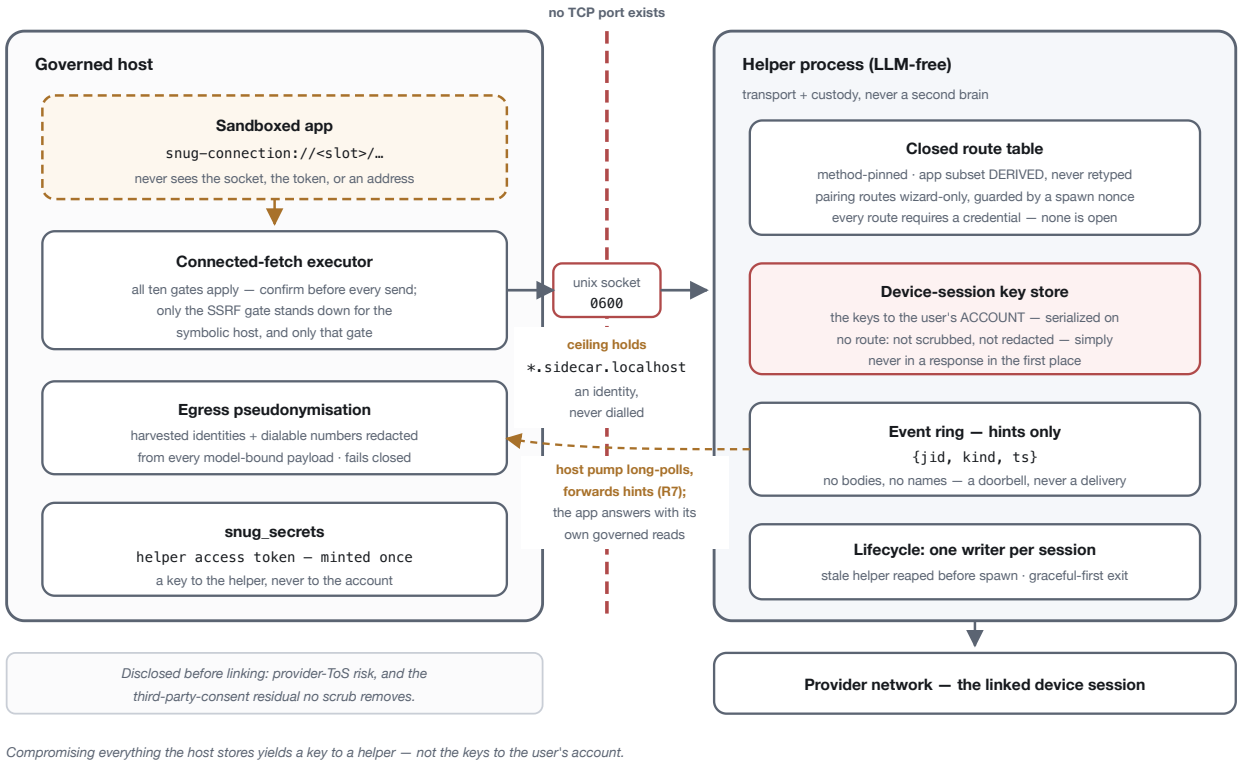

Figure 10. Linked-device connections. The application reaches the helper only through the governed executor; the ceiling holds a symbolic host that is an identity, never an address; the helper holds the device session's keys and serialises them on no route; pushes are hints (R7); and identities are pseudonymised before any model-bound payload leaves the host.

8.1 A helper is a capability, never a host

The device session lives in a **helper**: a local, user-installed companion process that owns the session keys, the transport, and the sync state — and nothing else. The helper is LLM-free by rule: transport and custody, never a second brain; every analysis or compose turn runs in the governed host, where the protocol's walls exist.

The interesting question is how the application reaches it, and the answer is the protocol's most instructive negative result. The obvious design — run the helper on a loopback port and put `127.0.0.1` in the ceiling — is wrong twice over: the ceiling is host-granular with no port dimension, so admitting the loopback address would grant every loopback port on the machine to any approved connection; and a `host:port` pair is not even a legal ceiling entry. The specification therefore forbids loopback addresses in ceilings outright. Instead, the helper listens on a **unix-domain socket with `0600` permissions** — no TCP endpoint exists, so there is no port to squat and no network path to filter; the filesystem decides who may connect — and the connection declares a **symbolic host**: a name under the reserved `.localhost` top-level domain, which can never resolve publicly and is never dialled. The symbolic name exists because hosts are the ceiling's unit, and a capability needs a host-shaped identity to be containable: the executor recognises it and routes to the helper transport, standing down exactly one gate — the SSRF literal guard — while every other gate, including the mutating-send confirmation, applies in full.

8.2 The custody split

Two secrets exist in this design, and they have opposite exposure rules. The **device session keys** — the keys to the user's account — live only in the helper's own store and are serialised on *no* route: not scrubbed, not redacted, simply never present in a response in the first place. The **helper access token** — the key to this helper — is minted exactly once at pairing, crosses the wire exactly once on the pairing poll that releases it, and lands in the user's file as the connection's ordinary credential, injected thereafter by the executor like any other. The asymmetry is the point: compromising everything Snug stores yields a key to a helper on the user's own machine, not the keys to the user's account.

The helper's reachable surface is a closed, method-pinned route table whose app-reachable subset is *derived* from the one table rather than maintained as a second list — two hand-written lists drift invisibly until an application reaches a route nobody intended. Pairing routes are wizard-only and guarded by a per-launch spawn nonce, because the pairing status route releases the access token and "local" is not "trusted". Every route requires a credential; none is open. Path admission decodes before it checks, refuses traversal segments, and caps request and response sizes by refusing — never truncating, because a truncated payload is a corrupt payload. Lifecycle rules keep exactly one helper writing a session at a time, reap gracefully first so the helper's final flush runs, and require a stale-process verdict to identify the process by its command line before killing anything — recorded process ids are recycled, and a bare number would kill a stranger.

8.3 Live updates are doorbells

A messaging surface needs push. Rule R7 was written for this moment: the host runs a pump that long-polls the helper's event route *through the governed executor* — the pump holds no privilege the application lacks — and forwards hint records whose every field the host rebuilds. No message bodies, no names, no media ride the push channel; the application answers a doorbell with its own governed reads, and media bytes stop at the application frame — rendered from memory, never written to the application's database, never placed in a model-bound payload.

8.4 Other people's names

A linked-device connection is the one place Snug routinely handles data about people who never consented and cannot opt out: the other participants in the user's conversations. Under a bring-your-own-key configuration their message content reaches the user's chosen model provider — that is the feature, and no honest design pretends otherwise — but their *identities* need not. The specification therefore requires a host-enforced **pseudonymisation backstop**: the host harvests third-party identities at the one governed seat every helper read crosses (extraction is the scrub — only identity fields enter the directory, never message text), and redacts those identities, together with dialable number runs, from every model-bound surface of every application holding a linked-device connection in *any* status — including a connection that arrived declared-but-unapproved through an import, because the replayable conversation data rides the same file. The scrub fails closed: an unreadable directory refuses the send rather than sending unscrubbed.

The specification also requires the claim to be stated at its true strength, and we repeat it here in the required form: the backstop is **anti-default and anti-naive, not anti-adversarial**. An application that obfuscates — homoglyphs, encodings, numbers smuggled as numerics — defeats substring redaction; a nickname that only ever appeared inside message text is invisible to the directory; and content itself reaches the provider by design. Third-party consent is a real residual, and the user is told so before linking, alongside the provider terms-of-service risk that unofficial automation carries. The harvested directory is itself third-party personal data, so it has lifecycle rules of its own: wiped, transactionally, when the last approved linked-device connection is revoked or its application deleted; deliberately surviving import, because the data it scrubs against travels in the same file; and reset per session, so one user file's contacts are never written into another's.

None of the reference binding — which messaging provider, which routes, which process runtime — is normative. What is normative is the shape: capability-not-host, the custody split, the closed derived route table, doorbell pushes, the pseudonymisation obligation, and the disclosures. Any device-session provider can be bound the same way.

9 Runtime contracts and the data surface

Two smaller surfaces complete the picture, and both exist to serve the same end: keeping the application's runtime relationship with the model cheap, portable, and owned by the user rather than the platform.

9.1 Runtime contracts: turns cheap enough for any brain

An installed application's model turn is one self-contained request (§4). What such a turn historically carried was the application *builder's* system assembly — several kilobytes of authoring instructions that a runtime move cannot act on, resent on every turn. A **runtime contract** replaces that with the handful of facts a turn genuinely needs: what the application is, which of its settings shape an answer, what state arrives each turn, and the minimal response shape expected back. The whole contract is bounded at 2560 (`RUNTIME_CONTRACT_MAX_BYTES` — serialized length; bytes for ASCII content), and the effect is protocol-level rather than cosmetic: every conforming application becomes cheap enough to run on a small local model, which is what makes a browser-resident brain with a 4K context a viable host at all.

Because a contract speaks with system authority at runtime, its custody rules are strict. It is host-assigned, never app-claimed — no frame carries one. It is version-linked, stored on the application's version row: an

ordinary edit copies it forward, and a revert restores the contract that shipped with the restored code, so code never runs under instructions written for a different version of itself. An imported contract is dropped unless it is byte-identical, after canonical serialisation, to one the importing host already holds — an untrusted file must not dictate the model's instructions; the affected application simply runs contract-less, degraded but never compromised. And a malformed contract reads as no contract: a corrupt row is not a reason to fail a user's move.

9.2 The data surface: your data answers to you

Because per-application data is stored as legible native tables (§6.2), a host can offer something no menu-driven application platform offers: the user asks anything of an application's data — "what did I spend on food last month?" against a budget app that never shipped that screen — and the model writes the SQL. The specification constrains how that surface may touch the data, with a lane model whose classification *fails closed*: a message is classified into one of eight intents, an exhaustive compile-checked map routes each intent to one of four lanes — data, feature, provider (§7), answer — and an unusable classification produces a clarifying reply, never a default lane, and in particular never the lane that writes code.

Reads are isolated by construction: generated SQL executes against a throwaway copy of the application's own materialised database — never live storage, and never behind a "read-only" flag, because isolation is a property of what the copy *contains*: other applications' tables and every host table are physically absent. Results are bounded (200 rows / 32 KiB) before re-entering the model's context, and truncation says so in-band. Writes are proposed, previewed, approved, then re-validated: the user sees the verbatim statements and an affected-row preview; execution happens only on explicit approval; and at execution the dry run reruns against live data and halts if the affected-row counts have drifted from what was approved. Declining executes nothing.

One asymmetry is disclosed rather than papered over: the *feature* lane — the one that edits the application itself — writes new versions on model authority, with no pre-write confirmation. The wall there is versioning: a visible reload, a revertable version write, and the pinned factory version. A host must disclose that asymmetry rather than let the data lane's approval gate imply coverage it does not have. And in every lane, stored rows are untrusted input — data crafted to read as instructions is the oldest trick in this genre — so prompts carrying query results delimit them as untrusted, with delimiter spellings defanged in the data.

10 Conformance

An implementation may claim conformance with the parts it implements; the wire protocol of §4 is the minimum. The list is deliberately testable; each item is asserted by the reference implementation's suite.

10.1 Hosts

- Validate every inbound frame against the published schemas, accepting unknown fields (R2) and rejecting unsupported versions with `UNSUPPORTED_VERSION` (R1); the strict net and open-url frames refuse unknown keys.
- Mint `instanceId`, deliver it in `snug:host-ready`, and route by message source, never by `appId` (R4).
- Emit exactly one terminal `snug:app-response` per accepted `requestId` (R3).

- Enforce the size classes, string caps, parse budget, and backoff of R6; push hints, never content, on host-initiated events (R7).
- Strip the five credential headers from application-originated requests at the envelope boundary (C1).
- Run application frames with `sandbox="allow-scripts"` only and `connect-src` blocked (C2); advertise only capabilities actually mediated.
- Parse agent replies with the graduated tolerance of §4.5 and convert failure to `PARSE_FAILED` with `rawExcerpt` and `attemptsRemaining`.
- Execute application code with no configured model, for applications that request none (G1).

10.2 Applications and SDKs

- Announce on mount and wait for `snug:host-ready` before issuing requests.
- Echo `instanceId` on every request and use a unique `requestId` per instance.
- Ignore unknown frame types and unknown fields (R2).
- Treat streaming frames as provisional and the terminal frame as authoritative (R3).
- Never assume storage or network APIs are available; use the frames, feature-detect from capabilities, and render honest fallbacks.

10.3 Hosts additionally implementing the storage layer (§6)

- Carry the schema version in `PRAGMA user_version`, migrate forward-only, refuse newer-stamped files, verify the table set on open, and run the legacy credential wipe once, scoped by key shape rather than prefix.
- Compute namespace tokens with the normative function, and refuse to persist any runtime whose object names violate the naming rules — failing closed.
- Materialise per-application runtimes so that isolation is physical, and store DDL verbatim.
- Offer export and import; strip secrets and vacuum by default; require re-confirmation of imported endpoint settings; demote imported connections and drop unrecognised imported contracts.
- Retain at least five unpinned versions and never prune the pinned factory version.
- Decide file format by leading bytes; read legacy names and adopt forward without deleting; honour every rule of the protected container, including locked-versus-corrupt reporting.
- Never require the backend for application execution.

10.4 Hosts additionally implementing the connected surface (§7–§8)

- Persist requirements and grants only through the five writers; enforce the slot cap; freeze ceilings at approval with stable, normalized output; stage edits as diffs.
- Apply the full validation stack — schema bounds, template lint, registry-borrow ban, provider-name guard — before a requirement is shown, stored, or acted on, and render every re-admitted seat verbatim in review: fields, walkthrough, templates uncollapsed, the complete host list, the freeze disclosure.
- Run the executor's gates in order: confirmation before any credential read on every mutating call, local transports included; redirects never followed; credentialed URLs never disclosed; responses size-capped and scrubbed.

- Disclose provenance honestly; disclose prior revocations keyed by provider, not slot; disclose armed standing approvals wherever the connection is shown.
- For linked devices: never admit a loopback address to a ceiling; enforce the custody split, the derived route table, decoded-form admission, refuse-don't-truncate caps, single-writer lifecycle, doorbell pushes, and the pseudonymisation backstop with its honest class statement and pre-link disclosures.

11 Related work

11.1 The Model Context Protocol

MCP standardises how a model reaches outward — to tools, resources, and external systems — and it does that job well enough that Snug does not attempt it. The relationship is complementary and directional, and it is the cleanest way to locate Snug in the current landscape:

	MODEL CONTEXT PROTOCOL	SNUG PROTOCOL
Direction	Agent → capability	Application → agent
Question	What can the model reach?	What can a user's application ask of the model — and reach through the host?
Author	A developer building a server	A model, on the user's instruction
Unit	Tools and resources	Sandboxed applications and their data
Trust	Server is a configured, trusted integration	Application is hostile by default
Credentials	Held and used by the server the user installed	Held by the user's file; attached only by the host, inside a human-frozen ceiling, behind a confirmation
Persistence	Out of scope	Central: one portable file

Table 3. MCP and Snug address inverse directions of the same boundary.

A host may well implement both, and we expect the common configuration to be exactly that: MCP for what the agent can do, Snug for what the user's applications can ask. The differing trust posture is the reason they are separate protocols rather than one. An MCP server is something a user or administrator deliberately installed; a Snug application is something a model wrote minutes ago from a prompt of uncertain provenance. Those two things cannot safely share a permission model — and the connected surface of §7 is what a permission model looks like when it is designed for the hostile case first: no configured trust anywhere, every reachable host frozen by a human, every mutation confirmed by one.

11.2 Vendor application platforms

Applications embedded in assistant products — sandboxed frames, model-mediated, distributed through a vendor's directory — validate the category emphatically, and their architectures rhyme with this one. The difference is ownership rather than mechanism. Those applications are publisher-built, vendor-reviewed, and vendor-hosted, and the user's data resides in the vendor's cloud under the vendor's terms. That is a reasonable arrangement for professionally published software with commercial support behind it.

It is a poor fit for the case this protocol addresses: an application built by one person for themselves, which no directory would list and no reviewer should need to approve, and whose value to its owner is precisely that it is theirs. Snug is the open, user-owned counterpart — not a competing store, but the layer that lets such software exist without one. The sandbox, not a submission queue, is the reviewer: a Snug application cannot phone home, cannot read another application's tables, and cannot touch a credential, whether or not anyone ever looked at its code.

11.3 Local-first software

The local-first tradition — Ink & Switch's articulation of it, and the broader body of work on user agency in software — supplies the values this protocol encodes: data on the user's device, no dependency on a vendor's continued existence, and real ownership rather than a licence. The essays in that tradition, and the related arguments for software as a home-cooked meal and for development practised by people who are not professional developers, describe a world in which people make small software for themselves.

What that literature largely predates is the arrival of a general-purpose authoring capability. It argued for personal software when producing it still required programming skill. The models removed that barrier and, in doing so, created the problem this specification addresses: with authoring solved, the binding constraints became isolation, custody, portability — and now connection. Snug's contribution is to answer those at the protocol layer, so that the answer is not one vendor's implementation detail.

12 Limitations and future work

Single-host at a time. The file is authoritative and synchronisation is compare-and-swap on a revision token, which detects divergence but does not merge it. Two devices editing concurrently produce a conflict the user resolves. Convergent replicated data types would allow genuine multi-device concurrency and are the natural next step; they are a substantial addition to a format whose present virtue is that any SQLite tool can read it.

Browser-bound. The isolation model is the browser's: frame sandboxing and content security policy. This is a well-understood boundary with wide deployment, but it does bind conforming hosts to browser-class environments, and it makes the platform's own correctness part of the trust base — the reference implementation's desktop shell ships on exactly one platform today because a second platform's webview demonstrably breaks the sandbox contract, and shipping it anyway would have made C2 false rather than fragile.

The standing-approval surface is provisional. The gate shape is specified; the arming channel is not, and the reference implementation's grant store does not yet survive a reload. Until both are settled and audited, unattended action remains a narrow, disclosed exception rather than a finished capability — and the specification says so rather than implying otherwise.

The pseudonymisation backstop has a stated class. It is anti-default and anti-naive, not anti-adversarial (§8), and the write-approval drift check compares row counts, not row identities. Both are disclosed limits with named upgrade paths, kept in the specification's residuals rather than its promises.

Size bounds are conservative. A 64 MiB file suits many personal applications and constrains others; a media-heavy application will meet the ceiling. Raising it interacts with synchronisation cost, since the present model transfers whole files.

No formal verification. The security properties are argued from structure and tested by suite, not proved. The isolation claim in particular — that materialisation makes cross-application reads impossible rather than merely difficult — would benefit from formal treatment, as would the executor's gate ordering.

The agent reply contract is prose-shaped. Tolerant parsing is a pragmatic response to how models behave today. As constrained decoding becomes universally available, the fallbacks in §4.5 should become dead code, and a later revision may make schema-constrained replies mandatory rather than optional.

13 Conclusion

The economics of software authorship have changed permanently, and the change is not primarily about productivity. It is that the minimum viable audience for a piece of software has fallen to one. A person can now have software that exists solely because they wanted it — which is a category that has never meaningfully existed before.

Whether that category becomes durable depends on questions models do not answer. Software built for one person must still be safe to run, since nobody reviewed it. It must be able to hold data without that data becoming somebody else's asset. It must survive its author's choice of vendor. And if it is to do real work in a person's life, it must be able to reach that person's services without anyone — vendor, publisher, or the application itself — acquiring custody of their credentials. These are the ordinary concerns of infrastructure, and they are settled by protocols or not at all.

Snug proposes a specific settlement. Treat the application as hostile and confine it absolutely; give it the model as a capability it calls rather than an engine it contains; let it reach the world only through a wall a human froze and a gate a human answers; and put everything the user accumulates into one file they can carry away — and seal, if they choose, with a passphrase only they hold. The whole surface — wire, storage, connected applications, runtime contracts, linked-device connections — is normative as of specification 1.0. All of it is MIT licensed, all of it is specified by schemas and contracts exported from a working implementation, and all of it is offered in the expectation that other implementations will find it wanting in specific, correctable ways.

The measure of this work is not whether the reference implementation succeeds. It is whether a person can build something small and strange and entirely their own — connected to their real life, answerable only to them — and still have it, on their own terms, years later, on a device and a provider nobody has chosen yet.

Normative references

1. **Snug Protocol Specification, 1.0.** The complete specification: wire protocol, portable user database, connected applications, runtime contracts, linked-device connections. snugprotocol/spec — `SPEC.md`
2. **Snug Protocol JSON Schemas.** Normative message shapes, published byte-identically from the reference implementation. snugprotocol/spec — `schemas/`
3. **Snug Threat Model, v3.** Enforced invariants with named enforcement points and tests; residuals with equal standing; re-attacked whole-surface before launch. snugprotocol/snug — `docs/threat-model.md`
4. **RFC 2119.** Key words for use in RFCs to indicate requirement levels. S. Bradner, IETF, 1997
5. **HTML Standard** — the `sandbox` attribute and its origin semantics. WHATWG

6. **Content Security Policy Level 3** — the `connect-src` directive. W3C
7. **RFC 6761.** Special-use domain names — the reserved `.localhost` top-level domain. S. Cheshire, M. Krochmal, IETF, 2013
8. **RFC 8018.** PKCS #5: password-based cryptography (PBKDF2). K. Moriarty et al., IETF, 2017
9. **NIST SP 800-38D.** Galois/Counter Mode (GCM) and GMAC. M. Dworkin, NIST, 2007
10. **SQLite File Format.** SQLite Consortium

AUTHOR

Jeetu Maker

SECURITY CONTACT

security@snugprotocol.org

STATUS

Specification 1.0 is normative: the wire protocol (published since v0.1), the storage layer at schema 6, the connected-application surface, runtime contracts, and linked-device connections. One surface — standing approvals — is explicitly provisional and so marked.

Appendix A · Message schema summary

Required fields for every message, from the normative schemas. All frames additionally require `v: 1` and `type`; validators MUST accept unknown fields on the tolerant frames (R2); the net and open-url frames are strict.

MESSAGE	REQUIRED BEYOND `v` AND `type`
`snug:app-announce`	`appId`, `displayName`
`snug:host-ready`	`instanceId`, `protocolVersions`, `capabilities`, `theme`
`snug:app-message`	`requestId`, `instanceId`, `appId`, `action`
`snug:app-cancel`	`requestId`, `instanceId`
`snug:app-response` streaming	`requestId`, `ok: true`, `streaming: true`, `text`
`snug:app-response` final	`requestId`, `ok: true`, `streaming: false`, `data`
`snug:app-response` error	`requestId`, `ok: false`, `error`
`snug:db-request`	`requestId`, `instanceId`, `op`, and the operation's own fields (e.g. `sql` for `exec`)
`snug:db-response`	`requestId`, `ok` — with `error` when `ok: false`
`snug:net-request`	`requestId`, `instanceId`, `url`, `method` strict; no appId seat
`snug:net-response`	`requestId`, `ok` — with `status`, `headers`, `body` when `ok: true`, `error` otherwise

MESSAGE	REQUIRED BEYOND `v` AND `type`
`snug:open-url-request`	`requestId`, `instanceId`, `url` strict; https-only, no userinfo
`snug:open-url-result`	`requestId`, `status`
`snug:host-event`	`event`
`snug:app-event`	`event`
`chat envelope`	`snug: 1`, `appId`, `instanceId`, `requestId`, `action`

Table 4. Required fields by message type. Optional fields are described in §4.

The envelope is not a frame: it is tag-delimited text sent to an agent endpoint, detected by the `[SNUG_APP_REQUEST]` prefix together with the `snug: 1` marker, and therefore carries no `type` field.