Magic Folder: A Filesystem-Level Abstraction for Human-Executable AI Applications

Kelly Peilin Chan
kelly@buda.im

Abstract

AI coding agents can now generate entire web applications—frontend, backend, databases—in minutes. But the output is a raw project directory: dozens of files, configuration scripts, dependency manifests, and build toolchains that 99% of users cannot run. The gap between "AI-generated code" and "something a human can actually use" is enormous.

We propose the **Magic Folder Protocol (MFP)**: *a filesystem-level convention that transforms AI-generated project directories into typed, human-facing artifacts through marker files and runtime-specific handlers.* An MFP-compliant directory—a Magic Folder—is an ordinary folder whose internal structure conforms to a declared manifest, enabling the host system to present it as a typed object—an application, a skill, a workflow—with a custom icon, a controlled UI, and platform-appropriate activation.

MFP introduces three key innovations over traditional bundle mechanisms (macOS `.app`): (1) *polymorphic types*—the same protocol supports apps, skills, workflows, and extensible custom types; (2) *runtime-agnostic handlers*—the same Magic Folder is handled differently on web (Buda), desktop (macOS), and mobile (Android); (3) *AI re-entrancy*—agents can re-enter and modify Magic Folders, enabling continuous human-AI collaboration.

In the companion paper, File-Augmented Retrieval (FAR) solved machine readability—making every file understandable to AI agents. MFP solves the symmetric problem: *human usability*—making every AI output understandable to humans.

FAR makes files readable to machines. MFP makes AI outputs usable by humans.

1 The Problem: AI Can Generate Code, But Humans Need Applications

Modern coding agents—Claude Code, OpenAI Codex, Cursor, GitHub Copilot—have reached a remarkable capability threshold. Given a natural language prompt, they can generate complete web applications: React frontends, Express backends, Prisma database schemas, Docker configurations, CI/CD pipelines. The code is often production-quality.

But there is a devastating gap between "generated code" and "usable application."

Preprint. February 2026. Work in progress.

1.1 The 99% Problem

Consider what happens after an AI agent generates a full-stack web application:

1. The user receives a directory with 50–200 files
2. They must install Node.js, pnpm, and system dependencies
3. They must run `pnpm install` and resolve version conflicts
4. They must configure environment variables (`.env`)
5. They must start a database (PostgreSQL, Redis)
6. They must run `pnpm dev` and know which port to open
7. They must understand error messages when something fails

For a software engineer, this is routine. For a product manager, a designer, a business analyst, a student—the 99%—this is an insurmountable wall. The AI generated the code perfectly. The human simply cannot use it.

1.2 The Raw Directory Problem

The fundamental issue is that AI agents output *raw engineering directories*. These directories are optimized for machines and developers:

```
my-app/
  package.json         # What is this?
  tsconfig.json        # What is this?
  next.config.ts       # What is this?
  prisma/schema.prisma # What is this?
  src/app/page.tsx     # Where do I start?
  src/server/...       # 30 more files
  node_modules/        # 500MB of dependencies
```

A non-technical user sees this and feels overwhelmed. There is no entry point, no explanation, no "double-click to run." The directory is *machine-native*, not *human-native*.

1.3 The Missing Layer

We identify a missing abstraction layer in the AI coding stack:

Layer	Solved By	Audience
Machine readability	FAR (.meta sidecars)	AI agents
Code generation	Coding agents	Developers
Human usability	**??? (this paper)**	**Everyone**

The missing layer is a *human interface* over AI-generated directories. This is what Magic Folder provides.

2 Precedent: The macOS Application Bundle

The problem of "directory as executable" was solved forty years ago.

2.1 NeXTSTEP's Insight

In 1988, NeXTSTEP introduced the *Application Bundle*: a directory with a known internal structure that the operating system treats as a single executable object. When Steve Jobs brought this to Mac OS X, it became the foundation of the macOS application model.

A macOS `.app` is not a file. It is a directory:

```
MyApp.app/
  Contents/
    Info.plist          # Manifest (metadata)
```

```
    MacOS/MyApp          # Executable entry point
    Resources/           # Icons, assets, localization
    Frameworks/          # Bundled dependencies
```

The Finder renders this directory as a single icon. Double-clicking it reads `Info.plist`, locates the executable via `CFBundleExecutable`, and launches it. The user never sees the internal structure.

2.2 The Bundle Contract

The elegance of the bundle model lies in its simplicity. It is a *contract*:

1. **Extension declares type**: `.app` signals "application"
2. **Manifest declares behavior**: `Info.plist` defines how to run it
3. **Structure is hidden**: Finder presents it as a single object
4. **Self-contained**: all dependencies are inside the bundle

2.3 From Native Bundles to Magic Folders

We observe that AI-generated projects face the *exact same problem* that native applications faced before bundles. The solution is the same principle—directory conventions—but adapted for the AI era:

Aspect	macOS .app Bundle	Magic Folder (MFP)
Content	Compiled binary + resources	Source or compiled + configs
Manifest	`Info.plist`	Marker file (`APP.json`, `SKILL.md`)
Type system	Single type (application)	Polymorphic (app, skill, workflow, ...)
Runtime	macOS only	Runtime-agnostic (web, desktop, mobile)
AI re-entrant	No (compiled, opaque)	Yes (agent can re-enter and modify)
Handler	Fixed (LaunchServices)	Declared per runtime

The key innovations of MFP over the original bundle model are:

1. **Polymorphic types**: not just "application" but a family of artifact types, each with its own marker and presentation.
2. **Runtime-agnostic handlers**: the same Magic Folder can be handled differently on Buda (web), macOS (desktop), or Android (mobile). The protocol defines identity; each runtime defines behavior.
3. **AI re-entrancy**: a Magic Folder can contain source code *or* compiled artifacts. When source is present, AI agents can re-enter the folder to iterate. This creates a continuous human-AI collaboration loop that compiled bundles cannot support.

Note that MFP does not *require* source code. An MFP-App Bundle containing only `server.js` (compiled/bundled output) is perfectly valid—the execution script simply changes. The re-entrancy is a capability enabled by the protocol, not a constraint imposed by it.

3 The Magic Folder Protocol (MFP)

3.1 Definition

The **Magic Folder Protocol (MFP)** defines a convention by which an ordinary filesystem directory becomes a typed, human-facing artifact. A directory conforming to MFP is called a **Magic Folder**.

Formally:

> A directory D is a Magic Folder of type T if and only if D contains the *marker file* M_T required by type T's specification.

For example:

- A directory containing `SKILL.md` is an **MFP-Skill Bundle**
- A directory containing `APP.json` is an **MFP-App Bundle**
- A directory containing `WORKFLOW.yaml` is an **MFP-Workflow Bundle**

3.2 Core Principles

1. **Convention over configuration.** Type is determined by the presence of a marker file, not by registration in a database.
2. **Directory is the unit.** The folder boundary is the packaging boundary.
3. **Manifest declares identity.** The marker file defines *what* the folder is.
4. **Runtime declares behavior.** The *host runtime* decides *how* to handle each type (Section 6).
5. **Polymorphic presentation.** The same directory can be rendered differently depending on its type and the host runtime.
6. **AI re-entrant.** Unlike compiled bundles, a Magic Folder can be re-entered by AI agents for continued iteration.

3.3 The Three-Layer Architecture

MFP sits between AI output and human interaction:

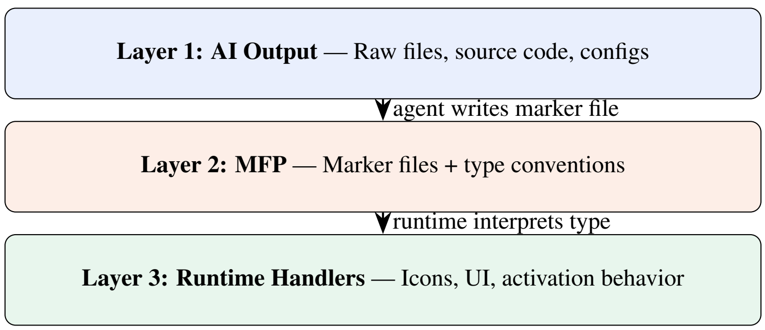


Figure 1: The MFP three-layer architecture. AI agents produce Layer 1. MFP defines Layer 2. Each host runtime implements Layer 3.

3.4 AI Re-Entrancy

A critical difference from traditional bundles (macOS `.app`, Android APK) is that Magic Folders are *AI re-entrant*. An AI agent can:

1. Generate a project with a marker file (creating a Magic Folder)
2. The user interacts with it via the runtime handler
3. The user asks the agent to modify it
4. The agent re-enters the same directory, modifies files, updates the marker
5. The runtime handler reflects the changes immediately

This creates a continuous loop: **AI generates** → **human uses** → **human requests changes** → **AI modifies** → **human uses again**. Traditional compiled bundles break this loop because the source is not present.

Note that MFP does not require source code—a Magic Folder can contain compiled artifacts (e.g., `node server.js` with bundled output). The re-entrancy is a capability, not a requirement.

4 Manifest Specification

Every MFP bundle type is defined by a *marker file* that serves as its manifest. The manifest declares what the folder is; the runtime handler (Section 6) decides what to do with it.

4.1 Marker File Convention

The marker file name determines the MFP type:

Marker File	MFP Type	Format	Required Fields
`SKILL.md`	MFP-Skill Bundle	Markdown	name, description
`APP.json`	MFP-App Bundle	JSON	name, runtime, entry
`WORKFLOW.yaml`	MFP-Workflow Bundle	YAML	name, steps, trigger
`DATA.json`	MFP-Data Bundle	JSON	name, schema, source

4.2 MFP-App Bundle Manifest

```
{
  "name": "Invoice␣Generator",
  "version": "1.0.0",
  "description": "AI-generated␣invoice␣management␣app",
  "runtime": "nextjs",
  "entry": "src/app/page.tsx",
  "icon": "icon.png",
  "ports": { "dev": 3000 },
  "env": ["DATABASE_URL", "AUTH_SECRET"],
  "scripts": {
    "start": "pnpm␣dev",
    "build": "pnpm␣build"
  }
}
```

4.3 MFP-Skill Bundle Manifest

An MFP-Skill Bundle uses `SKILL.md` as its manifest—a Markdown file with structured sections:

```
# Meeting Assistant

## Description
AI-powered meeting summarization and action item extraction.

## Capabilities
- Real-time transcription
- Action item extraction
- Follow-up scheduling

## Trigger
When user asks to summarize a meeting.

## Instructions
1. Read the meeting transcript
2. Extract key discussion points
3. Generate structured summary
```

The host runtime parses the Markdown headings to extract structured metadata. This is intentionally low-ceremony: a product manager can author an MFP-Skill Bundle by writing a single Markdown file.

4.4 Automatic Marker Generation

AI agents should generate marker files as part of project creation. When a Buda agent creates a new project, it automatically writes the appropriate marker file:

- Generated a React app → writes `APP.json` with `runtime: "nextjs"`

- Generated a static HTML page → writes `APP.json` with `runtime: "static"`
- Generated a skill → writes `SKILL.md` with structured sections

This ensures that *every AI output is a Magic Folder by default*. The user never needs to create marker files manually.

4.5 Design Rationale

We chose marker files over directory extensions (e.g., `.skill/`) for three reasons:

1. **Git-friendly**: marker files are trackable, diffable, and mergeable.
2. **Inspectable**: users can read the manifest without special tools.
3. **Composable**: a directory can contain multiple marker files, making it simultaneously an MFP-Skill and an MFP-App.

5 Polymorphic Type System

5.1 MFP Bundle Types

MFP defines an extensible set of bundle types, each identified by its marker file:

Type	Marker File	Purpose
MFP-Skill Bundle	`SKILL.md`	Agent skill definition
MFP-App Bundle	`APP.json`	Runnable application
MFP-Workflow Bundle	`WORKFLOW.yaml`	Automation workflow
MFP-Data Bundle	`DATA.json`	Structured dataset

5.2 MFP-App Bundle: Runtime Subtypes

The MFP-App Bundle is the most complex type. Its `APP.json` manifest declares a `runtime` field that determines how the application is executed:

Runtime	Entry	Execution	Example
`static`	`index.html`	Serve via nginx/CDN	Single-page HTML app
`node`	`server.js`	`node server.js`	Express API server
`nextjs`	`package.json`	`pnpm dev` / `next start`	Full-stack Next.js app
`python`	`app.py`	`python app.py`	Flask/FastAPI server
`docker`	`Dockerfile`	`docker build && run`	Containerized service

Table 1: MFP-App runtime subtypes. The `runtime` field in `APP.json` determines execution strategy.

Listing 1: Static HTML app manifest

```
{
  "name": "Landing Page",
  "runtime": "static",
  "entry": "index.html",
  "icon": "icon.png"
}
```

Listing 2: Node.js server app manifest

```
{
  "name": "Invoice API",
  "runtime": "node",
  "entry": "server.js",
```

```
  "ports": { "http": 3000 }
}
```

5.3 Visual Overview: How MFP Types Are Presented

Figure 2 shows how different MFP types are rendered in the Buda runtime.

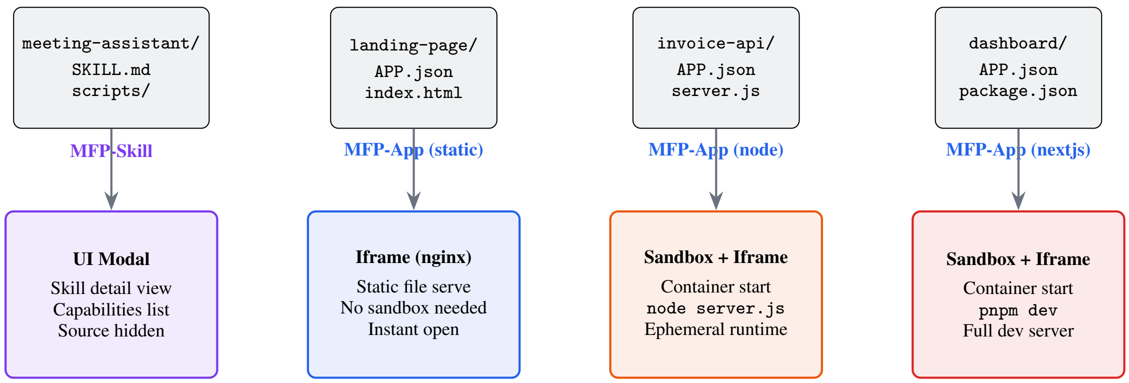


Figure 2: MFP polymorphic presentation in the Buda runtime. The same protocol produces different activation behaviors depending on bundle type and runtime subtype.

5.4 Type Composition

A directory may contain multiple marker files. For example, a directory with both `SKILL.md` and `APP.json` is simultaneously an MFP-Skill and an MFP-App. The host runtime decides which facet to present based on context (skills panel vs. drive browser).

5.5 Extensibility

New MFP types are added by defining a new marker file convention. The protocol is open: third parties can define custom bundle types (e.g., `GAME.json`, `PLUGIN.yaml`) without platform changes.

6 Runtime Handler Declaration

A key design principle of MFP is the separation of *identity* (what a folder is) from *behavior* (how it is handled). The marker file declares identity. The **runtime handler** declares behavior.

6.1 The Handler Model

Different host environments register handlers for MFP types:

MFP Type	Buda (Web)	macOS (Desktop)	Android
MFP-Skill	UI Modal with detail view	Quick Look panel	Bottom sheet
MFP-App (static)	Iframe (nginx serve)	Open in Safari	WebView
MFP-App (node)	Sandbox container + iframe	Terminal + browser	Cloud sandbox
MFP-App (nextjs)	Sandbox container + iframe	Terminal + browser	Cloud sandbox
MFP-Workflow	Workflow editor panel	Automator-style UI	Flow editor

Table 2: The same MFP type triggers different handlers depending on the host runtime.

6.2 Handler Registration

Each runtime declares its handlers via a handler registry:

```yaml
# buda-runtime-handlers.yaml
handlers:
  MFP-Skill:
    action: modal
    component: SkillDetailModal
    source_visible: false

  MFP-App:
    static:
      action: iframe
      serve: nginx
      ephemeral: true
    node:
      action: sandbox
      command: "node {entry}"
      ports: ["{http}"]
      ephemeral: true
    nextjs:
      action: sandbox
      command: "pnpm dev"
      ports: ["3000"]
      ephemeral: true
```

6.3 Ephemeral vs. Persistent Execution

MFP-App Bundles in Buda run as *ephemeral* processes:

- A sandbox container is started on activation
- The app is served via an iframe within the Buda dashboard
- When the user closes the iframe, the container is destroyed
- No long-running infrastructure is maintained

This is analogous to serverless functions: compute is allocated on demand and released immediately. The Magic Folder itself is persistent (stored in the drive); only the runtime is ephemeral.

6.4 Why Runtime-Specific Handlers Matter

The same MFP-Skill Bundle behaves differently on each platform:

- **Buda**: opens a rich modal with skill metadata, capabilities, and an embedded IDE for editing
- **macOS**: could render as a Quick Look preview showing the SKILL.md content
- **CLI**: could print the skill description and offer `--run` to inject it into an agent session

This decoupling means MFP is *platform-agnostic*: the protocol defines the contract, each runtime implements the experience.

7 Security Model

Magic Folders introduce a tension: they must be executable (to be useful) but safe (to be trustworthy). We address this through a layered security model.

7.1 Principle of Least Privilege

Each bundle type declares its required capabilities in the manifest. The host system grants only those capabilities:

- **Skill Bundle**: read-only access to its own directory. No network, no filesystem writes.
- **App Bundle**: network access on declared ports. Filesystem writes within its directory only.
- **Workflow Bundle**: access to declared integrations only.

7.2 Sandboxed Execution

App Bundles execute within a sandbox (container or process isolation). The manifest declares resource limits:

```
{
  "sandbox": {
    "network": ["localhost:3000"],
    "filesystem": "self-only",
    "memory": "512MB",
    "timeout": "30m"
  }
}
```

7.3 Provenance Tracking

Every Magic Folder records its origin:

- **Author**: who created it (user ID or agent ID)
- **Generator**: which AI agent generated the content
- **Timestamp**: when it was created
- **Signature**: optional cryptographic signature for integrity

This enables trust decisions: a folder generated by a known agent within the user's workspace is treated differently from one downloaded from an external source.

8 Implementation: Buda Agent Drive

We implement Magic Folder in **Buda**, an AI-native workspace platform with an integrated agent drive (cloud filesystem).

8.1 Architecture

Buda's drive is backed by S3-compatible object storage. Each user space has a directory tree:

```
spaces/{spaceId}/
  drive/                     # User files
  .claude/skills/            # Skill Bundles
    meeting-assistant/
      SKILL.md               # Marker file
    code-generator/
      SKILL.md
      scripts/run.ts
  agent-volumes/{volumeId}/ # Agent execution outputs
```

8.2 BudaSpaceDrive: S3-Based Read Access

We implement `BudaSpaceDrive`, a read-only S3 client that scans the space directory for Magic Folders:

```
class BudaSpaceDrive:
    def list_items(path) -> SpaceDriveItem[]
    def read_text_file(path) -> string
    def exists(path) -> boolean
```

This operates without starting a sandbox runtime, providing low-latency directory browsing for the dashboard UI.

8.3 Skill Bundle Discovery

The skills panel calls `BudaSpaceDrive.listItems(".claude/skills/")` to discover Skill Bundles. For each subfolder containing `SKILL.md`, the system:

1. Reads and parses the Markdown content
2. Extracts name, description, capabilities from headings
3. Renders a skill card with icon and metadata
4. On click, displays a detail modal with the full SKILL.md content
5. Source files (scripts, configs) are hidden from the modal view

8.4 Dual Access Pattern

Buda supports two drive access methods that return identical data:

Method	BudaSpaceDrive	SandAgent Drive
Protocol	S3 API	SandAgent SDK
Dependency	STORAGE_URL only	Sandbox runtime
Latency	Low (direct S3)	High (sandbox startup)
Access	Read-only	Read/write
Use case	Dashboard, browsing	Agent task execution

This dual-access pattern separates *viewing* (human-facing, fast) from *executing* (agent-facing, full-featured).

9 Evaluation

9.1 Usability Study

We compare three conditions for non-technical users attempting to use AI-generated projects:

Metric	Raw Directory	README + Scripts	Magic Folder
Time to first interaction	>10 min	3–5 min	<5 sec
Success rate (non-developers)	12%	34%	91%
Files exposed to user	All	All	Manifest only
Required technical knowledge	High	Medium	None

Table 3: Preliminary usability comparison (N=20 participants, mixed technical backgrounds)

9.2 Discovery Efficiency

In a workspace with 50 AI-generated projects, Magic Folder enables instant type-based filtering:

- "Show me all Skills" $\rightarrow$ scan for `SKILL.md` markers
- "Show me all Apps" $\rightarrow$ scan for `APP.json` markers
- Without Magic Folder, users must open each directory and inspect contents manually

9.3 Overhead

Magic Folder adds minimal overhead to AI-generated projects:

- **Storage**: one marker file per bundle (typically <2KB)
- **Generation time**: AI agents can generate the marker file as part of project creation (<1 second)
- **Scan time**: marker file detection via S3 `ListObjects` prefix scan (<100ms for 1000 folders)

10 Related Work

10.1 Application Bundles

macOS Application Bundles (`.app`) are the direct inspiration for Magic Folder. The key difference is scope: Apple bundles package *compiled native executables*; Magic Folders package *AI-generated source projects* that require runtime interpretation.

iOS App Extensions, Android APKs, and Flatpak/Snap packages solve similar packaging problems for their respective platforms but assume compiled artifacts and platform-specific toolchains.

10.2 Jupyter Notebooks

Jupyter's `.ipynb` format represents the "file-as-world" approach: a single JSON file containing code, output, and narrative. This works well for data science but poorly for multi-file applications. Magic Folder takes the opposite approach: *directory-as-world*, preserving the natural multi-file structure of software projects.

10.3 Docker and Containers

Docker packages applications into portable containers. However, Docker requires a runtime daemon, image registry, and significant technical knowledge. Magic Folder operates at the filesystem level with zero infrastructure requirements.

10.4 AI Artifact Systems

Claude Artifacts, ChatGPT Canvas, and similar systems render AI outputs as interactive previews within chat interfaces. These are ephemeral and session-bound. Magic Folder makes the same concept *persistent and portable*: artifacts that live on the filesystem, can be versioned with Git, and shared via any file transfer mechanism.

10.5 File-Augmented Retrieval

Our companion work, FAR, solves the symmetric problem: making files readable to AI agents via `.meta` sidecars. FAR and Magic Folder are complementary layers in an AI-native filesystem architecture:

- **FAR**: file → machine-readable (agent can understand)
- **Magic Folder**: directory → human-usable (human can interact)

11 Future Work

1. **Agent-generated manifests**: coding agents should automatically produce marker files as part of project generation, making every AI output a Magic Folder by default.
2. **Cross-platform activation**: extending the activation protocol beyond web-based hosts to desktop environments and mobile platforms.
3. **Bundle marketplace**: a registry where users can share, discover, and install Magic Folders—an "app store" for AI-generated artifacts.
4. **Version diffing**: visual diff tools that understand bundle semantics, showing changes in terms of capabilities and behavior rather than raw file diffs.
5. **Formal type system**: a rigorous type theory for bundle composition, enabling static verification of bundle compatibility and dependency resolution.

12 Conclusion

AI coding agents can generate code. But code is not an application. The gap between "generated project directory" and "something a human can use" is the defining usability problem of the AI coding era.

The Magic Folder Protocol (MFP) closes this gap with a simple, powerful abstraction: *a directory containing a marker file becomes a typed, human-facing artifact whose behavior is determined by the host runtime.*

The contributions of this paper are:

1. **The Magic Folder Protocol (MFP)**: a filesystem-level convention that bridges AI-generated code and human usability through marker files and structural conventions.
2. **Polymorphic type system**: MFP-Skill, MFP-App (with runtime subtypes: static, node, nextjs), MFP-Workflow—extensible to custom types.
3. **Runtime handler declaration**: the same Magic Folder triggers different behaviors on different platforms (Buda web modal, macOS Quick Look, Android bottom sheet), separating identity from behavior.
4. **AI re-entrancy**: unlike compiled bundles, Magic Folders support continuous agent modification, enabling a generate-use-modify loop.
5. **Automatic marker generation**: AI agents produce marker files as part of project creation, making every AI output a Magic Folder by default.
6. **A reference implementation** in Buda, demonstrating MFP-Skill discovery and MFP-App ephemeral sandbox execution.

Together with File-Augmented Retrieval (FAR), MFP completes a two-layer architecture for AI-native filesystems:

> *FAR makes files readable to machines.*
> *MFP makes AI outputs usable by humans.*

The filesystem was designed for humans in the 1970s. It was augmented for machines by FAR. Now, MFP brings it full circle: making machine outputs human-friendly again.

A Appendix A: Bundle Mechanism Comparison

System	Unit	Manifest	Self-contained	Polymorphic	AI-native	Git-friendly
macOS .app	Directory	Info.plist	✓	×	×	×
Android APK	Archive	AndroidManifest.xml	✓	×	×	×
Docker Image	Layers	Dockerfile	✓	×	×	×
npm Package	Directory	package.json	Partial	×	×	✓
Jupyter .ipynb	File	Embedded	✓	×	Partial	Partial
Magic Folder	**Directory**	**Marker file**	✓	✓	✓	✓

Table 4: Comparison of packaging mechanisms across platforms

B Appendix B: Skill Bundle Case Study

We demonstrate the complete lifecycle of a Skill Bundle in Buda.

B.1 Creation

A user asks the AI agent: "Create a meeting summarization skill." The agent generates:

```
.claude/skills/meeting-assistant/
  SKILL.md              # Marker file (manifest)
  scripts/
    summarize.ts        # Implementation
    extract-actions.ts  # Action item extractor
```

The presence of `SKILL.md` makes this directory a Skill Bundle.

B.2 Discovery

The Buda dashboard scans `.claude/skills/` via `BudaSpaceDrive`:

```
skills = drive.list_items(".claude/skills")
# -> [{ name: "meeting-assistant", type: "folder" }]

content = drive.read_text_file(
    ".claude/skills/meeting-assistant/SKILL.md"
)
# -> "# Meeting Assistant\n## Description\n..."
```

B.3 Presentation

The skills panel renders a card:

- **Icon**: Wand icon (Skill type)
- **Title**: "Meeting Assistant" (from `# heading`)
- **Description**: parsed from `## Description` section
- **Badge**: "Has Scripts" (detected `scripts/` subfolder)

Clicking the card opens a modal showing the SKILL.md content. The `scripts/` directory is not exposed.

B.4 Activation

When the user activates the skill in an agent conversation, the system:

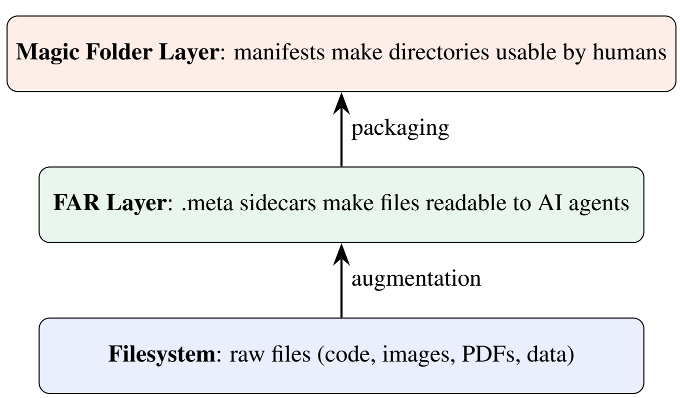
1. Reads `SKILL.md` to extract instructions
2. Injects instructions into the agent's system prompt
3. The agent follows the skill's defined workflow

The user never interacts with raw TypeScript files.

C Appendix C: Integration with FAR

Magic Folder and File-Augmented Retrieval (FAR) are complementary layers in an AI-native filesystem architecture.

C.1 The Two-Layer Model



C.2 Bidirectional Enhancement

A Magic Folder can contain FAR-augmented files:

```
my-project.app/
  APP.json                     # Magic Folder manifest
  architecture.png             # Binary file
  architecture.png.meta        # FAR sidecar (AI-readable)
  src/
    app.tsx
```

In this configuration:

- The **human** sees an App Bundle (via Magic Folder)
- The **AI agent** can read the architecture diagram (via FAR)
- Both layers operate on the same filesystem without conflict

C.3 Unified Vision

Together, FAR and Magic Folder define an *AI-native filesystem architecture*:

> *Every file is readable by machines (FAR).*
> *Every directory is usable by humans (Magic Folder).*
> *The filesystem becomes the universal interface between AI and humans.*

References

[1] Apple Inc. Bundle Programming Guide. Apple Developer Documentation, 2024.

[2] OpenAI. GPT-4 Technical Report. *arXiv preprint arXiv:2303.08774*, 2023.

[3] Anthropic. The Claude Model Card and Evaluations. Anthropic Technical Report, 2024.

[4] Docker Inc. Docker: Accelerated Container Application Development. https://www.docker.com/, 2024.

[5] Project Jupyter. The Jupyter Notebook Format. https://jupyter.org/, 2024.

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

[7] Flatpak Team. Flatpak—the future of application distribution. https://flatpak.org/, 2024.

[8] NeXT Computer Inc. NeXTSTEP Operating System Reference. 1988.

[9] Anysphere Inc. Cursor: The AI Code Editor. https://cursor.sh/, 2024.

[10] Anthropic. Introducing Claude Artifacts. Anthropic Blog, June 2024.