Skills

Plugins

Quick start (new to plugins?)

A plugin is just a **small code module** that extends OpenClaw with extra features (commands, tools, and Gateway RPC).

Most of the time, you'll use plugins when you want a feature that's not built into core OpenClaw yet (or you want to keep optional features out of your main install).

Fast path:

1. See what's already loaded:

```
openclaw plugins list
```

2. Install an official plugin (example: Voice Call):

```
openclaw plugins install @openclaw/voice-call
```

Npm specs are **registry-only** (package name + optional version/tag). Git/URL/file specs are rejected.

3. Restart the Gateway, then configure under `plugins.entries.<id>.config`.

See **Voice Call** for a concrete example plugin. Looking for third-party listings? See **Community plugins**.

Available plugins (official)

- Microsoft Teams is plugin-only as of 2026.1.15; install `@openclaw/msteams` if you use Teams.
- Memory (Core) — bundled memory search plugin (enabled by default via `plugins.slots.memory`)
- Memory (LanceDB) — bundled long-term memory plugin (auto-recall/capture; set `plugins.slots.memory = "memory-lancedb"`)
- **Voice Call** — `@openclaw/voice-call`
- **Zalo Personal** — `@openclaw/zalouser`
- **Matrix** — `@openclaw/matrix`
- **Nostr** — `@openclaw/nostr`
- **Zalo** — `@openclaw/zalo`
- **Microsoft Teams** — `@openclaw/msteams`
- Google Antigravity OAuth (provider auth) — bundled as `google-antigravity-auth` (disabled by default)
- Gemini CLI OAuth (provider auth) — bundled as `google-gemini-cli-auth` (disabled by default)
- Qwen OAuth (provider auth) — bundled as `qwen-portal-auth` (disabled by default)
- Copilot Proxy (provider auth) — local VS Code Copilot Proxy bridge; distinct from built-in `github-copilot` device login (bundled, disabled by default)

OpenClaw plugins are **TypeScript modules** loaded at runtime via jiti. **Config validation does not execute plugin code**; it uses the plugin manifest and JSON Schema instead. See **Plugin manifest**.

Plugins can register:

- Gateway RPC methods
- Gateway HTTP handlers
- Agent tools
- CLI commands
- Background services
- Optional config validation
- **Skills** (by listing `skills` directories in the plugin manifest)
- **Auto-reply commands** (execute without invoking the AI agent)

Plugins run **in-process** with the Gateway, so treat them as trusted code. Tool authoring guide: **Plugin agent tools**.

Runtime helpers

Plugins can access selected core helpers via `api.runtime`. For telephony TTS:

```
const result = await api.runtime.tts.textToSpeechTelephony({
  text: "Hello from OpenClaw",
  cfg: api.config,
});
```

Notes:

- Uses core `messages.tts` configuration (OpenAI or ElevenLabs).
- Returns PCM audio buffer + sample rate. Plugins must resample/encode for providers.
- Edge TTS is not supported for telephony.

Discovery & precedence

OpenClaw scans, in order:

1. Config paths

   `plugins.load.paths` (file or directory)

2. Workspace extensions

   `<workspace>/.openclaw/extensions/*.ts`

   `<workspace>/.openclaw/extensions/*/index.ts`

3. Global extensions

   `~/.openclaw/extensions/*.ts`

   `~/.openclaw/extensions/*/index.ts`

4. Bundled extensions (shipped with OpenClaw, **disabled by default**)

   `<openclaw>/extensions/*`

Bundled plugins must be enabled explicitly via `plugins.entries.<id>.enabled` or `openclaw plugins enable <id>`. Installed plugins are enabled by default, but can be disabled the same way.

Each plugin must include a `openclaw.plugin.json` file in its root. If a path points at a file, the plugin root is the file's directory and must contain the manifest.

If multiple plugins resolve to the same id, the first match in the order above wins and lower-precedence copies are ignored.

Package packs

A plugin directory may include a `package.json` with `openclaw.extensions`:

```
{
  "name": "my-pack",
  "openclaw": {
    "extensions": ["./src/safety.ts", "./src/tools.ts"]
  }
}
```

Each entry becomes a plugin. If the pack lists multiple extensions, the plugin id becomes `name/<fileBase>`.

If your plugin imports npm deps, install them in that directory so `node_modules` is available (`npm install` / `pnpm install`).

Security note: `openclaw plugins install` installs plugin dependencies with `npm install --ignore-scripts` (no lifecycle scripts). Keep plugin dependency trees "pure JS/TS" and avoid packages that require `postinstall` builds.

Channel catalog metadata

Channel plugins can advertise onboarding metadata via `openclaw.channel` and install hints via `openclaw.install`. This keeps the core catalog data-free.

Example:

```
{
  "name": "@openclaw/nextcloud-talk",
  "openclaw": {
    "extensions": ["./index.ts"],
    "channel": {
      "id": "nextcloud-talk",
      "label": "Nextcloud Talk",
      "selectionLabel": "Nextcloud Talk (self-hosted)",
      "docsPath": "/channels/nextcloud-talk",
      "docsLabel": "nextcloud-talk",
      "blurb": "Self-hosted chat via Nextcloud Talk webhook bots.",
      "order": 65,
      "aliases": ["nc-talk", "nc"]
    },
    "install": {
      "npmSpec": "@openclaw/nextcloud-talk",
      "localPath": "extensions/nextcloud-talk",
      "defaultChoice": "npm"
    }
  }
}
```

OpenClaw can also merge **external channel catalogs** (for example, an MPM registry export). Drop a JSON file at one of:

- `~/.openclaw/mpm/plugins.json`
- `~/.openclaw/mpm/catalog.json`
- `~/.openclaw/plugins/catalog.json`

Or point `OPENCLAW_PLUGIN_CATALOG_PATHS` (or `OPENCLAW_MPM_CATALOG_PATHS`) at one or more JSON files (comma/semicolon/`PATH`-delimited). Each file should contain `{ "entries": [ { "name": "@scope/pkg", "openclaw": { "channel": {...}, "install": {...} } } ] }`.

Plugin IDs

Default plugin ids:

- Package packs: `package.json` `name`
- Standalone file: file base name (`~/.../voice-call.ts` → `voice-call`)

If a plugin exports `id` , OpenClaw uses it but warns when it doesn't match the configured id.

Config

```
{
  plugins: {
    enabled: true,
    allow: ["voice-call"],
    deny: ["untrusted-plugin"],
    load: { paths: ["~/Projects/oss/voice-call-extension"] },
    entries: {
      "voice-call": { enabled: true, config: { provider: "twilio" } },
    },
  },
}
```

Fields:

- `enabled` : master toggle (default: true)
- `allow` : allowlist (optional)
- `deny` : denylist (optional; deny wins)
- `load.paths` : extra plugin files/dirs
- `entries.<id>` : per-plugin toggles + config

Config changes **require a gateway restart**.

Validation rules (strict):

- Unknown plugin ids in `entries`, `allow`, `deny`, or `slots` are **errors**.
- Unknown `channels.<id>` keys are **errors** unless a plugin manifest declares the channel id.
- Plugin config is validated using the JSON Schema embedded in `openclaw.plugin.json` (`configSchema`).
- If a plugin is disabled, its config is preserved and a **warning** is emitted.

Plugin slots (exclusive categories)

Some plugin categories are **exclusive** (only one active at a time). Use `plugins.slots` to select which plugin owns the slot:

```
{
  plugins: {
    slots: {
      memory: "memory-core", // or "none" to disable memory plugins
    },
  },
}
```

If multiple plugins declare `kind: "memory"`, only the selected one loads. Others are disabled with diagnostics.

Control UI (schema + labels)

The Control UI uses `config.schema` (JSON Schema + `uiHints`) to render better forms.

OpenClaw augments `uiHints` at runtime based on discovered plugins:

- Adds per-plugin labels for `plugins.entries.<id>` / `.enabled` / `.config`

Merges optional plugin-provided config field hints under: `plugins.entries.<id>.config.<field>`

If you want your plugin config fields to show good labels/placeholders (and mark secrets as sensitive), provide `uiHints` alongside your JSON Schema in the plugin manifest.

Example:

```
{
  "id": "my-plugin",
  "configSchema": {
    "type": "object",
    "additionalProperties": false,
    "properties": {
      "apiKey": { "type": "string" },
      "region": { "type": "string" }
    }
  },
  "uiHints": {
    "apiKey": { "label": "API Key", "sensitive": true },
    "region": { "label": "Region", "placeholder": "us-east-1" }
  }
}
```

CLI

```
openclaw plugins list
openclaw plugins info <id>
openclaw plugins install <path>                 # copy a local file/dir into ~/.o
openclaw plugins install ./extensions/voice-call # relative path ok
openclaw plugins install ./plugin.tgz           # install from a local tarball
openclaw plugins install ./plugin.zip           # install from a local zip
openclaw plugins install -l ./extensions/voice-call # link (no copy) for dev
openclaw plugins install @openclaw/voice-call # install from npm
openclaw plugins update <id>
openclaw plugins update --all
openclaw plugins enable <id>
openclaw plugins disable <id>
openclaw plugins doctor
```

`plugins update` only works for npm installs tracked under `plugins.installs`.

Plugins may also register their own top-level commands (example: `openclaw voicecall`).

Plugin API (overview)

Plugins export either:

- A function: `(api) => { ... }`
- An object: `{ id, name, configSchema, register(api) { ... } }`

Plugin hooks

Plugins can ship hooks and register them at runtime. This lets a plugin bundle event-driven automation without a separate hook pack install.

Example

```
import { registerPluginHooksFromDir } from "openclaw/plugin-sdk";

export default function register(api) {
  registerPluginHooksFromDir(api, "./hooks");
}
```

Notes:

- Hook directories follow the normal hook structure (`HOOK.md` + `handler.ts`).
- Hook eligibility rules still apply (OS/bins/env/config requirements).
- Plugin-managed hooks show up in `openclaw hooks list` with `plugin:<id>` .
- You cannot enable/disable plugin-managed hooks via `openclaw hooks` ; enable/disable the plugin instead.

Provider plugins (model auth)

Plugins can register **model provider auth** flows so users can run OAuth or API-key setup inside OpenClaw (no external scripts needed).

Register a provider via `api.registerProvider(...)` . Each provider exposes one or more auth methods (OAuth, API key, device code, etc.). These methods power:

```
openclaw models auth login --provider <id> [--method <id>]
```

Example:

```
api.registerProvider({
  id: "acme",
  label: "AcmeAI",
  auth: [
    {
      id: "oauth",
      label: "OAuth",
      kind: "oauth",
      run: async (ctx) => {
        // Run OAuth flow and return auth profiles.
        return {
          profiles: [
            {
              profileId: "acme:default",
              credential: {
                type: "oauth",
                provider: "acme",
                access: "...",
                refresh: "...",
                expires: Date.now() + 3600 * 1000,
              },
            },
          ],
          defaultModel: "acme/opus-1",
        };
      },
    },
  ],
});
```

Notes:

- `run` receives a `ProviderAuthContext` with `prompter`, `runtime`, `openUrl`, and `oauth.createVpsAwareHandlers` helpers.
- Return `configPatch` when you need to add default models or provider config.
- Return `defaultModel` so `--set-default` can update agent defaults.

Register a messaging channel

Plugins can register **channel plugins** that behave like built-in channels (WhatsApp, Telegram, etc.). Channel config lives under `channels.<id>` and is validated by your channel plugin code.

```
const myChannel = {
  id: "acmechat",
  meta: {
    id: "acmechat",
    label: "AcmeChat",
    selectionLabel: "AcmeChat (API)",
    docsPath: "/channels/acmechat",
    blurb: "demo channel plugin.",
    aliases: ["acme"],
  },
  capabilities: { chatTypes: ["direct"] },
  config: {
    listAccountIds: (cfg) => Object.keys(cfg.channels?.acmechat?.accounts ?? {}),
    resolveAccount: (cfg, accountId) =>
      cfg.channels?.acmechat?.accounts?.[accountId ?? "default"] ?? {
        accountId,
      },
  },
  outbound: {
    deliveryMode: "direct",
    sendText: async () => ({ ok: true }),
  },
};

export default function (api) {
  api.registerChannel({ plugin: myChannel });
}
```

Notes:

- Put config under `channels.<id>` (not `plugins.entries`).
- `meta.label` is used for labels in CLI/UI lists.

`meta.aliases` adds alternate ids for normalization and CLI inputs.

`meta.preferOver` lists channel ids to skip auto-enable when both are configured.

`meta.detailLabel` and `meta.systemImage` let UIs show richer channel labels/icons.

Write a new messaging channel (step-by-step)

Use this when you want a **new chat surface** (a “messaging channel”), not a model provider. Model provider docs live under `/providers/*` .

1. Pick an id + config shape

   All channel config lives under `channels.<id>` .

   Prefer `channels.<id>.accounts.<accountId>` for multi-account setups.

2. Define the channel metadata

   `meta.label` , `meta.selectionLabel` , `meta.docsPath` , `meta.blurb` control CLI/UI lists.

   `meta.docsPath` should point at a docs page like `/channels/<id>` .

   `meta.preferOver` lets a plugin replace another channel (auto-enable prefers it).

   `meta.detailLabel` and `meta.systemImage` are used by UIs for detail text/icons.

3. Implement the required adapters

   `config.listAccountIds` + `config.resolveAccount`

   `capabilities` (chat types, media, threads, etc.)

   `outbound.deliveryMode` + `outbound.sendText` (for basic send)

4. Add optional adapters as needed

`setup` (wizard), `security` (DM policy), `status` (health/diagnostics)

`gateway` (start/stop/login), `mentions`, `threading`, `streaming`

`actions` (message actions), `commands` (native command behavior)

5. Register the channel in your plugin

`api.registerChannel({ plugin })`

Minimal config example:

```
{
  channels: {
    acmechat: {
      accounts: {
        default: { token: "ACME_TOKEN", enabled: true },
      },
    },
  },
}
```

Minimal channel plugin (outbound-only):

```
const plugin = {
  id: "acmechat",
  meta: {
    id: "acmechat",
    label: "AcmeChat",
    selectionLabel: "AcmeChat (API)",
    docsPath: "/channels/acmechat",
    blurb: "AcmeChat messaging channel.",
    aliases: ["acme"],
  },
  capabilities: { chatTypes: ["direct"] },
  config: {
    listAccountIds: (cfg) => Object.keys(cfg.channels?.acmechat?.accounts ?? {}),
    resolveAccount: (cfg, accountId) =>
      cfg.channels?.acmechat?.accounts?.[accountId ?? "default"] ?? {
        accountId,
      },
  },
  outbound: {
    deliveryMode: "direct",
    sendText: async ({ text }) => {
      // deliver `text` to your channel here
      return { ok: true };
    },
  },
};

export default function (api) {
  api.registerChannel({ plugin });
}
```

Load the plugin (extensions dir or `plugins.load.paths`), restart the gateway, then configure `channels.<id>` in your config.

Agent tools

See the dedicated guide: .

Register a gateway RPC method

```
export default function (api) {
  api.registerGatewayMethod("myplugin.status", ({ respond }) => {
    respond(true, { ok: true });
  });
}
```

Register CLI commands

```
export default function (api) {
  api.registerCli(
    ({ program }) => {
      program.command("mycmd").action(() => {
        console.log("Hello");
      });
    },
    { commands: ["mycmd"] },
  );
}
```

Register auto-reply commands

Plugins can register custom slash commands that execute **without invoking the AI agent**. This is useful for toggle commands, status checks, or quick actions that don’t need LLM processing.

```
export default function (api) {
  api.registerCommand({
    name: "mystatus",
    description: "Show plugin status",
    handler: (ctx) => ({
      text: `Plugin is running! Channel: ${ctx.channel}`,
    }),
  });
}
```

Command handler context:

- `senderId` : The sender's ID (if available)
- `channel` : The channel where the command was sent
- `isAuthorizedSender` : Whether the sender is an authorized user
- `args` : Arguments passed after the command (if `acceptsArgs: true`)
- `commandBody` : The full command text
- `config` : The current OpenClaw config

Command options:

- `name` : Command name (without the leading `/`)
- `description` : Help text shown in command lists
- `acceptsArgs` : Whether the command accepts arguments (default: false). If false and arguments are provided, the command won't match and the message falls through to other handlers
- `requireAuth` : Whether to require authorized sender (default: true)
- `handler` : Function that returns `{ text: string }` (can be async)

Example with authorization and arguments:

```
api.registerCommand({
  name: "setmode",
  description: "Set plugin mode",
  acceptsArgs: true,
  requireAuth: true,
  handler: async (ctx) => {
    const mode = ctx.args?.trim() || "default";
    await saveMode(mode);
    return { text: `Mode set to: ${mode}` };
  },
});
```

Notes:

- Plugin commands are processed **before** built-in commands and the AI agent
- Commands are registered globally and work across all channels
- Command names are case-insensitive (`/MyStatus` matches `/mystatus`)
- Command names must start with a letter and contain only letters, numbers, hyphens, and underscores
- Reserved command names (like `help` , `status` , `reset` , etc.) cannot be overridden by plugins
- Duplicate command registration across plugins will fail with a diagnostic error

Register background services

```
export default function (api) {
  api.registerService({
    id: "my-service",
    start: () => api.logger.info("ready"),
    stop: () => api.logger.info("bye"),
  });
}
```

Naming conventions

- Gateway methods: `pluginId.action` (example: `voicecall.status`)
- Tools: `snake_case` (example: `voice_call`)
- CLI commands: kebab or camel, but avoid clashing with core commands

Skills

Plugins can ship a skill in the repo (`skills/<name>/SKILL.md`). Enable it with `plugins.entries.<id>.enabled` (or other config gates) and ensure it's present in your workspace/managed skills locations.

Distribution (npm)

Recommended packaging:

- Main package: `openclaw` (this repo)
- Plugins: separate npm packages under `@openclaw/*` (example: `@openclaw/voice-call`)

Publishing contract:

- Plugin `package.json` must include `openclaw.extensions` with one or more entry files.
- Entry files can be `.js` or `.ts` (jiti loads TS at runtime).

`openclaw plugins install <npm-spec>` uses `npm pack`, extracts into `~/.openclaw/extensions/<id>/`, and enables it in config.

Config key stability: scoped packages are normalized to the **unscoped** id for `plugins.entries.*`.

Example plugin: Voice Call

This repo includes a voice-call plugin (Twilio or log fallback):

- Source: `extensions/voice-call`
- Skill: `skills/voice-call`
- CLI: `openclaw voicecall start|status`
- Tool: `voice_call`
- RPC: `voicecall.start`, `voicecall.status`
- Config (twilio): `provider: "twilio"` + `twilio.accountSid/authToken/from` (optional `statusCallbackUrl`, `twimlUrl`)
- Config (dev): `provider: "log"` (no network)

See **Voice Call** and `extensions/voice-call/README.md` for setup and usage.

Safety notes

Plugins run in-process with the Gateway. Treat them as trusted code:

- Only install plugins you trust.
- Prefer `plugins.allow` allowlists.
- Restart the Gateway after changes.

Testing plugins

Plugins can (and should) ship tests:

In-repo plugins can keep Vitest tests under `src/**` (example: `src/plugins/voice-call.plugin.test.ts`).

Separately published plugins should run their own CI (lint/build/test) and validate `openclaw.extensions` points at the built entrypoint (`dist/index.js`).

‹ ClawHub Community plugins ›