# App-in-Skill: A Specification for Pairing Agent Skills with a Local Companion UI

Kelly Peilin Chan
kelly@buda.im

## Abstract

AI agent skills have become remarkably capable: a single skill can read external systems, reason over data, draft content, and execute actions on a user's behalf. Yet the interaction surface for these skills remains a single channel—a chat transcript. For workflows that involve many items, persistent state, human approval, or handoff to a teammate, a scrolling conversation is the wrong surface. The capability of the agent outruns the human's ability to see, steer, and share the work.

We propose App-in-Skill (AiS): a skill that, in addition to exchanging text with the user, can launch a local UI to assist the interaction. The essence is minimal—a skill bundles a small local web app; the app reads and writes local files; the skill performs all real work and external actions. App-in-Skill does not turn a skill into an application and does not replace chat with a UI. The chat and the app are complementary surfaces over one skill, used together rather than as alternatives: chat carries intent, judgment, and exceptions, while the app—running alongside the conversation—carries state, batch review, structured input, and collaboration. The skill remains the single engine doing the work, and a shared file-based handoff keeps the two surfaces in sync.

This paper defines App-in-Skill as a specification, not a single application shape. The specification covers only what is universal to every App-in-Skill: a skill-launched local UI, a file handoff between skill and app, a locking discipline, a pluggable data-provider layer, private configuration, an onboarding state, and a chat-only fallback. Everything else—review queues, dashboards, workspaces, control panels—is a derived app type, a recommended usage pattern rather than a constraint. We present a taxonomy of these app types, a catalog of real applications spanning email triage, content production, SEO, advertising, and CRM, and a pluggable data-provider model that ranges from local files to PostgreSQL, AITable.ai, Notion, and Busabase.

We argue that App-in-Skill enables a new category of software we call daily-disposable software: workflow tools cheap enough to generate on demand, specific enough to fit one operator's exact context, and disposable enough to rebuild when the workflow changes. As this becomes routine, software shifts away from the "big and comprehensive" product toward infinite fragmentation—many small, sharp, single-purpose tools—a shift with direct consequences for the incumbents of the software industry.

Preprint. June 2026. Work in progress.

The core is simple: a skill that can open a window. Everything else follows.

# 1 Introduction: The Pain That Started It

## 1.1 A Capability Without a Surface

Modern AI agents—Claude Code, OpenAI Codex, Cursor, Buda AI, and the growing ecosystem of skills built on them—can do remarkable things from a single natural-language prompt. An agent can triage an inbox, draft replies grounded in a knowledge base, generate a week of blog topics from search data, produce multi-platform versions of an article, hunt for sales leads across the web, or review a release for tone and accuracy.

The capability is real. The problem is the surface. All of this is delivered through one channel: a chat transcript. And a transcript is a poor instrument for most of the work that matters.

## 1.2 Three Pains

Three concrete pains recur across every non-trivial agent workflow.

**Chat fatigue.** When the agent prepares twenty drafts, fifty queue items, or a hundred generated assets, the only way to review them in chat is to scroll. There is no overview, no filtering, no persistent state. Each turn adds to the cognitive load. Users who start out delighted by the agent's speed end up exhausted by the transcript.

**The delegation barrier.** An expert can run a workflow through agent chat because she understands the agent's capabilities, the prompt structure, and the expected outputs. But she cannot hand that workflow to a teammate. The conversation is private and non-transferable: the teammate cannot read the history, does not know what the agent has decided, and cannot participate without starting over. Every chat-only agent workflow has a ceiling of one person.

**The trust gap.** The natural response to chat fatigue is to give the agent more autonomy—"just do it all." But silent execution removes the human's ability to see what is about to happen. Users will not authorize an agent to send email, charge accounts, or mutate records when they cannot inspect the proposed actions first. Without a place to show its work, the agent is stuck between exhausting the user and acting blind.

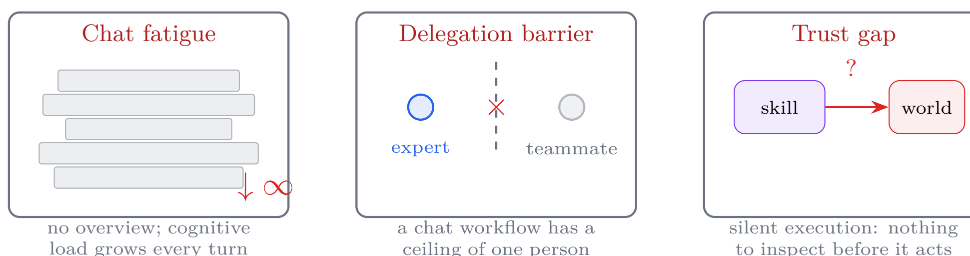


Figure 1: The three pains of running agent workflows through chat alone. Chat fatigue: a transcript has no overview and grows with every turn. Delegation barrier: a private conversation cannot be handed to a teammate. Trust gap: silent execution leaves nothing to inspect before the agent acts. App-in-Skill addresses all three by giving the workflow a visible, shareable surface.

### 1.3 The Observation

Underlying all three pains is a single observation, repeated in practice: once people enter a workflow through chat, they no longer want to operate it through chat. Chat is an excellent entry point—it is the fastest way to describe intent and adjust direction. But for ongoing operations, people want a stable surface: something they can look at, filter, click, and return to. It is the oldest instinct in any working conversation: when talk alone stops keeping up, two people reach for a whiteboard—not to end the conversation, but to give it a surface to work on. People want that whiteboard beside the chat, not a longer transcript.

The agent is close to omnipotent; the human is the bottleneck. And the bottleneck is not the human's judgment—it is the absence of an interface through which that judgment can be applied efficiently and shared with others.

### 1.4 The Proposal, in One Sentence

This paper's proposal is deliberately small: let a skill open a window. When you talk to an AI through a skill, the skill should be able to respond not only with text but, when the work calls for it, by launching a local UI that assists the interaction—showing state, accepting structured input, and saving your decisions—while the skill continues to own all reasoning and real-world action.

We call this pattern App-in-Skill. The remainder of the paper defines it precisely (Section 2), specifies the minimal universal contract every App-in-Skill must satisfy (Section 3), describes the pluggable data-provider layer that lets the same app run on local files or a cloud database (Section 4), presents a taxonomy of app types and a catalog of applications to show how broad the pattern is (Sections 5–7), argues for the emerging category of daily-disposable software (Section 8), and explains how to build one (Section 9).

### 1.5 Audience and Intent

This paper is written for skill and agent builders. Its goal is not only to define a pattern but to show that the space of useful App-in-Skills is enormous and largely unexplored—and to invite builders to claim parts of it. App-in-Skill is not a product or a proprietary framework; it is a convention thin enough to implement in an afternoon and general enough to fit almost any human-in-the-loop agent workflow.

## 2 What Is App-in-Skill

### 2.1 Definition

> App-in-Skill (AiS) is a design pattern in which an AI agent skill bundles a small local web application as an optional UI. When the skill is invoked, it may respond with text, or it may launch (or reuse) the local app and direct the user there. The app reads and writes local files; the skill owns all reasoning, external reads and writes, and safety gates. A file-based handoff connects the two.

The mental model is one sentence: the skill can open a window to help you work, instead of forcing the whole conversation through text.

### 2.2 The Two Channels

An App-in-Skill gives the user two channels to the same skill:

The text channel (chat). Best for describing intent, asking questions, adjusting direction, handling exceptions, and reconfiguring the workflow. This channel always exists.

The UI channel (the app). Best for seeing state, reviewing many items, editing drafts, making structured decisions, and collaborating. This channel is launched on demand when the work benefits from it.

The two channels are not redundant. They are complementary surfaces over one skill—chat is the talking, the app is the whiteboard the talk reaches for—and they stay synchronized through shared local files. A user can adjust the workflow in chat, switch to the app to process a batch, and return to chat to handle an edge case—without losing state in either place.

### 2.3 The Division of Labor

The defining structural property of App-in-Skill is a clean separation:

- The skill owns capability. It reads external systems, reasons, drafts, decides what to propose, and executes approved actions. It is the only component that touches the outside world.
- The app owns visibility. It renders what the skill prepared, accepts the user's input, and persists it locally. It never performs an external side effect.

We call this the boundary, and it is the one invariant that must never be violated.

### 2.4 The Boundary

Let $\mathcal{E}$ be the set of all external effects: sending email, calling an API, mutating a database, deleting files, charging an account, posting to a remote service. The boundary states:

> Boundary Invariant. Every external effect $e \in \mathcal{E}$ is initiated by the skill, never by the app. The app's world is confined to reading and writing local handoff files and serving a local UI.

The boundary exists for two reasons. First, safety: if the app could send email or mutate records directly, the human approval gate would be bypassed. Keeping all external effects in the skill means there is always exactly one place where authorization is enforced. Second, simplicity and testability: an app that only reads and writes local files is trivial to build, has no credentials, and cannot cause external harm if it has a bug. The hard logic lives in one testable place.

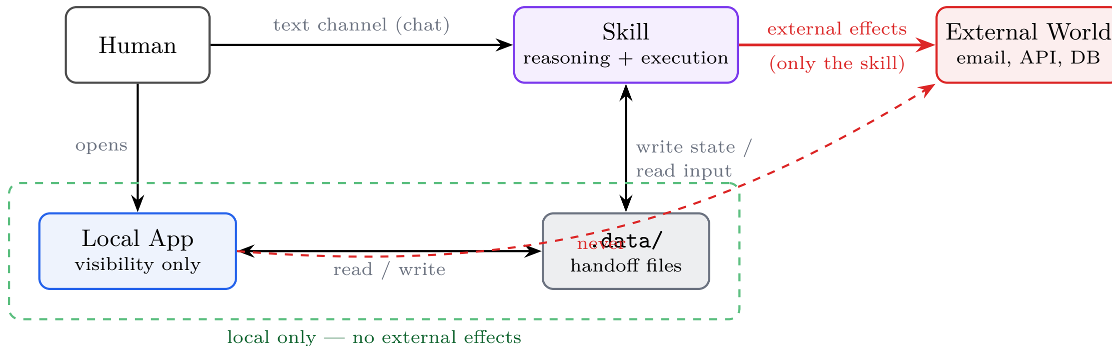


Figure 2: The two channels and the boundary. The human reaches one skill through two channels: the text channel (chat) and the local app. The app and the handoff files (`.data/`) live in a local-only zone; the skill is the only component that produces external effects (red). The app never reaches the external world.

### 2.5 What App-in-Skill Is Not

Not a SaaS product. No cloud tenancy, no account system, no uptime guarantee. An App-in-Skill is a local tool for one operator or small team. It needs no deployment to be useful.

Not in-chat widgets. Some agents render cards or forms inside the transcript. App-in-Skill launches a real, independent local web app that the user navigates to and that persists across chat sessions.

Not an agent framework. It does not prescribe how the agent reasons or which model it uses. It is a thin convention for attaching a UI to whatever skill you already have.

Not a replacement for chat. Chat remains primary for intent and judgment. The app is launched only when the work benefits from a visual interface, and every App-in-Skill must still support a chat-only fallback.

Not tied to one app shape. App-in-Skill does not mandate review queues, state machines, or any particular layout. Those are usage patterns (Section 5), not part of the specification.

### 2.6 The Smallest Possible App-in-Skill

To make the minimalism concrete: the smallest valid App-in-Skill is a skill whose instructions say "when asked to review X, generate a JSON file describing the items and start the local viewer," plus a single-file local server that reads that JSON and renders it, plus a way for the user to write a decision back. No database, no dependencies, no framework. Everything beyond this—schemas, states, dashboards, cloud data—is optional elaboration that the specification permits but does not require.

## 3 The Specification

This section defines what is universal to every App-in-Skill, regardless of what the app does. The specification is intentionally small. It says nothing about review queues, workflow states, or dashboards—those are derived patterns (Section 5). It specifies only the seven mechanisms that any skill-launched UI needs in order to be safe, recoverable, and usable.

### 3.1 Spec 1: A Skill-Launched Local UI

An App-in-Skill bundles a local web application that the skill can start on demand. Requirements:

- The app is served over local HTTP on `127.0.0.1`; it is never exposed externally.
- The server should use only platform built-ins where possible (for example Node's `node:http`, `node:fs/promises`), so that the app has zero or minimal dependencies and can run anywhere the skill runs—including inside a cloud drive or sandbox.
- The launcher prefers a stable port range (for example 3000–4000). If a port is occupied, it probes a health endpoint to decide whether to reuse the existing instance or pick the next port, and it always reports the actual URL it bound to.
- Launch is idempotent: invoking the skill twice should reuse the running app, not spawn duplicates.

The app is the optional channel: the skill decides, per invocation, whether to answer in text or to launch/refresh the app and send the user there.

### 3.2 Spec 2: A File Handoff

The skill and the app communicate through local files, not a live connection. This is what lets the two channels have independent lifecycles—the agent runs intermittently; the app stays usable in between.

The handoff has two directions, and every App-in-Skill must define both:

Skill → App (state files). The skill writes files describing what it prepared: items, drafts, metrics, status, options. The app reads these to render the UI.

App → Skill (input files). The app writes files capturing what the user did: decisions, edits, comments, selections. The skill reads these before it acts.

The specification fixes only the shape of the contract, not the schema of the data. A concrete App-in-Skill chooses field names and structure appropriate to its app type (Appendix A gives a worked example for a review app). The universal requirements are:

1. State and input live in separate files, so the skill never overwrites the user's input and vice versa.

2. Files are written atomically (write to a temporary file, then rename), so a reader never observes a partial write.
3. All workflow state lives on disk, never only in process memory, so the workflow survives the agent pausing or the browser closing.

These files belong in a directory named `.data`, not `.cache`. The distinction is deliberate: a cache is disposable and regenerable, but the handoff files are neither. The user's decisions and the execution audit trail cannot be recreated by re-running the agent—they are genuine workflow state. Naming the directory `.cache` would misrepresent recoverable, authoritative data as throwaway, and would invite tooling to delete it. We name it `.data` to reflect that it is the data.

The handoff carries a review model. The files are the local serialization of a single review model, shared verbatim with the recommended cloud provider (Section 4), so that one vocabulary holds across every backend. A change request is what the agent prepared—a proposed creation or edit awaiting review (the items in `current_batch.json`); an operation is one change inside it, carrying its before→after fields; a review is a human verdict (`decisions.json`); a merge applies an approved change to the canonical artifact (`execution_report.json`); a comment with an `@ai` mention, or an item left in the `changes_requested` state, becomes an agent task. The verdict verbs are `approve`; `request_changes`, which moves the item to `changes_requested` and returns it for re-review after the agent revises—a loop, not a rejection; `revise`, the human's own edit saved as a new version; and `block`. Because this vocabulary is provider-neutral, the file handoff and a Busabase backend are two serializations of the same model (Table 1); moving between them is a configuration change (Section 4), not a rewrite.

| Term | Meaning | Local file | Busabase |
|---|---|---|---|
| change request | agent's proposed creation/edit | `current_batch.json` | `change_request` |
| operation | one change, before → after | item fields + draft | `operation` |
| review | a human verdict | `decisions.json` | `review` |
| merge | apply approved change to canonical | `execution_report.json` | `merge` |
| comment | note; `@ai` asks the agent to act | note field | `comment` |
| agent task | queued work (`changes_requested`/`@ai`) | `agent_tasks.json` | `/agent/tasks` |

Table 1: The review model, one vocabulary across providers. The local file handoff and Busabase are two serializations of the same terms; `lib/data-provider/` (Section 4) selects which.

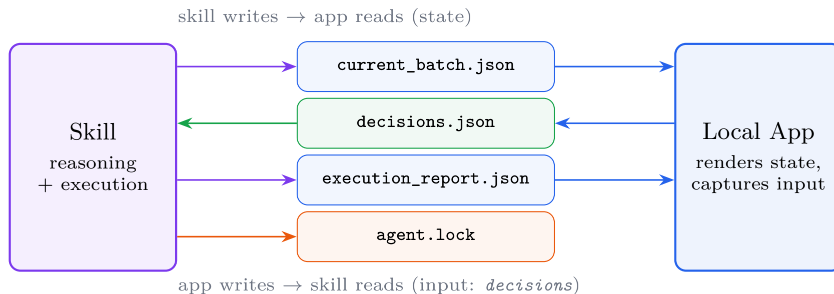


Figure 3: The file handoff. State files (blue) flow skill→app; the input file (green, `decisions.json`) flows app→skill; `agent.lock` (orange) is held while the skill writes or executes. The two processes never hold a live connection—the files are the contract, which is what makes the workflow recoverable across pauses and restarts.

### 3.3 Spec 3: Locking

Because the skill and the app are separate processes that both touch the handoff files, the specification requires a lock to prevent races (for example, the user editing a decision while the skill is mid-execution).

The mechanism is a single lock file written by the skill before it generates or executes, and removed in a `finally` step afterward. While the lock exists:

- The app server rejects write requests (returning a "locked" status) and disables editing controls in the UI, while continuing to display read-only state.
- The skill refuses to begin a second concurrent operation.

The lock carries an owner, a human-readable message, and a start timestamp; the timestamp lets a stale lock (from a crashed run) be detected. File-based locking is sufficient because exactly two same-machine processes participate and the critical sections are short.

### 3.4 Spec 4: A Pluggable Data Provider

Where does the handoff data live? By default, in local files. But the specification requires that data access go through a data-provider interface rather than hard-coded file reads, so that the same app can later be backed by a database or cloud service without rewriting the UI or the scripts. This layer is important enough that Section 4 is devoted to it.

### 3.5 Spec 5: Private Configuration

Most useful skills need private context: account credentials, an operator profile, brand and product information, canonical URLs, a knowledge base, style rules, and risk policies. The specification requires that this configuration be:

- Layered and private: resolved from environment variables and gitignored local config files, never committed; secrets only in environment variables, referenced from config by name.
- Read through the data provider, so the app and the execution scripts always agree on what is configured.
- Exposed only as a safe summary: the UI may show which accounts and sources are configured, but never secret values—secrets appear only as boolean "configured / not configured" flags.

### 3.6 Spec 6: Onboarding as the Default Initial Phase

Onboarding is not a fallback for missing configuration—it is the default initial phase of every App-in-Skill. A freshly installed skill always begins in onboarding: before it does any real work, it must learn its operating context from the user. This is the one phase every spec-compliant skill passes through on first run.

The phase is a loop with a latch:

1. Gate. On every invocation, the skill checks for an onboarding completion marker (`app/.data/onboarding.json`). While the marker is absent or `completed:false`, the skill is in onboarding and performs no external reads or writes.
2. Ask and configure. The app shows a setup wizard and the skill asks the user—turn by turn—for the configuration that makes it useful: accounts and credentials (created as local config/env files, never pasted into chat), operator profile, brand, canonical URLs, style, risk policy, and safe knowledge sources.
3. Validate. The skill validates as it goes (required secrets present, config parses, accounts reachable when checkable) and keeps prompting until the configuration is complete and the user confirms "done."
4. Latch. On confirmation, the skill writes the completion marker. Like a lock that, once set, lets work begin, the marker latches the transition: only once it exists (and the config still validates) does the skill enter normal operation.

If required config or secrets later go missing or fail validation, the skill drops back to onboarding rather than running with a broken context; onboarding can also be re-run deliberately to reconfigure. The guiding principle throughout: the user creates local config/env files and confirms when ready; they never paste credentials into chat or the UI.

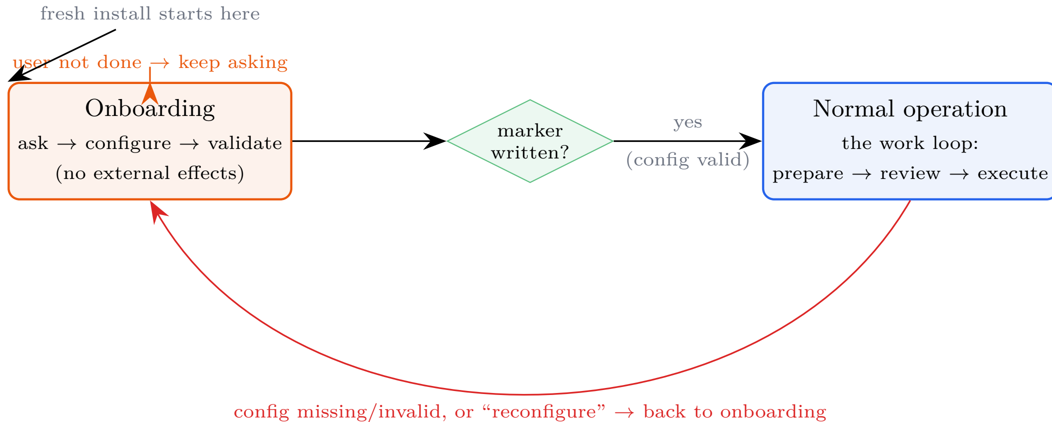


Figure 4: The lifecycle of every App-in-Skill. A fresh install begins in onboarding, which loops—asking the user and configuring—until the user confirms setup is complete and the configuration validates. Confirmation writes the completion marker (`onboarding.json`), which latches the transition into normal operation. If the configuration later breaks or the user chooses to reconfigure, the skill returns to onboarding rather than running with a broken context.

### 3.7 Spec 7: Chat-Only Fallback

The UI is an assist, not a requirement. Every App-in-Skill must remain operable in pure text. When the user asks for chat-only mode—or when no display is available—the skill presents items and accepts approvals directly in the conversation, using the same handoff files as the backing store. The app being unavailable must never block the work.

### 3.8 Summary of the Specification

| # | Mechanism | Why it is universal |
|---|---|---|
| 1 | Skill-launched local UI | the defining feature: a skill that can open a window |
| 2 | File handoff | decouples agent and app lifecycles; recoverable |
| 3 | Locking | safe concurrency between two processes |
| 4 | Pluggable data provider | local now, database/cloud later, same UI |
| 5 | Private configuration | real skills need private context, kept safe |
| 6 | Onboarding | the default initial phase; latched by a completion marker |
| 7 | Chat-only fallback | the UI assists; it must never be mandatory |

Table 2: The seven mechanisms of the App-in-Skill specification. Note what is absent: no workflow states, no required schema, no mandated layout. Those are derived patterns, covered in Section 5.

# 4 Data Providers: From Local Files to the Cloud

## 4.1 Why a Provider Layer

The most consequential design decision in App-in-Skill is that data access is polymorphic. The default backing store is local files—this is what makes an App-in-Skill zero-dependency, private, and runnable anywhere. But the same app must be able to graduate to a database or cloud service when the workflow needs persistence, sharing, or scale, without rewriting the UI, the scripts, or the skill logic.

This is achieved with a data-provider interface (in the reference implementation, `lib/data-provider/`). All components—the app server, the generation and execution scripts, onboarding, and the UI's configuration summary—read through this interface rather than importing a file reader directly. A provider is selected by an environment variable (for example `<SKILL>_DATA_PROVIDER=local`).

Listing 1: The data-provider interface (reference shape)

```
// lib/data-provider/index.mjs
// Every provider implements the same minimal interface:
//   getConfig()             -> resolved private configuration
//   getSafeConfigSummary()  -> sanitized summary for the UI (no
    secrets)
//   isConfigured()          -> readiness check
//   readState() / writeInput()  -> handoff access

const PROVIDER = process.env.SKILL_DATA_READER ?? 'local';

export async function createDataReader() {
  switch (PROVIDER) {
    case 'local':     return new LocalFileReader();    // default
    case 'postgres':  return new PostgresReader();
    case 'aitable':   return new AITableReader();
    case 'notion':    return new NotionReader();
    case 'busabase':  return new BusabaseReader();
    default: throw new Error(`Unknown data provider: ${PROVIDER}`);
  }
}
```

The interface is deliberately small so that new providers are cheap to add. The first and default implementation is `local`; the others are added as a workflow's needs grow.

## 4.2 The Provider Spectrum

| Provider | Best for | Trade-off |
|---|---|---|
| Local files | single operator, private, offline, fastest start | no sharing, no remote access |
| PostgreSQL | self-hosted teams, full SQL control | you operate the database |
| AITable.ai | visual-database teams, spreadsheet-like editing + 6,000+ app integrations | hosted service, API limits |
| Notion | doc-centric teams, content living in Notion | not built for high-volume rows |
| Busabase | AI-generated content needing a review-to-canonical pipeline | cloud dependency |

Table 3: The data-provider spectrum. The default is local files; every other provider is opt-in and reached through the same interface.

The key property is that moving along this spectrum is a configuration change, not a rewrite. An operator can prototype a workflow on local files in an afternoon, and—if it proves valuable and needs to be shared with a team—switch the provider to PostgreSQL or a managed service by changing one environment variable and supplying credentials. The user-facing workflow is unchanged.

### 4.3 Busabase: A Natural Fit

Among cloud providers, Busabase deserves special mention because its model maps onto App-in-Skill almost exactly. Busabase gives AI-generated articles, assets, and structured records a single review Inbox before they become canonical records. An operator can approve an item, request changes, and keep a full audit trail; approved items are then shipped downstream as trusted, canonical content.

This is the App-in-Skill loop expressed as a data service. Where a local App-in-Skill keeps "what the agent prepared" and "what the human approved" in local handoff files, Busabase keeps them as Inbox records and canonical records with an audit trail between them. The consequences:

- Shared review. Multiple operators see the same Inbox; decisions and audit history are centralized rather than trapped on one machine.
- Provenance by default. The review-to-canonical transition is recorded, so downstream consumers know which records were human-approved and when.
- A real backend without building one. The team gets persistence, multi-user access, and an audit trail without standing up and operating their own database—the App-in-Skill stays thin, and Busabase is the system of record.

In other words, an App-in-Skill backed by the local provider is a personal tool; the same App-in-Skill backed by Busabase becomes a team system of record for human-approved AI output, with no change to the app the operators use. We recommend Busabase as the default cloud provider for App-in-Skills whose output should become trusted, shared, canonical content.

The mapping is exact because the two providers share one vocabulary (Table 1): change request, operation, review, the verdicts `approve/request_changes/block`, merge, comment, and agent task. A local App-in-Skill and a Busabase-backed one differ only in where these records live and where the lock is held (a local file lock vs. a shared lock in the backend)—not in what the records are called or how the review loop behaves. This is why the file handoff of Section 3 is, precisely, the offline serialization of the Busabase review model.

### 4.4 The Local-to-Cloud Continuum

Taken together, the provider layer turns the local-versus-cloud question from an architectural fork into a slider:

1. Start local. Zero dependencies, private, instant. Prove the workflow.
2. Stay local if the workflow is personal and the operator is happy.
3. Graduate to a cloud provider (Busabase, Postgres, AITable.ai, Notion) when the workflow needs sharing, persistence, provenance, or scale.

The same App-in-Skill spans the whole continuum. This is what makes the pattern viable both for a one-person daily tool and for a team's shared operating UI.

## 5 App Types: One Specification, Many Shapes

The specification (Section 3) says nothing about what the app looks like or how the workflow is organized. That freedom is intentional. An App-in-Skill can be a review queue, a dashboard, a workspace, a control panel, or a collaboration space. This section presents these as a taxonomy of app types—recommended patterns, not requirements—and shows how they differ. A builder picks the type that fits the work, or combines several.

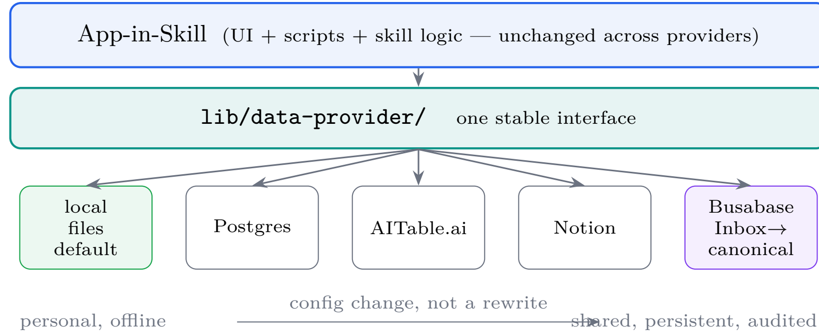


Figure 5: The data-provider spectrum. The app, scripts, and skill logic are written once against a single `lib/data-provider/` interface; the backing store is selected by configuration. The default is local files; cloud providers (Postgres, AITable.ai, Notion, Busabase) are reached through the same interface, so moving from a personal tool to a shared, audited team system is a configuration change rather than a rewrite.

## 5.1 The Taxonomy

| App Type | The user is... | Data shape | Stateful? |
|---|---|---|---|
| Review queue | approving / editing items | list of items + decisions | yes |
| Dashboard | monitoring | metrics, status, reports | no (read-mostly) |
| Workspace | creating / editing | drafts, assets, collections | partly |
| Control panel | configuring / launching | parameters, modes, triggers | no |
| Collaboration | handing off / deciding together | shared items + actors | yes |

Table 4: A taxonomy of App-in-Skill types. "Stateful" indicates whether items move through a workflow lifecycle. The review queue is one type among several—not the definition of the pattern.

## 5.2 Review Queue

The most common type. The agent prepares a batch of items, each with a proposed action and a draft; the human approves, edits, blocks, or comments; the skill executes the approved ones. Email triage, customer support, content moderation, and release approval are review queues.

**Recommended pattern: stage-orientation.** For review queues, we recommend organizing the primary navigation by workflow stage rather than by entity. Traditional software is entity-oriented—a list of emails, a list of tickets—and forces the user to inspect each item to learn its status. A review queue is better organized by the question the reviewer actually asks: what needs my attention, and at what stage? A practical set of stages is `needs_review` (the agent needs input before it can propose), `to_approve` (a proposal is ready), `approved` (the human confirmed; the skill may execute), `done`, and `blocked`. Categories and risk labels become badges within a stage, not top-level navigation.

We stress that this five-stage model is a recommended usage pattern for review queues, derived from the specification—it is not part of the specification, and other app types do not use it. Appendix A gives an example handoff schema for this pattern.

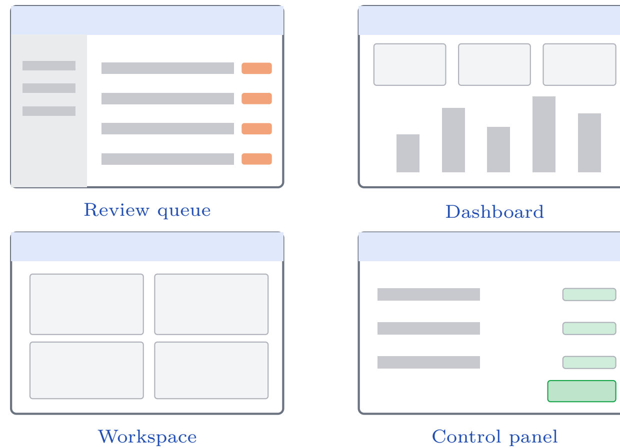


Figure 6: Four common App-in-Skill shapes (schematic wireframes). A review queue navigates by stage with a list of items and badges; a dashboard shows metric tiles and charts (read-mostly); a workspace is a bench of editable drafts/assets; a control panel exposes settings and a run trigger. The same specification (Section 3) underlies all four—only the layout and the handoff schema differ.

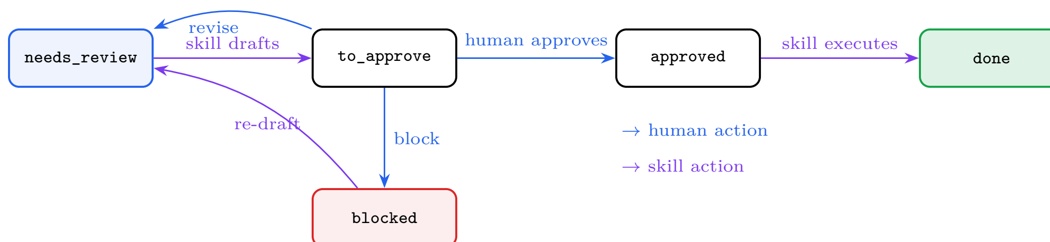


Figure 7: Recommended stage flow for the review-queue type (not part of the specification; other app types do not use it). Items advance through stages by alternating skill actions (purple) and human actions (blue). The reviewer navigates by stage—"what needs my attention, and at what stage?"—rather than by entity, so the queue answers the operator's real question at a glance.

### 5.3 Dashboard

A read-mostly view over agent-generated data: metrics, task progress, execution reports, analytics. The agent periodically writes a state file; the app renders charts, counts, and tables. There is no approval lifecycle—the user is monitoring, not deciding. Examples: an SEO performance dashboard, an ad-spend summary, a pipeline-status monitor, a lead-generation progress view. Dashboards use the same file handoff and locking as any App-in-Skill, but their handoff data is metrics rather than reviewable items.

### 5.4 Workspace

A space for creating and refining content with the agent: a draft box, an asset collection, a content-repurposing bench. Items here are works in progress rather than queue items awaiting a yes/no. A workspace often organizes content by creative stage (idea, draft, in progress, finished collection) and supports inline editing. Examples: an article drafting workspace, a short-video production bench, a portfolio of finished pieces alongside works in progress.

### 5.5 Control Panel

A panel for steering the agent rather than reviewing its output: launching batches, choosing modes, tuning parameters, scheduling, and running dry-runs. The handoff here is configuration and triggers flowing app-to-skill, and run status flowing back. Examples: a campaign launcher, a crawl configurator, a "run this workflow now / on schedule" console.

### 5.6 Collaboration

A shared space where more than one human participates around the agent's output: shared decisions, handoffs between roles, multi-stakeholder approvals. This type is where the data provider usually moves off local files (Section 4), because the shared state must be visible to several people. Examples: a video-production pipeline where an editor, a reviewer, and a publisher each act at different stages; an SEO workflow where an outsourced writer and an in-house reviewer collaborate.

### 5.7 Two Kinds of Work: Edit-to-Canonical vs. Approve-an-Action

Cutting across the taxonomy is a deeper distinction about what a review actually decides. It determines how much a canonical-content backend like Busabase (Section 4) is worth, and it is the reason the review model fits some skills natively and others only partially.

Edit-to-canonical. The change under review is a new version of an artifact the workflow owns—an article, a record, a piece of copy. "Merge" writes the approved version back as the canonical artifact. Here the review model's full machinery pays off: versioned commits, before→after field diffs, and the `request_changes` revision loop, because there is real content being refined. kelly-content is the canonical example—a draft per channel, reviewed and merged into published content.

Approve-an-action. The change under review is a proposed action on an external system the workflow does not own—archive this email, send this reply, charge this account. The reviewable item is a read-only observation (the email), not an editable record; "merge" means perform a side effect and log it, not write a canonical version. kelly-email triage is the example: the body is immutable, and the decision is an action, not an edit.

For approve-an-action work, the canonical-content superpowers (commits, diffs) are mostly idle, so for a single operator a cloud backend degenerates into an over-powered cache. Its value reappears at team scale: the shared store becomes the one place where everyone sees the same queue and leaves an audit trail of who approved what. There, a backend like Busabase is not a cache but a multi-operator system of record for approvals.

Team scale is a provider switch, not a new service. Going from one operator to a team does not require standing up a runner service. The skill stays exactly what it was—a single-machine skill that an operator runs, ingesting observations and performing approved side effects itself. The only thing that moves to the cloud is the system of record: the handoff state—the queue, the decisions, the execution log, and the lock—lives in the shared backend (Busabase) instead of local files. Because the lock is now shared, it serializes the whole team's writes and executions exactly as it serialized one operator's, so coarse coordination comes for free and fine-grained per-item claiming or optimistic concurrency is unnecessary at this scale. The boundary (Section 2) is unchanged: the shared store and the UI never touch the outside world; whichever operator's skill is running does, after approval. The step from personal tool to team system is therefore a provider switch—local files → shared backend—not a new architecture.

Hybrids are normal. Many workflows mix both. kelly-email is approve-an-action for triage but edit-to-canonical for the one part that is genuine content—the reply draft, which benefits directly from the revision loop. The clean design backs only that slice with the canonical model and leaves triage as an action queue. The litmus test for whether to reach for a

canonical backend at all: does merge produce a reusable canonical artifact, or merely a "done" record?

### 5.8 Types Are Composable

These types are not exclusive. A content workflow might present a workspace for drafting, a review queue for approval, and a dashboard for performance—three views in one app, or three apps that share a data provider. The taxonomy is a vocabulary for thinking about shape, not a set of boxes a builder must choose between. What unifies them is only the specification: a skill that can launch a local UI, a file handoff, a lock, a data provider, private config, onboarding, and a chat fallback.

## 6 The Human-AI Collaboration Loop

### 6.1 Overview

When an App-in-Skill is used for work that flows through the app—rather than a one-shot dashboard render—the interaction follows a recurring cycle between human and agent. We call this the Human-AI Collaboration Loop (HACL): a five-step cycle that replaces the informal turn-taking of chat with a structured, recoverable, and observable interaction model. The loop describes the typical usage of an App-in-Skill; it is not a mechanism the specification imposes. A pure dashboard may use only steps 1–2, while a review queue uses all five.

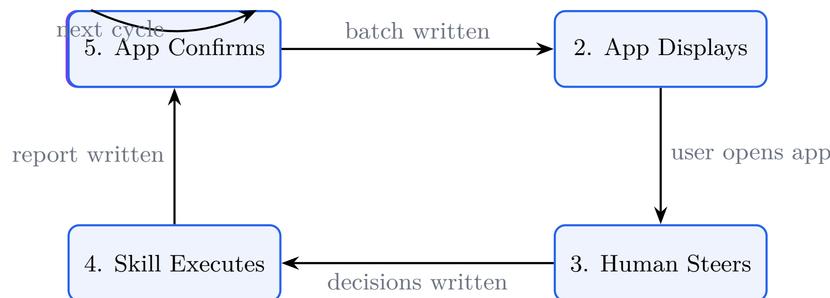


Figure 8: The Human-AI Collaboration Loop. Purple steps are skill-executed; blue steps are human-facing. All transitions are mediated by the file handoff.

### 6.2 Step 1: Agent Prepares

The skill runs its generation cycle: reads external sources, reasons over the data, produces drafts and recommendations, assigns workflow states and risk labels, and writes the result to `current_batch.json`. During this step, the lock is held.

The agent's output is not just raw data. It is structured guidance for the human: proposed actions, reasons, risk assessments, suggested replies. The agent's job at this step is to reduce the cognitive load of human review, not just to dump information.

### 6.3 Step 2: App Displays

The app reads `current_batch.json` and renders the batch in a visual interface. The user sees:

- The number of items in each workflow state
- A filterable list view with risk badges and category labels
- A detail view with the agent's proposed action, reason, and draft
- Controls appropriate to the current state of each item

The app also reads `decisions.json` to show any previously-saved decisions, supporting resumption after a break.

### 6.4 Step 3: Human Steers

The human interacts with the app to:

- Approve items where the agent's proposal is correct
- Revise items where the draft needs editing (the human's own edit, saved as a new version)
- Request changes on items the agent should redo — the item moves to `changes_requested` and is enqueued as an agent task; after the agent revises, it returns to `needs_review` for re-review (a loop, not a rejection)
- Block items that cannot proceed
- Comment on items for the agent's context in the next cycle; an `@ai` comment also enqueues an agent task

The `request_changes` verb is what makes the loop iterative rather than one-shot: instead of the human re-typing a prompt in chat, the request and its reason ride the handoff as an agent task, and the same item cycles through review until it is approved or blocked.

All human input is saved immediately to `decisions.json`. The human can close the browser, pause for lunch, return tomorrow—their decisions persist.

Critically, the human can also return to chat and say "change #2 to block" or "approve everything except the flagged items." The stable per-batch references (e.g., `Review #2`) in the UI correspond directly to item IDs in the batch, making chat references unambiguous.

### 6.5 Step 4: Skill Executes

When the user signals readiness (by invoking the skill again in chat, or clicking an "Execute" button that triggers the skill), the skill:

1. Acquires the lock
2. Re-reads `decisions.json` from disk (never from memory)
3. Verifies that approved items have complete decision records
4. Executes external actions for approved items in sequence
5. Writes per-item results to `execution_report.json`
6. Releases the lock

The re-read from disk is critical: the execution step must act on the user's final state, not a cached version from the last generation cycle.

### 6.6 Step 5: App Confirms

The app reads `execution_report.json` and shows the user what happened: which items were sent, which were blocked, which had errors. Items transition to `done` or remain in `blocked` state.

The user can review the report, follow up in chat on blocked items, and initiate the next cycle. The loop repeats.

### 6.7 Loop Teachability

A key property of the HACL is that it is teachable. A user who begins in chat can gradually shift their workflow to the app as they gain familiarity. A user who learns the batch references (`Review #1`, `Review #2`) can navigate both surfaces interchangeably. An expert can shape the workflow through chat; a teammate can participate through the app without understanding the agent conversation.

This teachability is the reason App-in-Skill scales from personal automation to team workflows without adding infrastructure.

### 6.8 Formal Properties of the HACL

Proposition 1 (Progress). In a well-formed HACL, every item in `approved` state will eventually transition to `done` or `blocked`, provided the skill is invoked for execution at least once.

Proposition 2 (Safety). In a well-formed HACL, no external effect is produced without a corresponding decision record in `decisions.json` with `action: "approve"` and a non-null `decided_at` timestamp.

Proposition 3 (Recoverability). For any interrupted HACL execution, the state can be recovered by reading the handoff files. No state is held exclusively in process memory.

These properties hold for the review-queue pattern, where "approve" is the gating decision. Other app types adapt the same loop with their own gating semantics, but the Safety property—no external effect without a recorded human decision—is universal and follows directly from the Boundary Invariant of Section 2.

## 7 A Catalog of Applications

The purpose of this section is to make one point vivid: the space of useful App-in-Skills is enormous. Almost any workflow where an agent prepares something and a human needs to see, steer, or approve it is a candidate. Below is a catalog drawn from real operating needs, grouped by domain. Each entry names the natural app type (Section 5) and the typical data provider (Section 4).

### 7.1 Communication and Support

Email triage (kelly-email). The agent reads the inbox, classifies messages, drafts replies grounded in a knowledge base, and flags items involving money, legal, or account risk. The operator reviews a stage-oriented queue and approves, edits, or blocks. Type: review queue. Provider: local files (personal) or cloud (shared support).

Customer support. The same shape as email, but multi-operator and tied to a shared inbox and knowledge base. An expert configures tone and risk policy in chat; associates operate the queue. Type: review queue / collaboration. Provider: cloud.

Omnichannel reply. One queue spanning email, Discord, and other channels—a "catch-all" queue for outbound messages awaiting approval. Type: review queue. Provider: cloud.

### 7.2 Content Production

Article / blog review. An in-house or outsourced writer (or the agent itself) produces a draft; the reviewer approves the topic, then the draft, then the published metadata. Especially useful where the writer and the reviewer are different people. Type: workspace + review queue. Provider: cloud (collaboration).

Video production (video-maker). Topics are selected (with search-volume evidence), drafts are produced, short and long versions are rendered, and titles and descriptions are reviewed before publishing. A reviewer can see, at a glance, which videos are at which stage and where each is stuck. Type: workspace + collaboration. Provider: cloud.

Multi-platform repurposing (post-maker). One article becomes platform-specific versions—a long-form post, a short social thread, a video script—each reviewed before it ships. Type: workspace. Provider: local or cloud.

### 7.3 Growth and Marketing

SEO keyword review. The agent proposes keywords with volume and CPC data (often from a spreadsheet); the reviewer accepts or rejects each for a landing page or content brief. Type: review queue. Provider: AITable.ai / cloud.

Content calendar. A scheduled agent proposes "write these blog topics this week" from search-rank data; the operator approves the week's plan. Type: control panel + review queue. Provider: cloud.

Advertising (ads). The agent drafts a campaign plan (budgets, audiences, creatives); the operator reviews and approves before anything goes live. Type: review queue + control panel. Provider: cloud.

Analytics dashboard. A read-mostly view over GA, ad spend, and campaign performance, refreshed on a schedule. Type: dashboard. Provider: cloud.

### 7.4 Sales and CRM

Lead generation. A long-running browser agent collects leads (for example, startups with public contact addresses); the operator reviews and qualifies the harvested list. The dashboard shows progress over a multi-hour crawl. Type: dashboard + review queue. Provider: cloud.

Outreach / link exchange. The agent drafts personalized outreach to partners; the operator approves each message before it is sent. Type: review queue. Provider: cloud.

CRM operations. Record updates, follow-up scheduling, and pipeline review—the same review-and-approve shape applied to customer records. Type: review queue. Provider: Busabase / Postgres.

### 7.5 Internal Operations

Meeting follow-up (meetup). After a meeting, the agent extracts action items and drafts follow-up messages; the organizer reviews before sending. Type: review queue. Provider: local or cloud.

Release management. The agent prepares release notes, email subjects, and summaries; the operator reviews copy across channels before a release ships. Type: review queue. Provider: cloud.

### 7.6 The Pattern Behind the Catalog

Read together, the catalog reveals a simple selection rule:

> Any workflow that suits a brief-then-review flow—the agent prepares a brief, draft, plan, or batch; a human reviews and steers; the skill executes—is a candidate for App-in-Skill.

This rule covers a striking fraction of knowledge work. Where the workflow is common enough, the App-in-Skill can be published and reused (email triage generalizes well); where it is idiosyncratic, it can be built for one operator's exact context. The breadth is the point: the same thin specification, instantiated again and again, covers an open-ended long tail of real needs that no single product could serve.

## 8 Daily-Disposable Software

### 8.1 A New Category

The catalog of Section 7 hints at something larger than a collection of tools. It points to a new category of software, which we call daily-disposable software (rìpāo xíng ruǎnjiàn): workflow applications that are cheap enough to generate on demand, specific enough to fit one operator's exact context, and disposable enough to rebuild when the workflow changes.

A daily-disposable application is not built for a market. It is built for a moment—one operator, one workflow, one phase of a business's evolution. It can be adapted overnight by describing the change to an agent. It carries no roadmap, no backlog, no migration burden. When the workflow changes, the app is regenerated rather than maintained.

### 8.2 Why It Is Possible Now

Daily-disposable software was not feasible before, because building any application—even a small one—required engineering effort that only paid off if the application served many users for a long time. Two developments change the economics:

1. The agent generates the application layer. The cost of producing a working app collapses when an agent writes it from a description. App-in-Skill keeps that app deliberately thin—a local UI over a file handoff—so the agent can generate and modify it reliably.
2. The data layer is pluggable, not bespoke. Because the data-provider interface (Section 4) ranges from local files to managed cloud services, a workflow needs no custom backend. "Connect a managed database and you are done" removes the last expensive piece.

Together these mean the marginal cost of a workflow-specific tool falls to roughly the cost of a conversation. When software is that cheap to make, it becomes rational to make it for workflows that were previously too small, too idiosyncratic, or too fast-changing to justify any product.

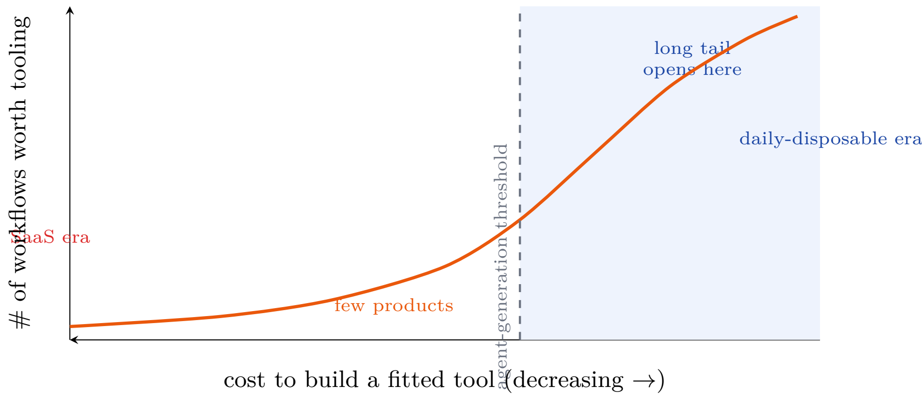


Figure 9: Why now. As the cost of building a fitted tool falls (axis reversed, so right→left is cheaper), the number of workflows worth building a dedicated tool for rises steeply. Below the agent-generation threshold (dashed), the long tail of small, idiosyncratic, fast-changing workflows becomes economically viable—the regime we call daily-disposable software.

### 8.3 From "Big and Comprehensive" to Infinite Fragmentation

This is a structural shift in what software is. For decades, software economics favored the big and comprehensive product: because building was expensive, value accrued to general-purpose platforms that amortized that cost across many customers and many use cases. Users adapted their workflows to the software, because the software could not affordably adapt to them. The incentive was always to add features, widen scope, and capture more of the workflow inside one large system.

Daily-disposable software inverts this incentive. When a tool can be generated to fit one workflow exactly, there is no reason to accept a general-purpose tool that fits approximately. Software fragments:

- Instead of one comprehensive suite, many small, sharp, single-purpose tools.
- Instead of users adapting to software, software generated to fit each user.
- Instead of a roadmap of features for an imagined average customer, a tool rebuilt on demand for the actual customer in front of you.
- Instead of software that must be maintained because it was expensive to build, software cheap enough to discard and regenerate.

The end state is infinite fragmentation: not a few large applications serving everyone approximately, but an unbounded number of small applications serving each workflow precisely. The unit of software shrinks from "product" to "workflow," and the lifespan shrinks from "years" to "as long as this workflow lasts."

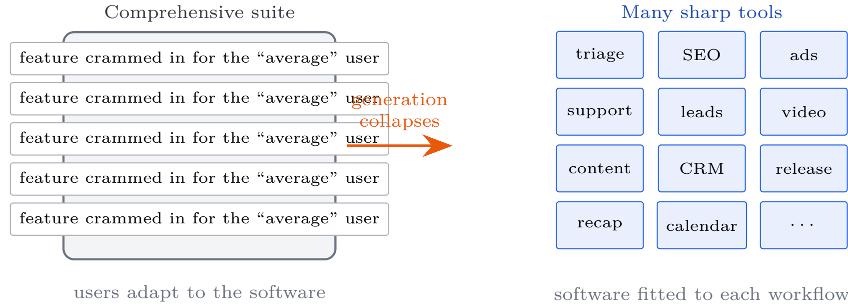


Figure 10: From “big and comprehensive” to infinite fragmentation. When building a fitted tool approaches the cost of a conversation, the incentive flips: instead of one large suite that serves everyone approximately (and that users must adapt to), an unbounded number of small, single-purpose App-in-Skills, each fitted to one workflow.

### 8.4 Implications for the Software Industry

This shift is not neutral for incumbents. The defensibility of large software companies has rested substantially on the cost of building and the lock-in of comprehensiveness: it was hard for a customer to leave a suite that did many things adequately, because rebuilding all of it elsewhere was prohibitive. When rebuilding any one piece costs a conversation, that moat drains.

We do not claim large software disappears; some workloads genuinely need scale, shared infrastructure, compliance, and reliability that a daily-disposable tool cannot provide. But the long tail—the vast number of small, specific, operational workflows that comprehensive products serve only approximately—becomes contestable by anyone who can describe the workflow to an agent. The value migrates from owning a comprehensive product to (a) the agents and skills that generate the tools, and (b) the data services, like Busabase, that turn a personal tool into a shared, trustworthy system of record.

### 8.5 The Honest Caveat

There is a tension worth naming. Infinite fragmentation risks reproducing, one App-in-Skill at a time, the very sprawl that comprehensive software was meant to tame—a household of single-purpose apps that no longer cohere. The mitigations are real but partial: a shared data provider can keep fragmented tools consistent at the data layer (this is part of Busabase's appeal), and the agent itself can orchestrate across tools, switching the operator's working view as the day's work changes. Whether fragmentation nets out as liberation or as a new kind of mess is, ultimately, an empirical question that the coming years will answer. What is no longer in doubt is that the cost structure has changed—and with it, the kind of software it makes sense to build.

## 9 How to Build One

App-in-Skill is a convention, not a framework: a builder can implement one from scratch in an afternoon. This section gives the practical path, including the scaffold we provide to make it routine.

### 9.1 The Creation Workflow

Building an App-in-Skill follows a consistent sequence:

1. Define the human story. What does the agent prepare, what does the human review or steer, and what does the skill execute? This determines the app type (Section 5).
2. Define the file handoff. Decide the state files (skill → app) and input files (app → skill) and their shapes before building any UI. The handoff is the contract.

3. Build the local app. Static UI at the app root, a minimal local server, served on `127.0.0.1` in the 3000–4000 range. Prefer zero dependencies.
4. Add the scripts. Thin generation, execution, and validation entrypoints that call shared logic in `lib/`.
5. Add locking to both the skill workflow and the app server.
6. Add the data provider. Start with the local-file provider behind the data-provider interface, so a database or cloud provider can be added later without rewrites.
7. Add private configuration with a committed template (placeholders only) and a gitignored local file for real values.
8. Add onboarding detection for missing or template-only configuration and missing secrets.
9. Expose a safe state summary at the app's state endpoint: configured accounts, sources, and provider—never secrets.
10. Add the chat-only fallback so the skill works without the UI.
11. Validate and dry-run before enabling any real external action.

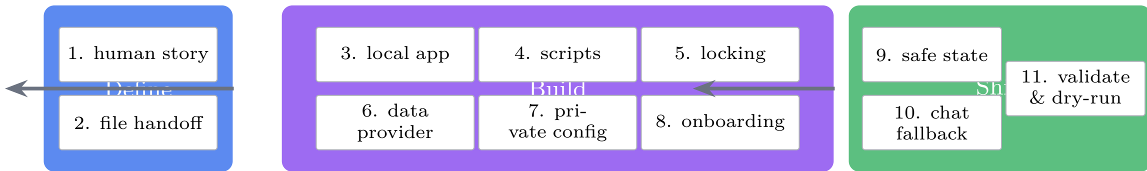


Figure 11: The creation workflow as a pipeline: Define the human story and file handoff, Build the app, scripts, locking, data provider, config, and onboarding, then Ship with a safe state summary, chat fallback, and a validated dry-run. The `app-in-skill-creator` scaffold provides the Build and Ship steps as correct-by-construction defaults, leaving the builder to specialize the human story and schema.

### 9.2 The Scaffold: `app-in-skill-creator`

To make this routine, we provide a skill whose job is to create App-in-Skills: `app-in-skill-creator`. Given a description of the workflow, it scaffolds the canonical directory layout (Appendix A), wires up the file handoff, the local server, the data-provider interface, locking, configuration, and onboarding, and leaves the builder to specialize the app type and the schemas. The scaffold encodes the specification of Section 3 as defaults, so a new App-in-Skill is correct-by-construction with respect to the boundary, locking, and safety rules.

### 9.3 Safety Defaults

Because an App-in-Skill can ultimately cause external effects (through the skill), the scaffold ships with conservative defaults that builders should keep:

- Treat money, legal, privacy, account access, destructive actions, and outbound messages as approval-required.
- Store only the minimum content needed for review; do not hoard external data locally.
- Never commit handoff data files (`app/.data/`), secrets, or personal configuration; never expose secret values through the UI, logs, reports, or screenshots—only boolean readiness.
- Make execution idempotent where possible (stable item ids, recorded results) so a re-run does not double-send.
- Keep the UI and the handoff schema in sync; if they disagree, stop and reconcile before executing.

### 9.4 An Invitation

The specification is small, the scaffold is provided, and the catalog of unmet needs is long. Most of the application space mapped in Section 7 is unbuilt, and much of it is not yet

imagined. We invite skill and agent builders to treat App-in-Skill as common ground: a shared, thin convention for giving any human-in-the-loop agent workflow a window. Build one for a workflow you know well, publish it if it generalizes, and the long tail fills in one App-in-Skill at a time.

## 10 Related Work

### 10.1 Human-in-the-Loop AI Systems

Human-in-the-loop (HITL) AI has a long history in machine learning and crowdsourcing. Settles [3] surveyed active learning, where human labelers are queried at decision boundaries. Amazon Mechanical Turk and similar platforms [4] established crowdsourced annotation as a scalable human-AI collaboration model.

These systems differ from App-in-Skill in two key ways: they are designed for large-scale data labeling (thousands of workers, millions of items), and the human role is limited to labeling or annotation. App-in-Skill targets the opposite end of the scale: a single operator or small team reviewing agent-generated content for high-stakes, context-rich workflows.

### 10.2 Agentic AI Frameworks

Recent work on AI agents has produced several frameworks for multi-step task execution. LangChain [5] and AutoGPT [6] popularized tool-use agents that can browse the web, execute code, and call APIs. ReAct [7] introduced an interleaved reasoning-action loop. LangGraph [8] adds graph-based workflow orchestration with human-in-the-loop checkpointing.

App-in-Skill is complementary to these frameworks: it defines the operator UI for human participation in agent workflows, independent of the underlying agent architecture. A LangGraph workflow can use App-in-Skill's file handoff as its checkpoint mechanism; a simple Claude Code skill can also use it without any additional framework.

### 10.3 Local-First Software

Kleppmann et al. [9] articulated the "local-first software" philosophy: applications should work offline, own the user's data, and use the local filesystem as the primary data store. App-in-Skill is aligned with local-first principles: the handoff files are local files, the app server is local, and no cloud backend is required.

The key difference is that App-in-Skill adds an agent layer between the user and their data. The agent is responsible for reading external sources; the local files are the coordination layer between agent and human.

### 10.4 Approval Workflows and Review Queues

Enterprise workflow systems (JIRA, ServiceNow, GitHub Pull Requests) implement review-and-approval patterns at scale. These systems are multi-user, persistent, and backend-heavy. They assume the workflow is stable enough to justify the engineering investment.

App-in-Skill targets "long-tail workflows": tasks that are important to one person or team, change often, and do not justify the overhead of a full enterprise system. The agent's ability to be reconfigured through chat makes App-in-Skill adaptive in ways that static workflow systems are not.

### 10.5 Intelligent User Interfaces

Horvitz [10] established principles for mixed-initiative user interfaces: AI systems that can take initiative but support human override and correction. App-in-Skill implements these principles through its app types: the agent takes initiative by preparing proposals, but the human retains override through the review and approval controls of the app.

More recent work on AI user interfaces [11] identifies challenges including calibrated uncertainty (knowing when to defer to humans) and managing context (maintaining state across sessions). App-in-Skill addresses context management through the file handoff and uncertainty through risk labels surfaced in the operator UI.

### 10.6 Prompt Engineering and Skill Systems

The AI skill ecosystem—Anthropic's Claude Code, OpenAI's Codex, GitHub Copilot Workspaces, Buda AI—has produced conventions for encoding agent instructions in files (CLAUDE.md, SKILL.md, AGENTS.md). App-in-Skill extends these conventions by defining the relationship between the instruction file and the app it launches.

The closest related concept is the Codex CLI's "operator pattern", but to our knowledge no prior work has formalized the operator UI as a thin, general specification—a skill-launched local UI over a file handoff, with a pluggable data provider and a boundary invariant—decoupled from any particular app shape.

### 10.7 No-Code and Low-Code Platforms

No-code and low-code platforms (Retool, Bubble, internal-tool builders) lowered the cost of building common application patterns. They differ from App-in-Skill in where flexibility lives: a no-code platform offers a fixed set of templated patterns that the user configures, whereas an App-in-Skill is generated by an agent to fit an arbitrary workflow, including the long tail that no template anticipates. App-in-Skill also keeps the agent in the loop at run time—reconfiguring the workflow is a conversation, not a rebuild—which no-code tools, being static once built, do not provide.

## 11 Future Work

### 11.1 Hosted, Shared App-in-Skill

The default App-in-Skill model is local-first: the app server, handoff files, and private config live on the operator's machine. The data-provider layer (Section 4) already supports cloud backends; the open work is hosting the app itself as a shared URL so that a distributed team can use one App-in-Skill without each running it locally. This would build on the cloud providers of Section 4—particularly Busabase, whose Inbox-to-canonical model already centralizes shared review state and audit trails—and define how multiple operators' decisions are attributed and merged.

### 11.2 Multi-Agent Orchestration

Current App-in-Skill designs assume a single skill writing to the batch. Future work should explore multi-agent App-in-Skill: multiple specialized agents contributing to a shared batch. For example:

- A retrieval agent that reads customer context from a CRM
- A drafting agent that generates reply text
- A risk agent that scores each item for sensitivity
- A routing agent that assigns items to the right operator

Each agent writes to a shared state file under a different namespace. The file handoff would need to define merge semantics when multiple agents contribute to the same item.

### 11.3 Streaming Batches

The current model uses a snapshot approach: the skill generates a complete batch, then the user reviews it. For high-volume or time-sensitive workflows, a streaming approach would be preferable: items appear in the app as the agent generates them, without waiting for the entire batch.

Streaming would require changes to the file handoff (append semantics rather than overwrite semantics) and the locking protocol (per-item locks rather than a single global lock).

### 11.4 Verified Execution

A limitation of the current locking protocol is that it relies on the skill to respect the lock. A compromised or buggy skill could ignore the lock and execute actions without approval.

Future work should explore verified execution: a sandboxed execution environment that enforces the boundary invariant at the OS level, ensuring that the app process cannot perform external effects regardless of the skill's behavior.

### 11.5 App-in-Skill as a Typed Magic Folder

Magic Folder Protocol (MFP) [2] defines a convention for typed, activatable directories. An App-in-Skill could be delivered as a Magic Folder with type `skill.app-in-skill`, enabling:

- Double-click to launch the local app on desktop
- Platform-appropriate icons and handlers
- Buda marketplace distribution and installation
- Automatic update and versioning via MFP

This would close the loop between the three papers: FAR makes agent inputs machine-readable; MFP makes agent outputs human-activatable; App-in-Skill makes agent workflows participatory. An App-in-Skill delivered as a Magic Folder is simultaneously all three.

### 11.6 Empirical User Studies

This paper's evaluation is based on case studies and design reasoning. Rigorous empirical evaluation is needed:

- Controlled studies comparing chat-only vs. App-in-Skill for batch review tasks
- Measurement of cognitive load (NASA-TLX), decision accuracy, and time-to-completion
- Longitudinal studies of skill adaptation over weeks and months
- Analysis of delegation patterns in team workflows

### 11.7 Formal Verification of Safety Properties

The Safety proposition in Section 6 states that no external effect is produced without an approved decision record. This should be formally verified, ideally using a model checker (TLA+, Alloy) applied to the loop. The review-queue state set is small enough to be fully enumerated; the challenge is modeling the file-system interaction correctly.

## 12 Conclusion

Talk to a capable colleague long enough and a moment always arrives when words stop being enough: someone reaches for a whiteboard, or turns over a napkin and uncaps a pen. The conversation does not stop. The sketch simply gives it a surface—somewhere to hold the list, draw the structure, mark what is decided and what is still open—so the talking can go further than speech alone could carry it.

Conversation with an agent has reached that moment. A skill can already read systems, reason, draft, and act; but when the work grows to many items, persistent state, approvals, or a second person, talk alone buckles. App-in-Skill is the surface the conversation was missing: a whiteboard the skill can pull up mid-conversation. Chat keeps carrying intent and judgment; the app holds the state, the queue, the drafts, and the decisions; and the skill, alone, reaches out to the world.

We kept the idea deliberately small. App-in-Skill is a specification, not an application: seven universal mechanisms—a skill-launched local UI, a file handoff, locking, a pluggable data

provider, private configuration, onboarding, and a chat-only fallback—and nothing about what the app must look like. Review queues, dashboards, workspaces, control panels, and collaboration spaces are things you draw on the whiteboard, not the whiteboard itself.

The consequence runs past UI convention. When a fitted tool costs about as much as a conversation, software no longer has to be big to be worth building. It fragments—from a few comprehensive products that serve everyone approximately, to an unbounded number of small, sharp tools that serve each workflow exactly, generated on demand and discarded when the work moves on. The unit of software shrinks from product to workflow, and the moats built on the cost of building and the lock-in of comprehensiveness begin to drain.

The whiteboard has been missing from the room where we work with our agents. App-in-Skill hands it over. The specification is small, the scaffold is provided, and the wall is blank—we invite builders to fill it.

# A File Handoff: A Worked Example

The specification (Section 3) fixes the shape of the file handoff but not its schema. This appendix gives a complete worked example for the review-queue app type, the most common shape. Other app types adapt the same idea with different fields.

## A.1 Canonical Directory Layout

Listing 2: Reference App-in-Skill layout

```
skill-name/
|-- SKILL.md                    # Agent instructions + safety rules
|-- agents/openai.yaml           # Agent configuration
|-- app/                         # The local UI (Part 2)
|   |-- index.html
|   |-- app.js
|   |-- styles.css
|   |-- start.sh                 # Launcher (port 3000-4000)
|   |-- i18n/messages.js         # Multilingual UI chrome
|   |-- server/                  # Local HTTP server (built-ins only)
|   `-- .data/                   # File handoff (gitignored)
|       |-- onboarding.json          # onboarding completion marker (
    latch)
|       |-- current_batch.json       # state: skill -> app
|       |-- decisions.json           # input: app -> skill
|       |-- execution_report.json    # state: skill -> app
|       `-- agent.lock               # lock
|-- scripts/                     # Deterministic entrypoints (Part 3)
|   |-- generate_batch.mjs
|   |-- execute_decisions.mjs
|   `-- validate_ui_schema.mjs
|-- lib/                         # Shared runtime (Part 3)
|   |-- paths.mjs
|   |-- common.mjs
|   `-- data-provider/           # Pluggable data provider
|-- config.example.yml           # Public template (committed)
`-- config.local.yml             # Private config (gitignored)
```

## A.2 State File (skill → app)

Listing 3: `current_batch.json` for the review-queue pattern

```
{
  "batch_id": "skill-name-YYYYMMDD-HHMMSS",
  "generated_at": "ISO 8601 timestamp",
  "source": "skill-name",
  "metrics": { "needs_review": 0, "to_approve": 0,
```

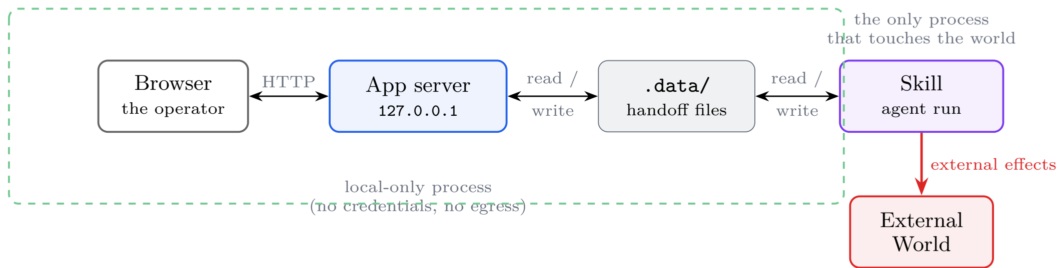


Figure 12: Runtime processes. The browser talks to a local-only app server over HTTP; the server and the intermittently-running skill process exchange data only through the `.data/` files (the green zone is local, has no credentials, and cannot reach the network). The skill process is the single point that produces external effects. The two processes never hold a live connection to each other—the files are the bridge.

```
                 "approved": 0, "done": 0, "blocked": 0, "total": 0 },
  "items": [{
    "id": "stable-local-id",
    "title": "Human-readable title",
    "summary": "One-line summary for the list view",
    "body": "Trimmed source content for the detail view",
    "category": "customer | system | finance | content | other",
    "risk": ["money", "legal", "privacy", "account"],
    "status": "needs_review | to_approve | approved | done | blocked",
    "proposed_action": "reply | archive | forward | flag | no_action",
    "reason": "Why this action is proposed",
    "suggested_reply": "Optional agent-recommended draft"
  }]
}
```

### A.3 Input File (app → skill)

Listing 4: `decisions.json`, keyed by item id

```
{
  "stable-local-id": {
    "action": "approve | revise | block | request_draft | no_action",
    "comment": "User note or instruction",
    "revised_draft": "User-edited draft, if action is revise",
    "decided_at": "ISO 8601 timestamp"
  }
}
```

### A.4 Invariants

1. Item `id` is stable across generation cycles: the same external item yields the same id (so decisions and execution results stay attached).
2. `metrics` counts sum to `total`.
3. An item is eligible for execution only if its decision has `action: "approve"` and a non-null `decided_at`.
4. The skill re-reads `decisions.json` from disk immediately before executing; it never acts on an in-memory copy.
5. All four files are written atomically (temp file then rename).

### A.5 Adapting to Other App Types

A dashboard replaces `items` with a metrics/series payload and omits `decisions.json` entirely. A workspace carries draft documents with a creative-stage field instead of an approval status. A control panel inverts the dominant direction: the input file carries parameters

and triggers, and the state file carries run status. In every case the handoff keeps the same discipline—separate state and input files, atomic writes, all state on disk—only the field schema changes.

# B Creation Workflow and Safety Checklist

This appendix collects the practical checklist for building a correct App-in-Skill, expanding Section 9.

## B.1 The Eleven Steps

1. Human story. Write down what the agent prepares, what the human reviews or steers, and what the skill executes. This fixes the app type.
2. File handoff. Define the state files and input files and their schemas before any UI work.
3. Local app. Static UI at the app root; minimal server under `app/server/`; bind to `127.0.0.1` in the 3000–4000 range; idempotent launch.
4. Scripts. Thin `generate`, `execute`, and `validate` entrypoints that import shared logic from `lib/`.
5. Locking. Acquire the lock before writes/execution; release in a `finally` step; enforce it on the server's write endpoints.
6. Data provider. Implement against the data-provider interface; ship the local-file provider as default; reserve provider names (`postgres`, `aitable`, `notion`, `busabase`) for later.
7. Private config. Commit a placeholder-only template; keep real values in a gitignored local file; secrets only in environment variables, referenced by name.
8. Onboarding detection. Detect missing, template-only, or secret-less configuration and greet the user with setup instructions instead of running.
9. Safe state summary. Expose configured accounts, sources, and active provider at the app's state endpoint—never secret values.
10. Chat fallback. Ensure the skill presents items and accepts approvals in pure text when the UI is unavailable or unwanted.
11. Validate and dry-run. Run the schema validator and a dry-run execution before enabling any real external action.

## B.2 Safety Defaults

- Treat money, legal, privacy, account access, destructive actions, and outbound messages as approval-required.
- Store only the minimum content needed for review; do not hoard external data.
- Never commit handoff data files (`app/.data/`), secrets, or personal configuration.
- Never expose secret values through UI state, logs, reports, batch files, or screenshots—expose only boolean readiness.
- Make execution idempotent (stable ids, recorded results) so a re-run does not double-act.
- If the UI and the handoff schema disagree, stop and reconcile before executing.

# C Integration with Magic Folder Protocol and FAR

## C.1 App-in-Skill as a Magic Folder Bundle

Magic Folder Protocol (MFP) [2] defines typed, activatable directories. An App-in-Skill directory is a natural candidate for the `skill.app-in-skill` MFP type.

A conformant MFP manifest for an App-in-Skill:

Listing 5: `.mf/manifest.yml` for an App-in-Skill

```
magic_folder:
  version: "1.0"
```

```yaml
  type: "skill.app-in-skill"
  name: "Email Triage"
  description: "AI-assisted inbox-zero workflow with review queue"
  author: "kelly@buda.im"

activation:
  primary: "app"            # double-click launches the app
  secondary: "chat"         # also available via agent chat

runtime:
  app:
    entrypoint: "app/start.sh"
    port_range: [3000, 4000]
    health_check: "/api/health"
  agent:
    instructions: "SKILL.md"
    config: "agents/openai.yaml"

file_contract:
  data_dir: "app/.data"
  files:
    - path: "current_batch.json"
      role: "agent_output"
      schema: "references/ui-schema.md"
    - path: "decisions.json"
      role: "human_input"
    - path: "execution_report.json"
      role: "execution_result"
    - path: "agent.lock"
      role: "lock"

privacy:
  gitignore:
    - "app/.data/"
    - "config.local.yml"
    - "*.env.local"
```

With this manifest, the host platform (Buda, macOS with an MFP handler) can:

- Display the App-in-Skill with a custom icon and typed presentation
- Launch the local app on double-click, opening the review queue in the default browser
- Open the agent skill in chat on secondary activation
- Show the batch count as a badge on the folder icon (derived from `current_batch.json`)

### C.2 FAR Integration

File-Augmented Retrieval (FAR) [1] makes files machine-readable by attaching `.meta` sidecars. The File Contract files of an App-in-Skill are natural candidates for FAR treatment:

Listing 6: `current_batch.json.meta` FAR sidecar

```json
{
  "far_version": "1.0",
  "file": "current_batch.json",
  "type": "app-in-skill/batch",
  "schema": "references/ui-schema.md",
  "description": "Latest agent-generated review batch",
  "agent_readable": true,
  "last_updated": "ISO timestamp",
  "item_count": 42,
  "items_by_state": {
    "needs_review": 12,
```

```
    "to_approve": 18,
    "approved": 5,
    "done": 7,
    "blocked": 0
  }
}
```

With a FAR sidecar on `current_batch.json`, an agent can re-enter the workflow with full context: it knows the schema, the current distribution of items by state, and where to look for the human's decisions. This enables agent re-entrancy—a key property of the Magic Folder Protocol.

### C.3 The Full Stack

The relationship between the three layers:

| Layer | Paper | Mechanism | Audience | Scope |
|---|---|---|---|---|
| File readability | FAR | `.meta` sidecars | AI agents | Static files |
| Output usability | MFP | Typed bundles + activation | Non-technical users | Static artifacts |
| Workflow participation | AiS | Skill-launched UI + file handoff | Operators and teams | Dynamic workflows |

Table 5: The three-layer human-AI interface stack. Each layer addresses a distinct scope and audience.

An App-in-Skill that is also a Magic Folder with FAR-annotated handoff files achieves all three properties simultaneously: its files are agent-readable (FAR), its directory is human-activatable (MFP), and its workflow is human-participatory (AiS).

## References


[1] Kelly Peilin Chan. File-Augmented Retrieval: Making Every File Readable to Coding Agents via Persistent .meta Sidecars. Preprint, February 2026.

[2] Kelly Peilin Chan. Magic Folder: A Filesystem-Level Abstraction for Human-Executable AI Applications. Preprint, February 2026.

[3] Burr Settles. Active Learning Literature Survey. Computer Sciences Technical Report 1648, University of Wisconsin–Madison, 2009.

[4] Panagiotis G. Ipeirotis, Foster Provost, and Jing Wang. Quality Management on Amazon Mechanical Turk. In Proceedings of the ACM SIGKDD Workshop on Human Computation (HCOMP), 2010.

[5] Harrison Chase. LangChain. https://github.com/langchain-ai/langchain, 2022.

[6] Significant Gravitas. AutoGPT: An Autonomous GPT-4 Experiment. https://github.com/Significant-Gravitas/AutoGPT, 2023.

[7] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing Reasoning and Acting in Language Models. In International Conference on Learning Representations (ICLR), 2023.

[8] LangChain Team. LangGraph: Building Stateful, Multi-Actor Applications with LLMs. https://github.com/langchain-ai/langgraph, 2024.

[9] Martin Kleppmann, Adam Wiggins, Peter van Hardenberg, and Mark McGranaghan. Local-First Software: You Own Your Data, in Spite of the Cloud. In Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!), 2019.

[10] Eric Horvitz. Principles of Mixed-Initiative User Interfaces. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI), 1999.

[11] Saleema Amershi, Dan Weld, Mihaela Vorvoreanu, Adam Fourney, Besmira Nushi, Penny Collisson, Jina Suh, Shamsi Iqbal, Paul N. Bennett, Kori Inkpen, Jaime Teevan, Ruth Kikin-Gil, and Eric Horvitz. Software Engineering for Machine Learning: A Case Study. In Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE), 2019.

[12] Anthropic. Claude: An AI Assistant by Anthropic. https://www.anthropic.com/claude, 2024.

[13] OpenAI. Codex: A Model for Code Generation. Technical Report, OpenAI, 2021.