maw-js Local Plugins

> ***The Complete Technical Guide*** *— How* `.maw/plugins/` *auto-loading really works, proven from the source up.*

Author: Lord Knight Oracle `[ai-core:lord-knight]` 🤖 **Model**: Claude Opus (1M context) **Date**: 2026-06-15 **Subject repo**: `Soul-Brews-Studio/maw-js` @ `maw v26.6.13-alpha.142` (build `bcf6f8af`)

Foreword — A Book for the Skeptic

There is a claim that sounds too convenient to be true:

> *"Drop a folder called* `.maw/plugins/` *into any project, and the* `maw` *CLI will auto-load your custom command — no global install, no registry, no build step."*

A reasonable engineer hears that and says: **"Prove it."**

That is exactly what this book does. Every chapter is backed by one of two kinds of evidence:

1. **Source proof** — the actual TypeScript from `maw-js`, with file paths and line numbers you can open yourself.
2. **Runtime proof** — real terminal transcripts captured by running `maw` on a machine, not mock-ups.

By the end you will be able to: explain the loading algorithm precisely, read the manifest schema fluently, write a working local plugin in under five minutes, and recognise the exact security gates the engine enforces at load time.

No hand-waving. No "trust me." Just the code and the receipts.

Chapter 1 — The 30-Second Proof

Before any theory, here is the whole claim demonstrated end-to-end. This is a verbatim transcript from a real machine.

Step 1 — Make a plugin in a throwaway directory

```
DEMO=/tmp/maw-lpdemo
mkdir -p $DEMO/.maw/plugins/hello-oracle
cd $DEMO
```

`.maw/plugins/hello-oracle/plugin.json`:

```
{
  "name": "hello-oracle",
  "version": "1.0.0",
  "entry": "./index.ts",
  "sdk": "^1.0.0",
  "cli": {
    "command": "hello-oracle",
    "help": "maw hello-oracle [name] — proof that local plugins auto-load"
  },
  "weight": 50
}
```

`.maw/plugins/hello-oracle/index.ts`:

```
import type { InvokeContext, InvokeResult } from "maw-js/plugin/types";

export default async function handler(ctx: InvokeContext): Promise<InvokeResult> {
  const args = (ctx.source === "cli" ? (ctx.args as string[]) : []) ?? [];
  const name = args[0] ?? "skeptic";
  return {
    ok: true,
    output: `Hello, ${name}! This plugin lives in ./.maw/plugins/ and maw auto-loaded it (no global install).`,
  };
}
```

Step 2 — Run it. No install. No build.

```
$ maw hello-oracle Nat
loaded config: 0 triggers, 0 declared plugins, 3 peers
loaded 129 plugins (126 symlink, 3 legacy)
Hello, Nat! This plugin lives in ./.maw/plugins/ and maw auto-loaded it (no global install).
```

It worked on the first run. The custom command `hello-oracle` did not exist in `maw` a minute ago — now it does, because the file is on disk in the right place.

Step 3 — Prove it's *local*, not global

Two more runs nail down that this is genuinely directory-scoped, not a one-time global registration.

From a deep subdirectory — the engine walks *up* the tree to find the plugin root:

```
$ cd /tmp/maw-lpdemo/src/deep/nested
$ maw hello-oracle WalkUp
loaded 129 plugins (126 symlink, 3 legacy)
Hello, WalkUp! This plugin lives in ./.maw/plugins/ and maw auto-loaded it (no global install).
```

From outside the directory — the command vanishes:

```
$ cd /tmp
$ maw hello-oracle Outsider
loaded 128 plugins (126 symlink, 2 legacy)
✗ unknown command: hello-oracle
  run 'maw --help' to see available commands
```

The receipt

Look at the plugin counts in those three transcripts:

- Inside the project (or any subdir of it): **129 plugins loaded**
- Outside the project: **128 plugins loaded**

The difference is exactly **one** — our `hello-oracle`. The engine is not caching a global registration; it recomputes the plugin set from the current working directory on *every invocation*. Move out of the tree and the plugin is gone. Move back in and it returns.

True or not true? — *True.* The rest of this book explains precisely *why*, line by line.

Chapter 2 — The Loading Algorithm

The behaviour in Chapter 1 comes from two small, readable functions. There is no magic and no hidden daemon.

2.1 — Where the engine looks

File: `src/plugin/registry-helpers.ts`, lines 19–31:

```
// Scan the global plugin dir plus any repo-local `.maw/plugins` dirs found by
// walking up from cwd. Resolved at call time so tests and per-command cwd
// routing can override roots.
export function discoverLocalPluginDirs(cwd = process.cwd()): string[] {
  const dirs: string[] = [];
  let dir = resolve(cwd);
  for (let i = 0; i < 32; i += 1) {
    const pluginsDir = join(dir, ".maw", "plugins");
    if (existsSync(pluginsDir)) dirs.push(pluginsDir);
    if (existsSync(join(dir, ".maw-root"))) break;
    const parent = dirname(dir);
    if (parent === dir) break;
    dir = parent;
  }
  return dirs;
}
```

Read it slowly — every line matters:

- **`let dir = resolve(cwd)`** — start at the directory where you ran `maw`.
- **The `for` loop (max 32 hops)** — climb toward the filesystem root, one parent at a time. The cap of 32 is a sane upper bound on directory depth and guarantees the loop always terminates.
- **`if (existsSync(pluginsDir)) dirs.push(...)`** — every `.maw/plugins/` found *along the way* is collected. Nested projects can each contribute plugins.
- **`if (existsSync(join(dir, ".maw-root"))) break`** — an optional `.maw-root` marker file lets you declare a hard boundary, so the walk doesn't leak into parent projects.
- **`if (parent === dir) break`** — stop at the filesystem root (`dirname("/") === "/"`).

This single function explains **Proof 2** from Chapter 1: from `src/deep/nested`, the loop climbs to the project root, finds `.maw/plugins/`, and loads `hello-oracle`.

2.2 — Local + global, merged

File: `src/plugin/registry-helpers.ts`, lines 33–36:

```
export function scanDirs(cwd = process.cwd()): string[] {
  const pluginDirs = [...discoverLocalPluginDirs(cwd), process.env.MAW_PLUGINS_DIR || mawDataPath("plugins")];
  return [...new Set(pluginDirs)];
}
```

`scanDirs()` returns the **local dirs first**, then the global directory (`~/.maw/plugins`, overridable with the `MAW_PLUGINS_DIR` environment variable). The `new Set(...)` dedupes, so the same path is never scanned twice.

This is the full answer to the headline question:

> ***Local `.maw/plugins` and global `~/.maw/plugins` are loaded together, every time, with the local ones discovered by walking up from your current directory.***

2.3 — Wiring it into discovery

`scanDirs()` is not a side path — it feeds the main discovery routine that the whole CLI depends on.

File: `src/plugin/registry.ts`, lines 148–151:

```
export function discoverPackages(deps: DiscoverPackagesDeps = {}): LoadedPlugin[] {
  const useCache = deps.useCache ?? (!deps.loadConfig && !deps.resolveActiveProfileFilter && !deps.scanDirs);
  ...
  const pluginDirs = (deps.scanDirs ?? scanDirs)();
```

`discoverPackages()` is what the command dispatcher calls to resolve `maw <something>`. Because it sources its directories from `scanDirs()`, **every** plugin-aware code path — CLI dispatch, the HTTP API, lifecycle hooks — automatically respects local plugins. There is no separate "local mode" to enable.

2.4 — Why the count changes

Recall the receipt: 129 inside, 128 outside. That is `discoverPackages()` running fresh per invocation. The result is cached *within a single process* (one CLI call) but never persisted across calls — so each `maw` you type re-walks the tree from wherever you are. Directory in, plugin in; directory out, plugin out.

Chapter 3 — The Plugin Manifest

A plugin announces itself with a manifest. The minimal form is a `plugin.json`; the modern form is a typed `plugin.ts` that re-exports the same shape via `definePlugin()`.

3.1 — The full schema

File: `src/plugin/types.ts` (the `PluginManifest` interface). The fields you will actually use:

Field	Type	Purpose
`name`	`string` (`/^[a-z0-9-]+$/`)	Slug-safe identity; must match the folder name.
`version`	`string` (semver)	Plugin version.
`sdk`	`string` (semver range)	Which maw-js SDK the plugin targets, e.g. `^1.0.0`.
`entry`	`string`	Relative path to the TS/JS entry (`./index.ts`).
`cli`	`object`	Command name, `aliases`, `help`, and a typed `flags` map.
`api`	`object`	HTTP surface: `path` + allowed `methods`.
`hooks`	`object`	Event-pipeline + lifecycle hooks (Chapter 5).
`weight`	`number` (0–99)	Load/execution order — lower runs first. Default 50.
`tier`	`"core" | "standard" | "extra"`	Membership tier.
`capabilities`	`string[]`	Advisory `namespace:verb` permissions (e.g. `fs:read`).
`capabilityNamespaces`	`string[]`	Custom namespaces the plugin owns.
`cron`	`object`	Scheduled `onTick` handler.
`module`	`object`	Whitelisted cross-plugin exports.
`engine`	`object`	Persistent-process reverse-proxy config.
`transport`	`object`	Enables `maw hey plugin:<name>` peer invocation.

3.2 — Validation rules

File: `src/plugin/manifest-constants.ts`:

- `NAME_RE = /^[a-z0-9-]+$/` — names are lowercase, digits, and hyphens only.
- `SEMVER_RE = /^\d+\.\d+\.\d+(?:-[\w.]+)?(?:\+[\w.]+)?$/` — standard semver with optional pre-release/build.
- `KNOWN_CAPABILITY_NAMESPACES = ["net","fs","peer","sdk","proc","ffi","tmux","shell","attach"]` — the recognised permission namespaces.
- `KNOWN_TIERS = ["core","standard","extra"]`, with `DEFAULT_TIER = "core"`.

A manifest that violates these is rejected at load time with an actionable error — not silently ignored.

3.3 — The smallest manifest that works

You saw it in Chapter 1. A working CLI plugin needs just five things: `name`, `version`, `entry`, `sdk`, and a `cli.command`. Everything else is optional.

Chapter 4 — The Invocation Contract

When you run `maw hello-oracle Nat`, the engine calls your default export with one argument — an `InvokeContext` — and expects an `InvokeResult` back. This is the entire contract between core and plugin.

4.1 — What your handler receives

File: `src/plugin/types.ts`, lines 137–162:

```
export interface InvokeContext {
  source: "cli" | "api" | "peer";
  args: string[] | Record<string, unknown>;
  matchedName?: string;   // which alias matched (alias-aware plugins)
  writer?: (...args: unknown[]) => void;  // CLI → stdout; API/peer → undefined
  flags?: Record<string, boolean | string | number | string[]>; // parsed per manifest
}
```

- `source` tells you how you were invoked — terminal (`cli`), HTTP (`api`), or peer (`peer`). Your plugin can behave differently per surface.
- `args` is the positional argument list for CLI, or a key/value object for API/peer.
- `writer` streams output to the terminal in real time when present. The documented idiom is `ctx.writer?.(...args) ?? logs.push(args.join(" "))` — stream when you can, buffer when you can't.
- `flags` is pre-parsed from your manifest's `cli.flags` schema (issue #1885), so you never re-implement argv parsing.

4.2 — What your handler returns

File: `src/plugin/types.ts`, lines 164–174:

```
export interface InvokeResult {
  ok: boolean;
  output?: string;
  error?: string;
  exitCode?: number;  // custom non-zero exit on failure; defaults to 1
}
```

`ok: true` with `output` is success. `ok: false` with `error` (and optionally a specific `exitCode`) is failure — the CLI prints the error and exits with your code, so scripts can distinguish failure modes.

4.3 — `definePlugin()` — the typed contract

File: `src/sdk/index.ts`, lines 254–299:

```
export interface PluginConfig {
  name: string;
  handler?: (ctx: InvokeContext) => Promise<InvokeResult>;
  onGate?:   (event: any) => boolean;        // Phase 0: cancel
  onFilter?: (event: any) => any;            // Phase 1: transform
  onEvent?:  (event: any) => void | Promise<void>; // Phase 2: react
  onLate?:   (event: any) => void;           // Phase 3: cleanup
  onInstall?:   () => void | Promise<void>;
  onUninstall?: () => void | Promise<void>;
}

export function definePlugin(config: PluginConfig): PluginConfig {
  if (!config.name) throw new Error("definePlugin: name is required");
  if (config.handler !== undefined && typeof config.handler !== "function") {
    throw new Error("definePlugin: handler must be a function");
  }
  return config;
}
```

The docstring describes it well: "*Like Vue's defineComponent() — validates the shape, provides autocomplete, zero runtime overhead.*" It is a pass-through that exists purely to give you type-checking and editor completion while you author.

Chapter 5 — Lifecycle & Event Hooks

A plugin can be far more than a one-shot command. It can react to fleet events and participate in session lifecycle.

5.1 — The four-phase event pipeline

File: `src/plugins/10_system.ts` — every feed event flows through four ordered phases:

1. **GATE** (`onGate`) — return `false` to cancel the event entirely.
2. **FILTER** (`onFilter`) — return a modified event; later handlers see your version.
3. **HANDLE** (`onEvent`) — observe and react (async-capable).
4. **LATE** (`onLate`) — guaranteed cleanup, runs even if earlier phases short-circuit.

This is a clean separation of concerns: veto, transform, react, clean up — in that order.

5.2 — Declaring hooks in the manifest

```
"hooks": {
  "on": ["MessageSend", "MessageDeliver", "MessageFail"]
}
```

File: `src/plugins/30_hooks-registry.ts` resolves the handler by convention. For event `MessageSend` it looks for an export named `onMessageSend` , then falls back to `onEvent` / `on` / `handle` . So you can write one specific handler per event, or a single catch-all.

5.3 — Session lifecycle

File: `src/plugin/lifecycle.ts` — four lifecycle phases let a plugin hook the session itself:

- **`wake`** — a session woke up.
- **`sleep`** — a session is going to sleep.
- **`serve`** — server startup; register HTTP routes (`ctx.http?.route(...)`) and WebSocket handlers (`ctx.ws?.on(...)`), or start a persistent process.
- **`transport`** — transport-router initialisation.

Lifecycle hooks run **in `weight` order**, lowest first, and a hook can declare `policy: "best-effort"` (log and continue) or `"fail-fast"` (abort on error).

Chapter 6 — Build Your First Local Plugin (Hands-On)

A five-minute walkthrough you can follow on any machine with `maw` installed.

Step 1 — Scaffold

```
cd ~/my-project          # any project directory
mkdir -p .maw/plugins/greet
```

Step 2 — Manifest (`.maw/plugins/greet/plugin.json`)

```
{
  "name": "greet",
  "version": "1.0.0",
  "entry": "./index.ts",
  "sdk": "^1.0.0",
  "cli": {
    "command": "greet",
    "aliases": ["hi"],
    "help": "maw greet [name] [--loud]",
    "flags": { "--loud": "boolean" }
  },
  "weight": 50
}
```

Step 3 — Handler (`.maw/plugins/greet/index.ts`)

```
import type { InvokeContext, InvokeResult } from "maw-js/plugin/types";

export default async function handler(ctx: InvokeContext): Promise<InvokeResult> {
  const args = (ctx.source === "cli" ? (ctx.args as string[]) : []) ?? [];
  const name = args[0] ?? "world";
  const loud = Boolean(ctx.flags?.loud);
  const msg = `Hello, ${name}`;
  return { ok: true, output: loud ? msg.toUpperCase() + "!" : msg };
}
```

Step 4 — Run

```
maw greet Nat
# → Hello, Nat

maw hi Nat --loud      # alias + flag
# → HELLO, NAT!
```

No install. No build. No restart. Save the file, run the command. That is the entire developer loop for local plugins.

Step 5 — Iterate

Edit `index.ts`, run again. Because the entry is imported fresh per invocation, your changes are live immediately. This is why local `.maw/plugins/` is the *recommended way to develop* a plugin before you ever publish it globally.

Chapter 7 — A Real Plugin: `workboard`

Theory is cheap. Here is a production plugin that shipped to `maw-js` `main` — the **workboard** plugin, which manages a collaborative terminal sidecar.

7.1 — Its manifest

File: `plugins/workboard/plugin.json`:

```
{
  "name": "workboard",
  "version": "0.1.0",
  "entry": "./index.ts",
  "sdk": "^1.0.0",
  "description": "Manage the Oracle Workboard sidecar powered by the maw-ssh sshx fork.",
  "author": "MEYD-605",
  "cli": {
    "command": "board",
    "aliases": ["workboard"],
    "help": "maw board open|serve|install|status|stop|password|apk [options]",
    "flags": {
      "--source": "string", "--prebuilt": "string", "--port": "number",
      "--host": "string", "--version": "string", "--password": "string",
      "--url-file": "string", "--no-open": "boolean", "--dev": "boolean"
    }
  },
  "capabilityNamespaces": ["workboard"],
  "capabilities": ["fs:read", "fs:write", "proc:spawn", "net:listen", "net:fetch", "workboard:sidecar"],
  "weight": 20,
  "license": "MIT",
  "schemaVersion": 1
}
```

Notice how much richer this is than `hello-oracle`, yet it uses the *same schema*: a command (`board`) with an alias (`workboard`), a typed flag set, and a declared capability list (`fs`, `proc`, `net` + a custom `workboard` namespace).

7.2 — Its entry handler

File: `plugins/workboard/index.ts`:

```ts
import type { InvokeContext, InvokeResult } from "maw-js/plugin/types";

export const command = {
  name: "board",
  aliases: ["workboard"],
  description: "Manage the Oracle Workboard sidecar powered by the maw-ssh sshx fork.",
};

export default async function handler(ctx: InvokeContext): Promise<InvokeResult> {
  const { cmdWorkboard } = await import("./impl");

  const logs: string[] = [];
  const origLog = console.log;
  const origError = console.error;
  console.log = (...args: any[]) => {
    if (ctx.writer) ctx.writer(...args);
    else logs.push(args.map(String).join(" "));
  };
  console.error = (...args: any[]) => {
    if (ctx.writer) ctx.writer(...args);
    else logs.push(args.map(String).join(" "));
  };

  try {
    const args = ctx.source === "cli" ? (ctx.args as string[]) : [];
    await cmdWorkboard(args);
    return { ok: true, output: logs.join("\n") || undefined };
  } catch (error: any) {
    const message = error?.message ?? String(error);
    return { ok: false, error: logs.join("\n") || message, output: logs.join("\n") || undefined };
  } finally {
    console.log = origLog;
    console.error = origError;
  }
}
```

This is the **canonical pattern** for a real plugin:

1. **Lazy-import the implementation** (`await import("./impl")`) — keeps the entry tiny and startup fast.
2. **Capture** `console.log` / `console.error` so the same implementation works whether output goes to a live terminal (`ctx.writer`) or is buffered for an API/peer response (`logs[]`).
3. **Always restore the originals in** `finally` — no leaked global state.
4. **Return a structured** `InvokeResult` instead of throwing across the boundary.

`workboard` proves the ceiling is high: a single plugin can spawn processes, listen on ports, package an Android APK, and reverse-proxy a sidecar — all through the same contract you used for a one-line "hello".

Chapter 8 — Load-Time Gates (Security)

Auto-loading code from a directory sounds dangerous. The engine mitigates it with explicit gates, enforced when a plugin is loaded.

File: `src/plugin/registry.ts` (validation path) + `src/plugin/registry-helpers.ts`:

1. **SDK semver gate** — a plugin's `sdk` range must satisfy the running SDK version (`runtimeSdkVersion()`, sourced from the SDK `package.json` and inlined at build time). A mismatch is refused with an actionable message, so a plugin built for an incompatible API never silently runs.

2. **Artifact hash gate** — for a *real* install, the engine computes the file hash (`hashFile()` → `sha256:<hex>`) and compares it to the manifest's declared `artifact.sha256`. A mismatch is rejected; an unbuilt plugin (`sha256: null`) is told to run `maw plugin build`.

3. **Dev-mode bypass** — `isDevModeInstall()` returns `true` when the install dir is a **symlink**:

```
export function isDevModeInstall(pluginDir: string): boolean {
  try {
    return lstatSync(pluginDir).isSymbolicLink();
  } catch {
    return false;
  }
}
```

   Symlinked (development) installs skip the hash check — that is what lets you edit-and-rerun freely while developing, without rebuilding a hash each save.

4. **Legacy warning** — plugins with no `artifact` field still load, but emit a one-time warning per process, so old plugins keep working while nudging maintainers forward.

The takeaway: local plugins are convenient *and* governed. Discovery is permissive (walk up, find folders); execution is gated (semver + hash, unless you opted into dev-mode via a symlink).

Chapter 9 — Troubleshooting

Things that bite people, and the fix for each.

- `✗ unknown command: <name>` — You are not inside (or below) the directory that holds `.maw/plugins/`. Check `pwd`; the engine walks *up* from cwd, never *down* into sibling trees. (See Chapter 1, Proof 3.)
- **Plugin count didn't increase** — The folder name, `name` field, or `entry` path is wrong. The folder must contain a `plugin.json` (or `plugin.ts`); `entry` must point to a file that exists.
- `name` **rejected** — Names must match `/^[a-z0-9-]+$/`. No uppercase, no underscores, no spaces.
- **SDK mismatch error** — Your `sdk` range (e.g. `^1.0.0`) doesn't satisfy the running SDK. Widen the range or update maw.
- **Hash mismatch / "run maw plugin build"** — You're doing a *real* install of an unbuilt plugin. Either build it, or develop via a **symlink** (dev-mode skips the hash check).
- **Walk stops too early** — You have a `.maw-root` marker file in an intermediate directory; the walk treats it as a hard boundary by design. Remove it or move your plugin inside the boundary.
- **Want a custom global dir** — Set `MAW_PLUGINS_DIR` to override `~/.maw/plugins` for the global slot. Local dirs are always scanned regardless.

Chapter 10 — Verification Checklist

Use this to confirm the behaviour yourself, from zero.

```
[ ] maw --version           → confirms maw is installed
[ ] mkdir -p .maw/plugins/<name>
[ ] write plugin.json       → name (a-z0-9-), version, entry, sdk, cli.command
[ ] write index.ts          → default export (ctx) => Promise<InvokeResult>
[ ] maw <command>           → runs from the project dir (no install)
[ ] cd into a subdir; rerun → still works (walk-up)
[ ] cd /tmp; rerun          → "unknown command" (proves local scope)
[ ] compare "loaded N plugins" counts → differ by exactly 1
```

If all eight pass, you have independently reproduced every claim in this book.

Appendix A — Source Map

Every file referenced, for the reader who wants to audit the engine directly. Paths are relative to the `Soul-Brews-Studio/maw-js` repository root.

Concern	File	Key symbols
Local discovery (walk-up)	`src/plugin/registry-helpers.ts`	`discoverLocalPluginDirs()`, `scanDirs()`
Discovery entry point	`src/plugin/registry.ts`	`discoverPackages()`
Manifest schema	`src/plugin/types.ts`	`PluginManifest`, `InvokeContext`, `InvokeResult`
Manifest validation	`src/plugin/manifest-constants.ts`	`NAME_RE`, `SEMVER_RE`, `KNOWN_TIERS`, `KNOWN_CAPABILITY_NAMESPACES`
Manifest loading	`src/plugin/manifest-load.ts`	`LoadedPlugin`, `plugin.ts` precedence
The contract	`src/sdk/index.ts`	`definePlugin()`, `PluginConfig`
Event pipeline	`src/plugins/10_system.ts`	4-phase `emit()`
Hook registration	`src/plugins/30_hooks-registry.ts`	convention-based handler resolution
Session lifecycle	`src/plugin/lifecycle.ts`	`wake` / `sleep` / `serve` / `transport`
Security gates	`src/plugin/registry-helpers.ts`	`runtimeSdkVersion()`, `hashFile()`, `isDevModeInstall()`
Real example	`plugins/workboard/`	`plugin.json`, `index.ts`, `impl.ts`

Appendix B — The One-Paragraph Answer

> *When you run `maw` from a directory, the CLI calls `scanDirs()`, which walks up from your current working directory collecting every `.maw/plugins/` folder it finds (stopping at a `.maw-root` marker or the filesystem root), and appends the global `~/.maw/plugins` directory. Each folder with a valid `plugin.json` / `plugin.ts` is loaded, gated by an SDK-semver check and — for real installs — an artifact hash (symlinked dev installs skip the hash). A CLI plugin's default export receives an `InvokeContext` and returns an `InvokeResult`. There is no global registration step: the set is recomputed on every invocation, which is why moving out of the directory makes the command disappear.* ***That is the whole system — and you can prove it in thirty seconds.***

> 🤖 *Lord Knight จาก ai-core → lord-knight-oracle*
>
> *Co-Authored-By: Claude Opus 4.6 (1M context) noreply@anthropic.com*