

METAGRID PRO

A TECHNICAL WHITE PAPER

Grounded AI

How MetaGrid Pro's AI Builder grounds language-model output in real software

Przemysław Mieszkowski
MetaGrid Pro

May 2026

Executive summary

AI Builder is a feature inside MetaGrid Pro, an iPad-based control surface used by professional composers, scoring mixers, post-production engineers, and other creative professionals. It generates working button sets from a natural-language prompt — typically eight to sixteen buttons in under a minute — covering DAWs such as Logic Pro, Cubase, Studio One, Pro Tools, and Ableton, as well as any application that exposes a standard menu bar on macOS or Windows.

The design of AI Builder rejects a common pattern in AI-powered creative tools, in which a language model invents commands and shortcuts from its training data and presents them as authoritative. That pattern produces grids full of plausible-looking but incorrect actions: outdated key bindings, shortcuts from the wrong platform, commands that exist in the model’s training data but not in the user’s installation.

This document describes the engineering approach we use to avoid that pattern. We call it *Grounded AI*. The core idea is straightforward: every proposed action is matched against a real, on-system catalog of commands before it earns a “Verified” label. Anything that cannot be matched is shown as “AI suggested” — a visible, deliberately distinct state that tells the user the model proposed something the system could not confirm. The user remains the decision-maker at every step.

This is not a marketing document. The intent is to describe, in plain technical English, how the system actually works — including its limitations. Numbers in this paper are derived from the production codebase as of May 2026.

1. The grounding problem

Language models are good at producing plausible-looking text. They are not reliable sources of fact about specific software installations. A model asked “what’s the shortcut to split a region at the playhead in Logic Pro” will produce an answer that sounds authoritative — but the answer may be from an older version of Logic Pro, from a different DAW the model confused with Logic, or simply fabricated from related context in its training data.

For most uses of generative AI, this is acceptable. The user reads the answer, evaluates it, and either uses it or doesn’t. In a control-surface context, this is unacceptable. A pro working on a deadline cannot afford to discover, mid-session, that the “Split at Playhead” button generated by the AI is bound to the wrong shortcut. The cost of a hallucinated command is not just an incorrect button — it is broken workflow, broken trust, and possibly a damaged project file.

We see this failure mode repeatedly in adjacent products. AI-powered shortcut tools that confidently produce grids of commands which turn out to be partly wrong. AI-enhanced creative software that suggests workflows referencing menu items that don't exist in the user's installed version. The failure is not the AI's — it's the product design that asked the AI to be authoritative about facts it cannot verify.

The architectural response is *grounding*: the AI does not invent commands. It chooses them from a known set, and the system enforces that constraint after the fact. This is a form of retrieval-augmented generation, applied to a very specific domain — the user's actual installed software, with its actual menu structure, its actual key bindings, and its actual plugin inventory.

The rest of this paper describes how that grounding works in practice.

2. The Catalog architecture

We use the term *Catalog* (capitalized, treated as a proper noun) to refer to the named retrieval system that grounds AI Builder. The Catalog is not a single data source. It is a layered architecture composed of four distinct mechanisms, each with its own update model, its own coverage characteristics, and its own limitations.

2.1 DAW integration commands

For each supported DAW, AI Builder draws from a *command library* — the canonical list of operations the DAW exposes through its host-control protocols (Generic Remote MIDI, AppleScript, or vendor-specific control APIs, depending on the DAW). These commands are not the DAW's keyboard shortcuts; they are the operations the DAW will respond to when MetaGrid Pro communicates with it through the appropriate channel. A “Split at Playhead” command in Logic Pro is a binding to a scripted instruction, not a key press.

Three mechanisms populate these libraries:

Hardcoded MIDI mappings. For most DAWs (Cubase, Studio One, Digital Performer, Reaper, Ableton Live), the command library is shipped with MetaGrid Pro as a static data structure. Each entry contains the command name and its MIDI mapping. These libraries are transcribed manually from the DAW's documented Generic Remote / MIDI Controller specification, because none of these DAWs expose a runtime command directory we can introspect.

The exact counts as of May 2026:

DAW	COMMANDS	SOURCE
Cubase	2,044	Hardcoded MIDI mappings
Logic Pro	1,330	Dynamic — synced per session
Studio One	1,267	Hardcoded MIDI mappings
Digital Performer	916	Hardcoded MIDI mappings
Reaper	382	Hardcoded MIDI mappings
Ableton Live	90	Hardcoded MIDI mappings

Dynamic per-session sync (Logic Pro). Logic Pro is unusual because it exposes its active command set through AppleScript. When MetaGrid Pro connects to Logic Pro via MetaServer, the active command map is queried and transferred from the DAW to the iPad. This means user customizations are reflected — if a Logic Pro user has remapped a command in their preferences, that remap is visible to AI Builder.

Empty by design (everything else). For non-DAW catalogs (Apple Shortcuts, Keyboard Maestro macros, application menus), the static command library is intentionally empty. These catalogs are populated at request time, by scanning the user’s actual system. See sections 2.2 and 2.4.

A note on currency: DAW command libraries are updated manually when DAW major versions change. There is no automated import pipeline from a vendor specification — we transcribe the documented control-surface command set, and revisions ride with our app updates. This is an honest engineering tradeoff. The benefit is that the libraries are stable and predictable. The cost is that we occasionally lag a DAW release by weeks while we transcribe new commands.

A note on filtering: we deliberately exclude certain categories. MIDI routing variants (the secondary MIDI buses some DAWs expose for parallel control surfaces) return empty command lists, to prevent double-counting. Disabled menu items — those whose macOS title begins with “Can’t...”, indicating a transient disabled state — are dropped from the menu catalog at scan time.

2.2 Menu scanning

The second layer of the Catalog is the *menu scan*: every menu command currently exposed by the active application’s menu bar.

Menu scanning is performed by MetaServer, the macOS or Windows companion application that pairs with MetaGrid Pro on the iPad. The iPad requests; MetaServer scans the active app’s menu structure and returns the result. This is an on-demand operation — there is no background scan, no daemon, no incremental refresh.

On macOS, the scanner uses Apple’s Accessibility framework to walk the menu bar of the frontmost application. It reads menu titles, submenu structure, the keyboard shortcuts associated with each menu item (key, modifiers, and any special-character glyphs), and the enabled/disabled state of each item. Disabled items — menu placeholders for transient unavailable states — are dropped from the result.

Each scanned menu item produces a record carrying both its menu path (e.g., `File > Save`) and its keyboard binding (e.g., `⌘S`). The two are inseparable on macOS — one scan produces both data sources.

Dynamic menus — submenus that populate themselves only when shown — are handled with a layered materialization strategy. The scanner attempts progressively more invasive techniques to coax submenus into revealing their contents, falling through to simulated UI interactions only when standard accessibility queries fail. If none of these produces children, the item is recorded as a leaf and a warning is logged for diagnostic purposes.

On Windows, the scanner uses Win32 menu enumeration. The Win32 path produces menu titles and submenu structure, but the operating system does not expose keyboard shortcuts in a structured way the scanner can read. This means the Windows catalog is a *menu-paths-only* catalog. AI Builder is informed of this limitation in its system prompt, and adjusts its behavior accordingly: on Windows, proposed keyboard shortcuts are explicitly marked as AI-suggested rather than verified.

Coverage limitations. Native AppKit menus on macOS are read cleanly. Electron applications (Slack, VS Code, Figma desktop) present standard AppKit menu shells, so their menu bars are AX-readable. Custom in-window menus — Adobe Acrobat’s dropdowns, some plugin GUIs, certain web-based applications with simulated menus — are not part of the menu bar tree and are therefore not captured. This is the largest single category of “Catalog incomplete” failures: a command genuinely exists in the user’s app, but the scanner cannot see it because it’s not exposed through the OS’s standard menu architecture.

2.3 Keyboard shortcuts

Keyboard shortcuts are a *byproduct* of menu scanning, not a separate scan. When the macOS scanner walks a menu item, it captures both the menu path and the keyboard binding in one pass.

The shortcut string is assembled from several independent accessibility attributes — a key character or glyph or virtual key code, plus a set of modifier flags — combined into the standard Carbon-style representation that the rest of the system consumes. Two parallel code paths exist for the modifier flags, because macOS changed the underlying representation around Sonoma; both paths converge on the same output format.

There was, until recently, a subtle bug in the legacy decoder. It misread the modifier nibble’s bit 0 as Command rather than Shift, which produced a spurious Option flag on every single-character shortcut. The visible symptom was that Figma’s `⌘]` shortcut appeared as `⌘⌥]` in AI Builder’s output. The fix is in the current codebase. We mention it here because the lesson is more interesting than the bug: when the data layer is wrong, the AI produces wrong output even though the AI itself is doing exactly what it was asked to do. Grounding is only as good as its underlying data sources.

User-customized shortcuts. Captured for menu-tied bindings on macOS, because the Accessibility layer returns live binding state. If a user has remapped `⌘S` to `⌘⇧S` in their app’s preferences, the next scan reflects that remapping. On Windows, the scanner does not read user rebindings.

Global hotkeys not tied to menus — Spotlight’s `⌘Space`, Alfred’s hotkey, Stream Deck plugin shortcuts, system-level shortcuts that don’t appear in any menu bar — are not captured. A scanner that captures these would require either an event-tap registration or a runtime daemon, neither of which we want to ship. This is a deliberate scope decision, not an oversight.

Output format. Once assembled, the keyboard shortcut joins the menu path to form a single line in the catalog passed to the AI:

```
File > Save · ⌘S  
View > Refactor > Rename...
```

The separator between path and shortcut is U+00B7 MIDDLE DOT. Lines are sorted alphabetically by full path. The sort matters for performance reasons explained in section 3.2 — a stable byte-for-byte representation of the catalog is critical to prompt caching.

2.4 External catalogs

Four further catalogs sit outside the menu/shortcut pipeline. Each is loaded on demand, never speculatively pre-warmed.

macOS Universal commands — system-wide actions such as Mission Control, Spotlight, screenshot variants, window tiling, accessibility zoom, and similar OS-level operations. Each entry has an identifier, a name, a category, an execution type (keyboard shortcut, menu invocation, AppleScript, HID key event, or Core Audio), and where relevant a `symbolicHotkeysKey` that lets us read the user’s actual bound key from macOS’s Symbolic Hotkeys preference plist. The catalog is JSON-backed, loaded by MetaServer from a bundled resource.

Keyboard Maestro macros — enumerated at request time. The format passed to the AI is one line per macro: `<group> > <macro name>`. We do not parse the macro contents; only the identifiers. A two-second polling fallback handles the case where Keyboard Maestro has not yet pushed its initial macro list.

Apple Shortcuts — the same pattern: one line per shortcut, in the format `<folder> > <shortcut name>`. Both Keyboard Maestro and Apple Shortcuts are loaded only when the user explicitly names them in their prompt. They are never auto-loaded, because the pre-warming cost (a synchronization round-trip per app per session) is significant and the user usually doesn’t need them.

Cubase plugin catalog — the most elaborate of the four. The Steinberg-side plugin manager lists every VST, AU, and AAX plugin installed on the user's system. The catalog is filtered (by vendor, kind, category) at request time, with a 100-entry cap per slice to bound token cost. Queries that exceed the cap return a vendor/category index instead, with a hint to refine the filter. A warm cache is preferred — we re-fetch only when the user says they just installed something new.

On-demand loading is the rule, not an exception. None of these external catalogs is loaded until the model requests it through an action request to MetaGrid Pro. The reasoning is operational: each catalog has a non-trivial loading cost (a Mac-side scan, a synchronization round-trip, or both), and there is no benefit to paying that cost speculatively. The model knows what to ask for from the conversation context.

3. Verification methodology

The Catalog is the source of truth. Verification is the process of checking, after the AI has proposed an action, whether that action exists in the loaded Catalog. If it does, the action is *Verified*. If it doesn't, it is *AI suggested*. Both states are visible to the user.

3.1 The matching algorithm

Verification uses exact match after lowercasing — not fuzzy similarity, not embedding similarity, not token-subset matching.

For each AI-proposed action of type `keyboard`, the verifier consults two lookup tables built from the loaded catalog:

1. A dictionary mapping `shortcut display string` (e.g., `⌘S`) to the matching catalog item
2. A dictionary mapping `lowercased item title` (e.g., `save`) to the matching catalog item

The verifier then runs three checks, in order:

1. If the AI's proposed shortcut is a key in the shortcut dictionary, the action is **Verified**.
 2. Otherwise, if the AI's proposed label is a key in the title dictionary, the action is **Verified** — with auto-correction: the catalog's canonical shortcut overwrites whatever shortcut the AI proposed. This is the recovery path for the common case where the AI picks the right command but the wrong shortcut.
 3. Otherwise, the action is **unverified**. The reason depends on the AI's self-declared source: if the AI flagged the action as `ai_suggested`, that designation is preserved (yellow badge). If the AI claimed the action was from the catalog (`source: "catalog"`), but the verifier finds nothing, the action gets a `notInCatalog` reason (red badge, visible to the user — this is the AI-cheating case).
-

Non-keyboard action types (MIDI commands, DAW commands, menu invocations, Keyboard Maestro macros, Apple Shortcuts, delays, text actions, app switches, macOS Universal commands, Cubase plugins) are *structurally verified*: the act of choosing the type implies the action is grounded. A `daw` action carries a DAW-library row identifier; a `cubase_plugin` action carries a plugin UID that the picker resolves at apply time; a MIDI action is fully parameterized.

There is no fuzzy fallback. A near-miss on shortcut combined with a near-miss on label produces “not in catalog.” This is conservative on purpose. The cost of falsely verifying a near-miss is that the user trusts the green badge and bakes the wrong shortcut into a grid. The auto-correction step is the one calibrated concession to fuzziness: it allows the label to drive a confident match even when the AI hallucinated the shortcut, because the catalog’s shortcut is then substituted in.

A token-subset matcher (e.g., to match the AI’s “Front” against the catalog’s “Bring to Front”) was discussed earlier in development but did not land as code. The verifier still uses exact match. If a token-subset path is added later, it would sit between steps 2 and 3 as a soft-confirmation rule, not a verification rule — anything matched by token-subset would carry a distinct visual state, not the green Verified badge.

3.2 What the AI receives

The AI’s input is assembled at request time and is highly conditional. There is no single static system prompt.

Always present in the system prompt:

- A preamble that constrains the response to valid JSON
- Platform rules, conditioned on whether the user is on macOS or Windows, connected to MetaServer or not, and using a DAW or not (three distinct combinations of platform copy exist, and only the relevant one is included)
- A data-source format specification that explains the catalog block format
- The supported action-type list, also conditional (the `daw` action type is only mentioned when a DAW is the target; `macos_universal` is only mentioned on Mac-connected sessions; `cubase_plugin` only on Cubase)
- MIDI detection and format rules
- Action-priority rules: DAW commands take priority over keyboard shortcuts over menu invocations, when multiple types could implement the same intent
- The muscle-memory exclusion list (see section 5.1)

Dynamic data, injected once requested by the AI:

- The full menu/shortcut catalog for the active application
- The full DAW command library for the active DAW
- The plugin vendor and category index for Cubase, when plugins have been loaded

- The user’s currently-copied button’s style, when the user has invoked “load copied style”
- The current visual styles of the selected buttons, in theme-only restyle operations

Contextual flags about user state:

- The number of selected buttons and the active profile name
- A flag indicating whether a copied button is present on the clipboard

Notably, the grid contents themselves are **not** sent by default. Only the buttons currently selected get their slot summaries serialized, and only when the operation is a grid-batch generation rather than a chat refinement turn. This is a deliberate minimization: the AI does not need to see the full grid to do its job.

Serialization is plain UTF-8 text. The catalog block is one line per menu item, sorted alphabetically. The DAW block is one line per command. The plugin catalog uses pipe-separated fields. There is no JSON or structured format inside the system prompt — the AI reads it as documented in the format spec.

Prompt caching. The entire system prompt is wrapped in a single ephemeral cache control block. Anthropic’s prompt-caching beta is used. On the first turn with a fresh app and catalog, the full input cost is paid; on subsequent turns with the same prompt, cache reads cost 10% of the input price. Loading a new catalog mid-session invalidates the cache and the next turn pays full cost again. The alphabetical sort of catalog entries is therefore not cosmetic — it ensures that two sessions with the same catalog produce byte-identical system prompts, which is the prerequisite for cache hits.

3.3 Where verification happens

Verification happens entirely client-side, on the iPad, after the AI’s response arrives. The AI has no access to the verification logic. The AI’s job is to propose actions, in valid JSON, following the rules in the system prompt. The verifier’s job is to check those proposals against the loaded catalogs and label them appropriately.

This is a deliberate architectural choice. Server-side verification (asking Anthropic’s API to enforce catalog membership) is not possible — the API does not know what our catalogs contain. We could, in principle, send a verification check back to the AI in a second turn (“here are the actions you proposed, here is the catalog, please confirm each one is grounded”). We don’t, because it doubles the cost and the latency for no improvement in correctness — the catalog is on the device, the proposed actions are on the device, the match check is trivial.

Anthropic, in other words, runs the language model. We run the verifier.

A common drift mode: the AI occasionally emits the full path string in the shortcut field — `Save · ⌘S` rather than `⌘S`. The parser handles this with a preprocessing step that strips the catalog prefix before verification. This is a soft correction, not a verification step — it normalizes the AI’s output to the expected format. It applies only to keyboard-type actions; MIDI patterns and other types preserve their original format.

4. Model selection and Anthropic integration

AI Builder uses Anthropic’s Claude models, accessed through Anthropic’s public API. The user supplies their own API key. There is no MetaGrid Pro proxy or relay between the iPad and the API endpoint.

4.1 Which models, for which operations

Two model identifiers are wired into the system as of this writing:

- `claude-sonnet-4-20250514` — Claude Sonnet 4
- `claude-haiku-4-5-20251001` — Claude Haiku 4.5

OPERATION	MODEL
Per-object edit (Refine With AI)	Sonnet 4
Chat refinement turns inside the wizard	Sonnet 4
Grid batch generation (the Build flow’s final step)	Haiku 4.5
Ideas (contextual suggestion strips)	Haiku 4.5
API key sanity check	Sonnet 4 (minimum tokens)

The split reflects a structural observation: Haiku 4.5 is the workhorse for catalog-heavy, prompt-cached, batch-generation operations where the input is large, the output is well-formed JSON, and the throughput matters. Sonnet 4 handles single-object refinement and conversational turns, where the dialogue quality and reasoning depth matter more than throughput.

This division is a tuning choice, not an architectural constraint. As Anthropic releases new models, the mix will be updated to reflect the best balance of quality and cost at any given moment. The model identifiers are not stable assumptions for users to encode in their workflows.

Generation parameters. The default `max_tokens` is 4,096, overridden to 16,384 for grid batch generation, and 16 for validation calls. Temperature and `top_p` are not set explicitly; the API defaults apply. We have not formally measured the effect of explicit temperature on output quality and have not tuned it. The structured, schema-constrained nature of the output (JSON conforming to a known shape) makes temperature less critical than it would be for open-ended generation.

4.2 Prompt structure and token shape

Two distinct prompt shapes exist.

The **chat system prompt** is the conditional document described in section 3.2 — preamble, platform rules, format spec, action priority, exclusion list. Combined with the chat history, it drives every conversational turn.

The **generate prompt** is shorter and more directive. It references the chat system prompt's rules and adds an instruction to produce JSON with a specific number of items. The full system rule blocks are stored in an external template file (`ai-prompts.json`), interpolated with current state at request time, and refreshed in the background every 24 hours from a low-priority HTTPS endpoint. If the refresh fails, the bundled-with-app version is used. No request data flows through this endpoint — it is a pull-only template download.

Approximate token sizes, derived from the prompt assembly logic (not from telemetry, which we do not collect):

COMPONENT	APPROXIMATE TOKENS
Empty-catalog system prompt	4,000 – 6,000
Mid-sized non-DAW catalog (Figma, Notion)	adds 2,000 – 4,000
Full Cubase joined catalog with plugin index	adds 10,000 – 15,000
Typical Build response	500 – 2,000 output
Typical chat-turn response	200 – 800 output

These ranges are not statistically derived. They are magnitude estimates from the structure of the prompt and the schema of the response.

4.3 Cost reality

We do not collect cost telemetry. The only instrumentation is a development log line that prints input/output/cache-creation/cache-read token counts. No USD conversion, no aggregation, no upload.

Using current Anthropic API pricing — Sonnet 4 at \$3 per million input tokens and \$15 per million output tokens (fresh), Haiku 4.5 at \$0.80 per million input tokens and \$4 per million output tokens (fresh), with cache reads at 10% of input price and cache writes at a 25% premium — the cost of a typical Build operation lands somewhere in the range of \$0.005 to \$0.05.

The variation depends primarily on two factors: catalog size (a Cubase session with plugins loaded is more expensive than a Notion session) and cache state (a warm cache is roughly an order of magnitude cheaper than a cold one). A typical second-or-later Build in the same session, with a warm cache, costs well under a cent. A first Build of the session with a large catalog can approach the higher end of the range.

This is not a precision claim. We do not log per-user costs and have no aggregate view. The range is derived from token magnitudes and current API list prices. Pricing fluctuates, and users monitor their actual usage in their own Anthropic console.

5. Privacy and data handling

The privacy posture of AI Builder is structural, not aspirational. It follows from the architecture, not from policy.

5.1 What leaves the device

Two outbound HTTPS endpoints are reachable from the AI Builder code path:

1. `api.anthropic.com/v1/messages` — every AI request, sent directly from the iPad to Anthropic's API
2. `metagrid.app/config/ai-prompts.json` — a 24-hour background refresh of the prompt template file, pull-only, with no request data attached

Every AI request body contains: the selected model identifier; `max_tokens` and `stream` parameters; the system prompt (with the loaded catalogs); and the messages array (for chat, the running conversation; for generation, the slot summary and user request). The user's Anthropic API key is set as the `x-api-key` header on each request. The key is read from the iOS Keychain and never logged.

Direct to Anthropic. The iPad calls `api.anthropic.com` directly using `NSURLSession`. There is no MetaGrid Pro proxy, no relay, no enrichment service in between. Failures are returned to the device and surfaced in the UI. We could not log API traffic even if we wanted to — there is no machine of ours that the traffic passes through.

5.2 What stays on the device

DATA	STORAGE LOCATION
Anthropic API key	iOS Keychain, kSecAttrAccessibleAfterFirstUnlock
Grid contents and configuration	Core Data, on-device
Loaded catalog data	In-memory only, not persisted
Recent prompts (max 10 per profile, 5 displayed)	UserDefaults, profile-keyed
Ideas cache (24-hour TTL)	UserDefaults, profile-keyed
Wizard UI defaults (theme, layout)	UserDefaults

None of these phone home. The persistence is local-only on every storage site. iCloud Drive or iTunes backups may include this data if the user has opted into device backup, but the data is in their backup, not ours.

5.3 What Anthropic sees

The user supplies their own Anthropic API key. We use Anthropic's standard `/v1/messages` API with the documented prompt-caching beta. We do not enable, claim, or configure any non-default data-handling option on the user's behalf.

Anthropic's standard API retention policy applies to requests made through AI Builder, because those requests are made under the user's own API account. Users who require zero-data-retention must arrange that directly with Anthropic through their organization settings on Anthropic's side. MetaGrid Pro cannot grant or revoke that flag, and we do not bundle a separate enterprise relay.

The plain framing: **MetaGrid Pro forwards your request to Anthropic on your behalf, using your key, under the data-handling policy of your Anthropic account.** That is the privacy contract.

5.4 What we deliberately do not measure

We do not collect telemetry. We do not log AI Builder usage. We have no aggregate view of:

- How many Build, Refine, or Ideas operations have been performed
- The verification hit rate (what percentage of AI proposals verify on the first pass)
- Build latency (end-to-end time from user request to populated grid)

- Most common prompts
- Operation distribution by app or DAW

This is a deliberate engineering choice, not an oversight. Opening a telemetry channel would be straightforward — a standard analytics SDK, a few well-placed event calls, a backend ingestion endpoint. We have not done it.

The consequence is that some of the numbers a white paper would otherwise quote are unavailable. We cannot say “87% of AI proposals verify on the first pass” because we have never measured it. We can say that the system is designed for grounding and that the failure modes are visible to the user; we cannot offer aggregate evidence that the design works at population scale.

This stance is consistent with the privacy framing. We do not have the numbers because we deliberately do not collect them.

6. Design principles

Three design choices shape AI Builder more than any others. Each was contested during development; each, in retrospect, defines what the product is.

6.1 The keyboard-reflex exclusion

AI Builder deliberately does not propose commands that users already trigger from the keyboard without thinking — Undo, Redo, Cut, Copy, Paste, Select All, Find, Save, Save As, New, Open, Close, Quit, Print. The full list lives in the system prompt and is communicated to the AI as a constraint: do not propose these unless the user names them explicitly.

The principle behind the exclusion: **the grid is for the work the keyboard can’t do**. Users come to MetaGrid Pro because there are operations that benefit from a dedicated physical button — operations that are nested under three levels of menus, operations whose default shortcut conflicts with other software, operations that need to be reached without taking the hands off the work. Putting ⌘Z on the grid is wasteful: the user already has ⌘Z on the keyboard, and the keyboard is faster than any iPad tap.

The exclusion is implemented prompt-side, not client-side. There is no post-hoc filter that strips these actions after the model returns them. The model is instructed to avoid proposing them in the first place, which preserves the model’s limited proposal slots for higher-value actions. The exception clause permits the model to include excluded commands if the user names them — “give me a big red Quit button” still works.

This is the kind of decision that AI tools rarely make. Most AI-powered creative software is optimized for “show what the AI can do.” MetaGrid Pro is optimized for “give the user the buttons they actually need.” The exclusion is a small encoded opinion about what a control surface is for.

6.2 The “AI suggested” honesty stance

The decision to surface a distinct *AI suggested* state — visually different from *Verified*, with different badge color and different copy — was treated as load-bearing from early in the design.

We had seen, in adjacent products, the failure mode of silent guessing: an AI feature confidently produces output, the user discovers later that some of it is wrong, and trust in the feature collapses. The internal framing was: *either we can prove it works, or we say we can't*. Anything that cannot be grounded against a loaded Catalog gets the AI-suggested badge, period. The model is also told in the system prompt that misrepresenting a fabricated shortcut as `source: "catalog"` is a hard error — the verifier catches the misrepresentation and flags the result distinctly (the `notInCatalog` reason, with the red badge).

Two alternatives were considered and rejected.

A **unified single badge** (just “AI Builder produced this”) would have been visually cleaner. It was rejected because it conceals the load-bearing distinction. The user has no way to know which actions to trust at a glance.

A **scoring system** (Verified / Likely / Guess) was considered and rejected because we cannot honestly score “Likely.” The verifier can determine whether an action matches the catalog. It cannot determine whether an unmatched action is *probably* right. A three-state UI with one state that is essentially a guess about a guess felt false.

The binary Verified versus AI-suggested split is honest about what we can and cannot prove. It also creates a natural escalation path: if the user sees many yellow badges, they know which catalogs to load.

6.3 The variable-step wizard

AI Builder’s user-facing flow is a five-state stepper (Intent, App, Actions, Skin, Generate), with variable journey lengths depending on what the user is building. A “commands path” goes Intent → App → Actions → Skin → Generate. A “MIDI path” goes Intent → MIDI → Skin → Generate.

The split serves three purposes. **Each step is a single decision** — the user is choosing one thing at a time, and can change direction without restarting. **The AI participates only where it’s useful** — the Actions step is heavily AI-driven, but the Skin step is a deterministic preset chooser, because asking the AI to pick a theme is the wrong use of a language model. **Verification state lives at the Actions step** — where the Verified and AI-suggested badges are visible, and where the user can dismiss bad rows before they propagate.

Single-shot generation — a one-prompt, instant-grid flow — was considered earlier in the design and rejected. The reason: a single-shot flow would either hide the Verified versus AI-suggested distinction, or surface it as a post-hoc warning, both of which weaken the trust contract. The current flow makes the badges part of the decision, not a footnote.

This is a place where opinionated design beat user-research-driven design. A one-prompt flow tests better in usability sessions — it is faster, it feels more magical, it is the kind of demo that wins a conference talk. We chose the slower flow because the slower flow is the one that does not break in production.

7. Edge cases and limitations

This section names the failure modes honestly. A grounding system is only as good as its catalogs, and the catalogs have gaps.

7.1 Real commands that don't verify

There are cases where a command genuinely exists in the user's installed software but does not earn the Verified badge — because the menu scanner could not see it, because it lives in a plugin's UI that does not register with the menu bar, or because it requires a document state the scanner cannot reproduce.

In those cases, the command appears as *AI suggested*, even though it is correct. The user can still apply it; the action is functional. The badge tells them we could not confirm it through our catalog.

The mitigation is the auto-correction step in the verifier: when the AI's label matches a catalog entry even though the AI's shortcut does not, the action verifies with the catalog's shortcut substituted in. This catches cases where the AI knew the right command name but proposed an outdated or platform-incorrect shortcut.

What it does not catch is the case where the entire command is unknown to the catalog — the plugin GUI menu, the floating palette command, the dynamic submenu the scanner failed to materialize. Those remain AI-suggested.

7.2 Customized environments

User customizations of menus and shortcuts are captured to the extent that the underlying scan can see them.

CUSTOMIZATION TYPE	CAPTURED?
Remapped macOS menu shortcut (e.g., <code>⌘S</code> → <code>⌘⇧S</code>)	Yes — the AX layer returns live bindings
Custom menu items added by an app feature	Yes if standard menu items; no if in floating palettes or web views
DAW command remappings	Logic Pro only — dynamic per-session sync
Custom Keyboard Maestro macros	Yes — fully enumerated at request time
Custom Apple Shortcuts	Yes — fully enumerated at request time
Global hotkeys not tied to menus	No — out of scope

The largest gap is DAW command remappings on non-Logic DAWs. A Cubase user who has remapped their DAW-internal Cubase shortcuts will see those remappings reflected in the Cubase menu/shortcut catalog (because that catalog comes from an Accessibility scan of Cubase’s live menus), but the DAW command library — which drives the MIDI/control-surface bindings, not key bindings — is hardcoded against the documented Cubase command set and does not reflect user remappings.

7.3 Plugin commands

Plugin commands inside a DAW are handled separately from the DAW command library. Cubase plugins are first-class catalog entries; the `cubase_plugin` action type identifies a plugin and a slot, and AI Builder can propose commands like “load Pro-Q 3 on the first free insert.”

For other DAWs (Logic, Studio One, Pro Tools, Ableton, others), plugins are not currently part of the AI-proposable surface. This is an honest limitation of the current implementation, not a permanent design decision.

The architectural split between DAW commands and plugin commands is deliberate. DAW commands are stable, transcribed sets; plugins are per-system, per-installation realities that we discover dynamically. Mixing them would have made the DAW library volatile across users. The cost of the split is that the plugin pathway only extends as far as the plugin manager protocol we can introspect (currently Cubase).

7.4 Windows

On Windows, the keyboard shortcut layer of the Catalog is empty. Win32 menu enumeration produces menu titles and submenu structure, but the OS does not expose keyboard shortcuts in a structured form the scanner can read.

AI Builder is informed of this and adjusts: on Windows, proposed keyboard shortcuts are explicitly marked as AI-suggested, and the user is expected to verify them against their app’s actual binding. The flow, the verification of non-keyboard actions, and the Grounded AI architecture work identically on both platforms — but the *keyboard shortcut layer specifically* is grounded only on macOS.

8. Future direction

This section describes what is planned, what is being researched, and what is explicitly not on the roadmap. It is not a marketing wish list.

Telemetry. Currently no. The privacy posture (section 5) is a deliberate choice. Opening a telemetry channel would let us answer questions like “what is the verification hit rate at population scale” — but at the cost of the no-telemetry contract. The gating decision is whether opt-in, locally-anonymized usage statistics could give us aggregate insight without breaking the contract. We have not built this.

Token-subset matching. Section 3.1 noted that a soft-confirmation rule between the exact-match steps and the unverified case has been discussed but not implemented. If added, it would carry a distinct visual state (not the green Verified badge) and would target cases like Figma’s “Bring to Front” being matched against an AI-emitted “Front.” The risk to manage is false confidence — we do not want a fuzzy match to look like a verification.

Wider plugin coverage. The Cubase plugin pathway is the most elaborate. Extending similar coverage to Logic Pro, Studio One, Pro Tools, and Ableton requires per-DAW work — each DAW’s plugin manager exposes plugin inventory through a different protocol. This is a known gap and a known direction.

Better menu materialization on macOS. The four-stage materialization strategy for dynamic submenus (section 2.2) handles most cases but misses some. Improvements here are mostly engineering effort against known failures, not novel research.

On-device models. Periodically asked. Currently we use Anthropic’s hosted API because the quality of the models matters more than the locality. As on-device models improve and converge on the cloud models in capability, the tradeoff will be revisited. Today, the latency, the quality, and the breadth of catalog reasoning that hosted models offer are not matched by on-device alternatives — and the user’s privacy contract with Anthropic, under their own API key, is a reasonable substitute for fully on-device processing.

9. References

Anthropic API documentation, prompt caching beta — docs.claude.com

macOS Accessibility framework reference — developer.apple.com

MetaGrid Pro AI Builder feature page — metagrid.app/ai-builder

MetaGrid Pro Discourse community — community.metagrid.app

Steinberg Generic Remote specification — referenced for Cubase command library transcription

Apple AppleScript Language Guide — referenced for Logic Pro dynamic command sync

Colophon

This document was produced in May 2026. The numerical claims (DAW command counts, model identifiers, endpoint URLs, token size ranges, cost ranges) are derived from the MetaGrid Pro production codebase at the time of writing. Catalog sizes, model identifiers, and cost estimates are subject to change as the product evolves.

The MetaGrid Pro production codebase is closed-source. The file references used during this document's preparation are listed in an internal appendix maintained alongside this paper.

All claims about privacy and data handling reflect the implementation at the time of writing. Users evaluating AI Builder for high-stakes deployments are encouraged to verify the current behavior independently — the architecture is auditable through network inspection on a running instance.

— Przemysław Mieszkowski, MetaGrid Pro May 2026