

Yes—Kiduna should eventually have a plugin bundle similar to Vercel’s, but the foundation should be a secure, remote Kiduna MCP server. That gives Codex, Claude Code, ChatGPT, and future agent clients one consistent way to authenticate, discover Kiduna capabilities, and act on behalf of people and organizations.

The plugin then teaches each agent how to use those capabilities well.

```
None
```mermaid
flowchart LR
    U["Person using Codex or Claude"] --> P["Kiduna plugin"]
    P --> S["Shared Kiduna skills"]
    P --> A["Platform-specific agents, commands, and hooks"]
    S --> M["Kiduna MCP server"]
    A --> M
    M --> O["OAuth and delegated authority"]
    M --> K["Kiduna API"]
    K --> D["People, agents, groups, initiatives, decisions, and audit history"]
```
```

## The central architecture

Think of the pieces this way:

| Component        | Purpose                                                  |
|------------------|----------------------------------------------------------|
| Kiduna API       | The actual platform and source of truth                  |
| MCP server       | Secure, agent-friendly access to Kiduna                  |
| Skills           | Reusable instructions for accomplishing Kiduna workflows |
| Commands         | Explicit shortcuts such as <code>/kiduna:status</code>   |
| Agents           | Specialists for larger, delegated jobs                   |
| Hooks            | Lightweight automatic behavior and safety checks         |
| Plugin manifests | Packaging for Codex and Claude Code                      |

Marketplace

Installation and updating

MCP is the most important component because it is an open interoperability layer. Codex and Claude Code both support remote HTTP MCP servers with OAuth. [OpenAI MCP documentation](#), [Claude Code MCP documentation](#)

## One repository, two native plugin packages

I would use a monorepo with a common core and small platform-specific adapters:

None

```
kiduna-agent-plugins/
├── .agents/
│   └── plugins/
│       └── marketplace.json           # Codex marketplace
├── .claude-plugin/
│   └── marketplace.json             # Claude marketplace
├── plugins/
│   └── kiduna/
│       ├── .codex-plugin/
│       │   └── plugin.json
│       ├── .claude-plugin/
│       │   └── plugin.json
│       ├── .mcp.json
│       └── .app.json                 # Future ChatGPT hosted
└── integration                       # Shared wherever possible
    ├── skills/
    ├── commands/
    ├── agents/
    │   ├── codex/
    │   └── claude/
    ├── hooks/
    │   ├── codex.json
    │   └── claude.json
    ├── scripts/
    ├── references/
    ├── assets/
    └── tests/
```

```
└─ packages/
   └─ mcp-server/
   └─ api-client/
   └─ schemas/
```

Skills are the most portable component because both platforms use `SKILL.md` and the open Agent Skills format. Commands, agents, and hooks have overlapping concepts but slightly different schemas, so I would generate their platform-specific versions from shared source material rather than trying to make every file identical.

Codex requires `.codex-plugin/plugin.json`; Claude Code uses `.claude-plugin/plugin.json`. Both expect components at the plugin root. Codex also recognizes a legacy-compatible Claude marketplace location, which is encouraging for interoperability, but native manifests remain safer. [OpenAI plugin structure](#), [Claude Code plugin reference](#)

## The Kiduna MCP server

I would host this at something like:

None

`https://mcp.kiduna.ai/mcp`

It should expose purpose-built actions—not generic database access or a broad `execute_action` tool.

### Identity and authority

- `get_my_identity`
- `get_effective_permissions`
- `list_my_delegations`
- `preview_delegation`
- `grant_delegation`
- `revoke_delegation`
- `list_agent_activity`

This is especially important for Kiduna. Every agent action should answer:

- Who is acting?

- On whose behalf?
- Within which organization?
- Under what mandate?
- With which permissions?
- When does that authority expire?

A Kiduna agent should never inherit every permission its human possesses.

## People and relationships

- `search_people_and_agents`
- `get_profile`
- `list_connections`
- `request_connection`
- `request_introduction`
- `list_invitations`

Search results must respect privacy and relationship visibility rules.

## Groups and organizations

- `list_my_organizations`
- `get_organization_context`
- `create_working_group`
- `invite_member`
- `change_member_role`
- `list_members`
- `get_governance_rules`

## Initiatives and coordination

- `list_initiatives`
- `create_initiative`
- `get_initiative_context`
- `create_task`
- `assign_task`
- `update_task`
- `post_update`
- `propose_action`
- `record_decision`

## Decisions and governance

- `create_proposal`
- `list_open_proposals`
- `preview_vote`
- `cast_vote`
- `record_consent`
- `get_decision_history`

Financial, membership, publishing, voting, and irreversible actions should require explicit confirmation.

## Notifications and auditability

- `list_notifications`
- `mark_notification_read`
- `get_audit_events`
- `explain_action_authority`

Every write should return an audit ID, acting identity, organization, timestamp, and authority used.

## Skills I would include first

I would not begin with 30 skills. Start with six or eight that represent Kiduna's most important user journeys:

1. `kiduna-get-oriented`  
Explain the user's identity, memberships, active initiatives, notifications, and available authority.
2. `kiduna-find-collaborators`  
Find relevant people or agents while respecting relationship and privacy boundaries.
3. `kiduna-form-a-group`  
Turn an intention into a working group with members, purpose, roles, and initial norms.
4. `kiduna-launch-an-initiative`  
Create a structured initiative, invite collaborators, define outcomes, and create initial work.
5. `kiduna-run-a-decision`  
Prepare a proposal, identify the applicable governance rule, gather input, and record the result.
6. `kiduna-coordinate-work`  
Review current work, surface blockers, assign tasks, and publish a progress update.

7. `kiduna-delegate-to-an-agent`

Create a narrowly scoped, time-limited mandate for an agent.

8. `kiduna-audit-agent-activity`

Explain what an agent did, under whose authority, and what changed.

A second plugin, `kiduna-developer`, could eventually teach coding agents how to build applications and integrations on Kiduna.

## Specialized agents

After the workflows are stable, I would add three or four specialists:

- **Kinship Mapper** — finds relevant relationships and potential collaborators.
- **Organization Steward** — helps structure groups, roles, governance, and membership.
- **Initiative Coordinator** — keeps plans, tasks, decisions, and updates coherent.
- **Authority Auditor** — reviews delegated permissions and agent activity.

Agents should orchestrate skills and MCP tools; they should not introduce secret capabilities unavailable through the MCP permission system.

## Commands

Useful explicit commands might include:

None

```
/kiduna:status  
/kiduna:people  
/kiduna:form-group  
/kiduna:launch  
/kiduna:decide  
/kiduna:delegate  
/kiduna:audit
```

Skills should remain usable without commands, since both Codex and Claude can select a skill automatically based on its description.

## Hooks

Keep hooks sparse and predictable. Vercel's plugin deliberately shifted toward lightweight session-start behavior rather than injecting information on every prompt or tool call. That is a good model for Kiduna. [Vercel plugin architecture](#)

Good Kiduna hooks:

- **SessionStart:** detect whether the current repository is connected to a Kiduna organization or initiative and provide a short status.
- **PreToolUse:** block or warn when an agent attempts a Kiduna write without a clear organization or mandate.
- **PostToolUse:** record a local audit reference after important Kiduna actions.
- **Stop:** optionally remind the agent about unpublished changes or unfinished delegated work.

Avoid injecting the entire relationship graph or organization history into every turn. Retrieve context on demand.

Codex intentionally supports compatibility variables such as `CLAUDE_PLUGIN_ROOT`, but the precise event and matcher support differs between hosts. Therefore, shared hook scripts are reasonable; separate hook configuration files are safer.

## Authentication and security

Use OAuth 2.1 with authorization code and PKCE for people. Use client credentials only for explicitly registered service agents with no interactive human.

Recommended scopes:

None

```
identity:read
profile:read
relationships:read
relationships:write
organizations:read
organizations:write
initiatives:read
initiatives:write
decisions:read
decisions:write
delegations:read
delegations:write
```

`audit:read`

Important safeguards:

- Short-lived access tokens and rotating refresh tokens.
- Organization and resource-level authorization on every tool call.
- No tokens stored in the plugin repository.
- No token passthrough to downstream services.
- Idempotency keys for writes.
- Preview/commit separation for consequential actions.
- Rate limiting per person, agent, organization, and installation.
- Immutable audit events.
- Strict tenant isolation.
- Minimal, paginated tool responses.
- Explicit confirmation for publishing, invitations, voting, money, deletion, and permission changes.

The MCP specification recommends OAuth authorization, resource-specific token validation, least-privilege scopes, HTTPS, and avoiding token passthrough. [MCP authorization guidance](#)

Each tool should also declare accurate MCP annotations:

- `readOnlyHint`
- `destructiveHint`
- `idempotentHint`
- `openWorldHint`

These help clients present appropriate approval behavior, although they are hints—not substitutes for server-side authorization. [MCP tool schema](#)

## Development sequence

I would build this in stages:

1. Define Kiduna's identity, organization, delegation, and audit models.
2. Build a conventional authenticated Kiduna API.
3. Add a read-only remote MCP server.
4. Connect OAuth and granular scopes.
5. Add a small set of safe write tools using preview/commit.
6. Write the shared `SKILL.md` workflows.
7. Add the Codex and Claude manifests.
8. Add platform-specific commands, agents, and minimal hooks.



9. Create fixtures and automated MCP contract tests.
10. Test both plugins locally.
11. Publish a private Kiduna marketplace for early users.
12. Gather real usage and add capabilities gradually.
13. Submit to the OpenAI and Anthropic public directories when the product and policies are ready.

For public OpenAI submission, prepare production branding, support/privacy/terms pages, accurate tool annotations, starter prompts, and five positive plus three negative test cases.

[OpenAI plugin submission](#)

Claude supports independent Git-hosted marketplaces, allowing users to add the marketplace and install versioned plugins. [Claude marketplace documentation](#)

## My strongest recommendation

Build **Kiduna MCP** as the platform product and treat the Codex/Claude plugins as elegant clients of it.

That avoids putting business logic into dozens of markdown files, gives every agent client the same permission and audit model, and lets Kiduna become infrastructure for agentic organizations rather than merely a bundle of prompts.