

[Skip to content](#)

# Chat history

This is a copy of a conversation between ChatGPT & Anonymous.

[Report conversation](#)

**You said:**

Help me with strategic planning of next steps for the taew-py library.

Background: my primarily business is providing "Rental CTO" services to early-stage startups. This service usually includes rapid development of initial versions of Minimal Evolvable Product (MEP - probably a more accurate name than classical MVP) without premature technology lock-in through systematic application of the Ports and Adapters architecture pattern. The taew-py library provides support for such rapid, CLI-first, development in Python 3.14 and above. I also plan to develop similar libraries in other languages such as TypeScript.

The current version of taew-py could be found in <https://github.com/asterkin/taew-py>. Confirm that the background is clear or ask clarifying questions if required before we proceed.

**ChatGPT said:**

Thanks — the background is clear, but before we start planning next steps strategically, I want to confirm a few details to ensure we anchor the plan in your actual constraints and your intended positioning.

Here are the clarification questions; please answer any that matter:

## 1. Target Users

You mentioned *early-stage startups* and *Rental CTO services*.

Do you envision taew-py being used:

- Primarily by *you* during engagements,
- Or also by *external teams* (founders, junior devs, startup hires) who may adopt it for their own projects?

## 2. Scope of abstraction

taew-py currently offers:

- dynamic port loading,
- adapters for collections, logging, filesystem, cryptography,

- experimental channel abstractions (pipes, SQS),
- multiprocessing matrix infrastructure,
- library structure aligned with Hexagonal architecture.

Do you want to:

- Keep taew small and orthogonal, providing only essential core primitives,
- Or evolve it into a full-stack MEP toolkit, with batteries included (CLI scaffolding, project generator, common adapters, local cloud emulation, etc.)?

### 3. Time investment

How much *ongoing weekly capacity* do you want to dedicate to maintaining and evolving taew-py?

- 0.5 day / wk
- 1 day / wk
- 2+ days / wk
- Opportunistic only

### 4. Adapter strategy (major open question from your memory)

You noted concerns about packaging strategy — avoiding bloat while still providing adapters:

- S3
- SQS
- PostgreSQL (local/docker + cloud)
- Logging
- Pipelines
- CLI

Which philosophy do you prefer?

1. "taew-core" + many small plugin packages  
e.g., `taew-core`, `taew-s3`, `taew-postgres`, `taew-sqs`.
2. One monolithic library with optional imports, relying on lazy loading.
3. Hybrid: core + a curated "default adapters" bundle.

### 5. Python-first or multilingual?

Where do you want taew-py to sit compared to future TypeScript, JVM, C++ implementations?

- Python is the *flagship*, others follow later
- All languages share a unified spec (like OpenTelemetry)
- TypeScript will become equally important soon

## 6. The role of dynamic wiring

You have dynamic loaders for ports and aspects; you also mentioned future support for static wiring.

Is the long-term vision:

- Dynamic-first, static optional,
- or Static-first for production, dynamic for prototyping?

## 7. Expected marketing trajectory

Do you want taew-py to:

- Eventually become a public-facing framework (like FastAPI, Typer, Pydantic)?
- Or stay a private professional toolkit that only select clients see?

## 8. CLI-first philosophy

Should taew-py provide:

- A generic CLI framework?
  - A project template?
  - A recommended file layout?
  - Or remain completely unopinionated?
- 

Once I have these answers, I will craft:

## A strategic roadmap under 3 horizons

- Horizon 1 (next 2–4 weeks): high-leverage foundational actions
- Horizon 2 (1–3 months): ecosystem build-out and adapter strategy
- Horizon 3 (6–12 months): unification with other languages and business alignment

## Plus:

- Risk analysis
- Opportunity analysis
- Clear success criteria
- Optional: business-facing presentation version of the roadmap (for clients/investors)

Let me know your answers — or say “use reasonable assumptions” if you prefer me to proceed directly.

### You said:

1. Target users: myself at initial stages of engagement, prospect customers to continue product evolution by themselves, hopefully academic freshers who are going to be challenged to find an entry level jobs due to the current AI Agents trend (whether this will be a pure charity/back to community or could be an additional business model is TBD).
  2. Scope of abstraction: taew-py is planned to stay at the level of Python standard library (kind of language extension making it Ports and Adapters friendly). Specific 3rd party technology adapters (e.g. aws, gcp, azure, etc.) are planned as separate projects.
  3. Time investment: I normally commit 50% of my time to the current customer leaving other 50% for personal life and taew. For that reason, I'm looking for hyper productivity in taew development assisted by AI Agents and hopefully attracting some Open-Source contributors (see academic freshers concerns above).
  4. Adapter Strategy: separate projects per 3rd party technology keeping the taew-py core at the Python stdlib level.
  5. The language question is a part of strategic issue I want to discuss. Considering my personal time constraints alone I won't be able to implement all what is possible/required. Some switches will be opportunistic caused by the paying customer needs.
  6. Wiring strategy: dynamic first, static as deployment optimization (not always required)
  7. It's a back-to-community Open-Source project. I would be glad to attract young generation and contributors to it, but as a business activity it serves my Rental CTO needs.
  - 8 taew-py already has it and it will evolve. CLI-first very often is the easiest way to explore core product workflows. Once clarified, it could be extended to TUI, Web or whatever is required adapters.
- Do not craft anything yet. I have a fundamental strategic issue I want your opinion on.
- Let's clarify the background first and then proceed to it.

### ChatGPT said:

Understood — and thank you for the clarifications.

Your background and constraints are now very clear, and we can proceed to the strategic issue when you're ready.

Let me restate the clarified background in concise form so we both operate with the same mental model:

---

## **taew-py: Clarified Strategic Context**

### **Purpose**

A Python 3.14+ “language extension” that makes *Ports & Adapters* natural and fast, enabling rapid development of Minimal Evolvable Products (MEPs) for early-stage startups.

Used by:

- You (Rental CTO) for rapid prototyping.
- Prospect customers to evolve the product after handoff.
- A potential community of newcomers (juniors/academic freshers) for skill-building and contributions.

### **Scope**

- taew-py core stays at roughly the “standard library + hexagonal tooling” level.
- Third-party adapters (AWS/GCP/Azure/Postgres/SQS/etc.) are separate repositories.
- Goal: clean, orthogonal, minimal core — extensible outward via plugins.

### **Time Constraints**

- ~50% of working time is available for taew + personal projects.
- High leverage is essential.
- AI assistance and OS contributors will be key.

### **Architecture Positioning**

- Dynamic wiring by default, static optionally for deployment optimization.
- CLI-first, then TUI/web/etc. interfaces via additional adapters.
- Multilingual expansion is desired but opportunistic, aligned with customer needs.

## Business/Community Orientation

- taew-py is open-source and community-friendly, but practically serves as your toolkit for delivering Rental CTO engagements.
  - If it attracts junior contributors, even better — both as community impact and potential future collaborators.
- 

Everything is consistent and well-defined.

You mentioned you have one fundamental strategic issue you want my opinion on.

I'm ready — what is the strategic question you want to explore?

**You said:**

My major strategic issue is finding a reasonable path to make the library AI Native.

These are my, partially conflicting, observations:

1. New frameworks have no chance to cut through the current AI Agents trend since it is not in the LLM training set and won't be there since nobody is using it because AI Agents do not suggest it.
2. The only way to break through this vicious cycle is for library to come with its own customization of leading AI Agents (Claude CLI, Codex, Gemini).
3. The taew-py/HELLO\_TAEW\_PY.md at main · asterkin/taew-py tutorial was the first attempt to make Hello World tutorial AI Agent friendly such that every step is actually a prompt to be submitted to the agent. Humans are not supposed to execute these steps manually anymore.
4. While Claude CLI and Codex CLI are capable of executing these prompts - they are still imperfect and waste too many tokens. Sometimes lead to mistakes a new user won't be able to fix.
5. It could be possible to reverse the sequence. Rather than to install the library, the user will be provided with a simple prompt to install taew-py Claude plugin which comes with all skills, may be sub-agents and slash commands required for taew-py application or 3rd party adapters development. The Hello World tutorial then will be a sequence of typical prompts to activate these skills (e.g. project init, defining domain data structures, ports, workflows and adapters) thus creating a basis for real development.
6. And that what creates a strategic problem. The AI Agents landscape is unstable and constantly changing, major players are releasing new features at high speed but are

unequal in terms of features (e.g. only Claude has plugins which seem to be the perfect starting point). In addition, multiple second tier players (e.g. Kilo Code) are trying to fill the gaps.

7. Sticking with one AI Agent CLI, say Claude CLI, seems to be contradicting the core taew philosophy of isolating 3rd party technologies in adapters. It also poses a fundamental risk of betting on the wrong horse. On the other hand, trying to support all major players, say Claude, Codex and Gemini, would stretch even further limited timing and financial resources (I could afford Pro editions of Claude and OpenAI plus Github Copilot). Just keeping up with all news in this field is a full-time job. Interesting to notice, that taew seems to be less in need for co-pilots since most of the code is usually generated by agents but requires tight supervision.

8. If I decide to stick with one player. There are many technical questions to address:

1. Should it be a part of taew-py core project or a separate one?
2. Proper responsibility allocation between context (aka CLAUDE.md etc files), configuration, skills, sub-agents, slash commands and scripts (bash vs taew cli). The current setup is definitely sub-optimal and too often leads to exhausting session limits on trivial problems. Trying to resolve all issues would stop the project development for long time. Some roadmap is required. Needless to say that core taew skills etc. should be equally applicable to applications, 3rd party adapters and taew itself (add new Python adapter).
3. Optimal model selection. Currently, it works with the default Sonnet 4.5. With proper allocation between skills, scripts, m.b. sub-agents and slash commands, may less expensive and demanding models (Haiku) could be delegated to. Currently, it's unclear when and how to do this.

9. To sum I have three major concerns:

1. Cannot ignore that AI Agents trend and need to lockstep taew development with it from the day one
2. One vendor vs cohort dilemma
3. Even with one vendor selected, there is a danger of swallowing all available resources on tools rather than product

10. Please, voice your opinion on this. Ask clarifying questions if any.

ChatGPT said:

This is the right strategic issue to surface — and you're thinking about it in exactly the right depth.

Let me respond carefully, because this decision determines whether taew-py becomes:

- AI-native, discoverable, and relevant...  
or
- Just another brilliant framework the LLMs never mention.

Your summary is accurate, and the dilemma is real.

Here is my considered opinion, with no sugar-coating and no premature planning — just strategic guidance.

---

## ★ **My Strategic Opinion: The AI-Native Path Must Be Real, But Narrow**

And it must not be “vendor support.”

It should be taew’s own adapter layer for AI agents themselves, exactly like any other 3rd-party technology.

This is the only approach that preserves your philosophy *and* your sanity.

Let me address your concerns one by one.

---

### **1. Your observation is correct: a framework outside the LLM training set is invisible.**

Every agent (“Write a Python CLI app...”) suggests the same frameworks because:

- They are in the pretraining data
- Lots of developers use them
- Agents self-reinforce what agents suggest

This is a feedback loop, and new frameworks cannot enter unless:

→ **They ship their own agent integration**

→ **They become discoverable *through their own AI affordances*, not via training.**

Your HELLO\_TAEW tutorial experiment is exactly the correct direction.

But we can do it more systematically.



You cannot rely on “maybe one day models will ingest taew-py.”

That will not happen unless it first becomes commonly used.

So taew must be an AI-first framework, not AI-assisted.

---

## **2. Your idea of a taew-py custom plugin / agent / skills package is correct.**

But the concept needs refinement.

The model should be:

### **taew-py ships:**

- A taew agent schema
- A set of commands (`/init`, `/port`, `/workflow`, `/adapter`, etc.)
- A domain-driven prompt library
- Context files for self-use (design docs, architecture principles, naming conventions)
- Scriptable plans that an agent can execute deterministically

This makes taew:

- AI-discoverable
- AI-usable
- AI-extendable

This is precisely what *new frameworks* must do in the AI-native era.

BUT...

---

## **3. You absolutely should NOT support all AI agents. That would destroy the project.**

This is essential:

**You need one “first-tier” agent tightly integrated. Only one.**

Because:

- Each vendor (Claude/OpenAI/Google etc.) has incompatible agent frameworks
- Supporting 2–3 engines is *much* more work than maintaining taew itself
- You risk becoming an “AI tools builder” instead of building taew
- Half your time is already allocated to paying customer work

**Stick to one vendor now.**

Later:

- Provide a *taew-agent protocol* so others can build adapters around it
- But *you* maintain only one canonical integration.

Which one?

---

## 4. The correct strategic choice today is: *Claude first.*

Reasoning:

**Why Claude is uniquely aligned with taew’s needs:**

- It is the only major agent framework that currently:
  - Has real plugin support
  - Supports agentic workflows out-of-the-box
  - Can run structured commands (slash commands)
  - Has a CLI designed for development workflows
- Claude is exceptionally strong at:
  - Architectural reasoning
  - Refactoring
  - Generating protocol-oriented and adapter-oriented code (your core philosophy)

**Practical reasons:**

- Your HELLO\_TAEW tutorial already aligns with Claude CLI
- Sonnet is very strong at multi-step reasoning
- Haiku is incredibly cost-effective for repetitive structured generation

**Vendor risk?**

Low.

Anthropic is not disappearing.

If OpenAI adds first-class plugin support later, you can add an adapter.

### **But right now?**

Claude is *years ahead* in agentic development tooling.

---

## **5. But — and this is crucial — Claude integration should NOT live inside taew-py core.**

It should be:

**a separate project:** `taew-agent-claude`

Why?

1. Keeps the core pure
2. Prevents vendor lock-in *inside the framework*
3. Allows a future `taew-agent-openai` or `taew-agent-gemini`
4. Matches your philosophy: 3rd-party tech = adapters
5. Prevents a marketing mistake: taew-py should not appear “Claude-only”

Core principle:

taew-py is AI-native through adapters, not through vendor entanglement.

---

## **6. Your concern about resource drain is legitimate. The solution is: limit scope aggressively.**

The taew-agent system should:

### **Only automate the steps that:**

- New users struggle with
- Are 100% deterministic
- Speed up your own consulting work

Examples:

- Project initialization
- Port definition
- Workflow skeleton generation
- Adapter scaffold generation
- Tests layout
- Documentation layout
- Static analyzer setup
- CLI command generation

### Do NOT automate:

- Complex architectural choices
- Long-running interactive sessions
- Anything that burns tokens unnecessarily

The agent must be a *precision tool*, not a general-purpose assistant.

---

## 7. Model selection: treat models like you treat adapters.

- Sonnet for reasoning-heavy tasks
- Haiku for boilerplate generation
- Provide a thin abstraction so model choice can shift automatically

Your agent should decide:

```
/workflow add PaymentWorkflow
# uses Sonnet
```

```
/adapter scaffold filesystem
# uses Haiku
```

This mirrors taew's philosophy exactly.

---

## 8. The real long-term strategy is this: A portable taew-agent-schema.

Not per-vendor tools.

A vendor-neutral protocol:

```
{  
  "command": "create_port",  
  "name": "ForCalculatingPrices",  
  "module": "domain.ports"  
}
```

Each vendor integration is simply:

- A thin adapter around the schema
- One plugin you officially maintain (Claude)
- Others optional

This keeps taew multivendor-ready without you having to do the work now.

---

## 9. Your three concerns, addressed directly

**(1) You must sync taew with the AI Agents trend — this is mandatory.**

Yes.

Ignoring agentic workflows = invisibility.

**(2) One-vendor-vs-many: choose one (Claude) + define a neutral protocol.**

This resolves both risk and resource constraints.

**(3) Fear of drowning in tools: scope the agent tightly.**

Focus only on:

- scaffolding
- deterministic workflows
- boilerplate
- architectural glue

Not on becoming a universal AI IDE.

---

## 10. My final opinion, stated crisply

- ✓ taew must be AI-native from day one
- ✓ You should integrate tightly with only one vendor initially (Claude)
- ✓ Keep the integration outside of taew core
- ✓ Define a vendor-neutral taew-agent schema for future-proofing
- ✓ Scope agent features strictly to avoid tool-building rabbit holes
- ✓ Use AI to amplify your consulting productivity, not consume it
- ✓ Design the agent so juniors can onboard quickly and contribute safely

This is the only strategy that:

- Keeps taew pure
- Gives it AI visibility
- Protects your time
- Minimizes vendor risk
- Enables future expansion
- Makes it attractive to junior contributors

---

If you want, I can now help you articulate:

1. A clear AI-Native strategy statement
2. A roadmap for integrating Claude
3. A minimum viable taew-agent design
4. A risk analysis of different AI vendor choices

## 5. A project structure for the adapter (`taew-agent-claude`)

## 6. A vendor-neutral taew-agent protocol

Just tell me which direction you want to take next.

You said:

I accept the idea of focusing on specialized taew Claude plugin. Other AI Agents/Model, such as Codex CLI, Gemini CLI or Github Copilot could still be incorporated into specific workflows where appropriate. My next question is about marketplace repository. There are two options:

1. Generic claude-taew with taew-py plugin
2. Specialized claude-taew-py with taew plugin

What are pro and contra of each option and which one would you recommend?

ChatGPT said:

Below is a clean, strategic comparison of both naming options, followed by a recommendation grounded in your goals, taew philosophy, future multilingual plans, and developer psychology.

---

### Option 1 — `claude-taew` (generic)

The plugin is “taew for Claude,” independent of language.

#### ✓ Pros

1. Language-agnostic branding
  - Leaves room for future `taew-ts`, `taew-cpp`, `taew-java` without multiplying plugin repos.
  - Keeps “taew” as a unified concept in AI-agent space.
2. Clarifies that the plugin is part of the Claude ecosystem
  - Good for marketplace search (“claude <framework>”).
3. Avoids fragmentation
  - One repo handles:
    - taew-py
    - taew-ts
    - future taew integrations
  - Much easier to maintain the AI-spec layer once.
4. More in line with taew philosophy

- The plugin is an *adapter* for the AI agent itself.
  - Should sit next to taew, not inside a specific implementation.
5. Better marketing
    - “Claude plugin for taew” sounds like *taew* is the framework, not *Python*.
    - This elevates taew as a concept.

## ✖ Cons

1. Slightly abstract for first-time users
  - People may wonder: “Is this for Python or TS or both?”
  - Requires docs that clarify: *first implementation targets taew-py*.
2. Responsibility creep risk
  - If you later add TypeScript, C++, etc., plugin logic could get messy if not modularized correctly.

---

## Option 2 — `claude-taew-py` (Python-specific)

The plugin is explicitly for taew-py.

## ✓ Pros

1. Absolute clarity for Python developers
  - No ambiguity: this plugin is for taew-py.
2. Lower initial cognitive load
  - New users instantly understand scope and expectations.
3. Better for quick onboarding in your current business model
  - Right now, taew-py is the fastest path for your consulting workflows.

## ✖ Cons

1. Locks you into the Python namespace prematurely
  - When taew-ts or taew-cpp appear, you’ll need:
    - `claude-taew-ts`
    - `claude-taew-cpp`
    - etc.
2. Dilutes the concept of taew as a unified architecture philosophy
  - Makes it look like “taew = Python thing,” which is not what you want long-term.
3. Increases maintenance burden
  - Separate plugin repos for each language = duplicated agent logic.



- A vendor-neutral agent schema becomes harder to share.
4. Confuses the architectural abstraction
- The AI agent becomes “just another Python adapter,” not a cross-language taew adapter.
- 

## ★ Strategic Recommendation

**Choose:** `claude-taew` (generic)

With the following design principles:

### 1. The repo is language-agnostic

But contains per-language submodules, like:

```
claude-taew/  
  spec/                # vendor-neutral taew-agent schema  
  taew_py/             # taew-py bindings and commands  
  taew_ts/             # future  
  taew_cpp/            # future
```

### 2. Marketplace listing title:

Claude Plugin: taew (Python support included)

This preserves clarity *and* long-term strategic thinking.

### 3. Keep taew-py as the first implementation

No extra confusion—just document that:

The first supported language is Python (taew-py).

Additional languages will be added as needed.

### 4. This aligns 100% with taew philosophy

- taew core is technology-neutral
  - language implementations are adapters
  - so the Claude integration should also be adapter-first, not language-first
-

## Why this is the right long-term move

You are not building a Python framework.

You are building an architectural pattern framework with language implementations.

If the AI plugin is Python-first, it unintentionally signals:

“taew is a Python thing.”

But your vision is clearly broader:

- Python now
- TypeScript soon
- C++ maybe
- Possibly C#/Kotlin (for VR/mobile integrations)

So the AI plugin must reflect the philosophy, not the current implementation.

---

## One practical improvement

To avoid confusion for Python users, provide:

### A pip-installable helper:

```
pip install taew-py-agent
```

which simply:

- configures Claude CLI
- installs the `claude-taew` plugin
- adds Python bindings
- exposes `/taew-py <commands>`

This keeps:

- developer experience: simple
  - plugin architecture: clean
  - long-term roadmap: clear
- 

## Final Answer

- ✓ Use `claude-taew`
  - ✓ Make it language-agnostic
  - ✓ Put taew-py support inside a `taew_py/` submodule
  - ✓ Use marketplace phrasing: *"Claude plugin for taew, with Python support"*
  - ✓ Provide a convenience package for Python users (`taew-py-agent`)
  - ✓ Keep the agent schema vendor-neutral to support future multi-agent adapters
- 

If you'd like, I can help you next with:

- Designing the folder structure for `claude-taew`
- Defining the taew-agent schema
- Establishing initial slash commands and skills
- Drafting the marketplace listing text
- Crafting a minimal version roadmap (v0.1 → v0.2 → v1.0)

Just tell me where you want to go next.

You said:

It looks like language-specific `claude-taew-py` plugin repository is more pragmatic for near-term than a generic `claude-taew`. The real value of the latter one would come when and if language choice decision is going to be made/recommend by an AI Agent based on the application specification. I tried to experiment with this approach in <https://github.com/asterkin/bz-taew> (generic) and <https://github.com/asterkin/bz-taew-py> (Python). Therefore, we are probably going to combine some conceptual pieces from Promise Theory (bottom-up, what language-specific plugin can do) and Meta-system Transition (abstracting out programming language specific and gradually shifting main focus to application spec rather than implementation). In practical terms. it means that it would be better to start with `claude-taew-py` as a language-specific plugin and proceed

with generic `calude-taew` once more than one language-specific plugin exists and we could start seriously talking about polyglot development, flexible front-end/backend functionality allocation, language/runtime as a configuration choice etc. The only question I have is whether a future `calude-taew` plugin will be able to utilize language-specific `claude-taew-py`? While everything could probably be done with MCP I'm looking for direct yet gradual plugin imports from different marketplaces. Will it be possible?

ChatGPT said:

Thought

Short answer: your instinct is sound, and the constraint is this:

Today, “one plugin importing another from the marketplace” is not really a thing. You *can* absolutely design for reuse, but that reuse happens in your code / MCP servers, not via marketplace-level composition.

So: starting with `claude-taew-py` now and evolving to a generic `claude-taew` later is both pragmatic and compatible with your Promise Theory / Meta-system Transition vision — you just need to structure things right.

Let's break it down.

---

## 1. Is it reasonable to start with `claude-taew-py` and later introduce `claude-taew`?

Yes. Given:

- Your near-term reality is Python (`taew-py`),
- Polyglot use (`taew-ts`, `taew-cpp`, etc.) is contingent on future demand,
- You already experimented with `bz-taew` vs `bz-taew-py`,

it's entirely rational to:

1. Start with `claude-taew-py` as a Python-first plugin, and
2. Later introduce `claude-taew` once you have at least one more language-specific plugin.

That matches:

- Promise Theory: do the smallest reliable thing that a single “agent” (here: Python + Claude) can actually deliver.
- Meta-system transition: only generalize once you have more than one concrete system to abstract over.

So conceptually: 👍

---

## 2. Can a future `claude-taew` plugin *reuse* `claude-taew-py`?

Here we need to be very precise.

There are two different layers:

1. Claude / MCP / marketplace level
  - A “plugin” from Claude’s POV is typically an MCP server or tool spec plus metadata.
  - Today, plugins do not natively “import” each other dynamically from the marketplace. There’s no standard “use plugin X as a dependency of plugin Y” mechanism exposed to end users.
2. Your implementation level
  - You control the code, packaging, and MCP servers behind these plugins.
  - Here, you can absolutely do gradual, layered reuse.

So:

A future `claude-taew` plugin will not magically “import” `claude-taew-py` at the marketplace level.

But it *can* reuse the same code and/or MCP tools underneath.

### Concretely, you have at least three viable strategies

#### Strategy A — One MCP server, evolving from “py-only” to “polyglot”

1. Today:
  - Implement an MCP server (or tool set) that only exposes taew-py tools.
  - Publish that as `claude-taew-py` in the marketplace.
2. Tomorrow:
  - Extend the *same* MCP server to also expose TS / other tools later.

- Add a new plugin manifest `claude-taew` that points to the same server, but:
  - exposes additional commands like `/choose-language`,
  - uses the same tool endpoints with extra parameters like `language = "py" | "ts"`.

From Claude's POV:

- `claude-taew-py` and `claude-taew` are different “plugins” / configs,
- But under the hood they talk to the same process / server.

From your POV:

- You're reusing the same code base, just exposing different entry points.

This gives you:

- A clean path: start language-specific, later add a generic facade.
- No duplicated logic.

## Strategy B — Shared core library, multiple MCP servers

1. Create a Python package, say `taew_agent_core`, inside `claude-taew-py`:
  - Holds all shared concepts:
    - `taew-agent` schema
    - prompt patterns
    - validation logic
    - “plans” for scaffolding ports/workflows/adapters
2. The `claude-taew-py` plugin:
  - Depends on `taew_agent_core`
  - Adds Python-specific commands and bindings.
3. Later, a new `claude-taew` plugin:
  - Imports `taew_agent_core`
  - Exposes polyglot commands:
    - e.g. `/plan-architecture`, `/suggest-language`, `/design-ports`
  - Might call out to language-specific MCP servers (py, ts, etc.).

So `claude-taew` is conceptual meta-system; `claude-taew-py` is a concrete agent.

## Strategy C — `claude-taew` orchestrates other language plugins as separate MCP endpoints

If Claude (or MCP clients) allow configuring multiple MCP servers at once (which is the design goal of MCP), then:

- You can have:
  - `claude-taew-py` MCP server,
  - future `claude-taew-ts` MCP server,
- And a thin config/plugin (`claude-taew`) that simply:
  - connects to both,
  - provides high-level commands that internally dispatch to the right server.

From the marketplace POV, there might be three “plugins”:

- `claude-taew-py` — Python-specific tools
- `claude-taew-ts` — TS-specific tools (future)
- `claude-taew` — high-level, polyglot orchestrator that assumes those two are installed/enabled.

Whether Claude’s UX makes that smooth is a product question, but at a protocol level (MCP) it’s fine: multiple tool servers, one client.

---

### 3. About “direct yet gradual plugin imports from different marketplaces”

Given how things stand conceptually:

- There isn’t (yet) a cross-marketplace dependency resolution mechanism like:
  - “Plugin A from Marketplace 1 depends on Plugin B from Marketplace 2; auto-install it.”
- Marketplace-level relationships are typically:
  - “Here’s a plugin; user installs it.”
  - That’s it.

So:

You shouldn’t *plan* on marketplace-level composition being your integration mechanism.

Instead:

**Plan on code-level and MCP-level composition, for example:**

- Code reuse:

- Shared Python packages (like `taew_agent_core`) used by both `claude-taew-py` and future `claude-taew`.
- Server reuse:
  - One MCP server serving multiple logical “plugins” via different configurations/manifests.
- Client orchestration:
  - Use Claude’s ability to connect to multiple MCP servers in a single session.

That gives you the “direct yet gradual imports” you want — just not via the *marketplaces*; via your own architecture.

---

## 4. Mapping this back to Promise Theory & Meta-system Transition

You basically sketched the right path already; let’s map it explicitly:

### Bottom-up (Promise Theory)

- Step 1: A single agent (`claude-taew-py`) that promises:
  - “I can reliably scaffold taew-py projects”
  - “I can reason about ports, workflows, adapters in Python”
- This is concrete, testable, and delivers immediate value to you + early taew adopters.

### Meta-system Transition

- Step 2+: Once a second concrete agent exists (say `claude-taew-ts`):
  - You now have multiple “systems” to abstract over.
  - At this point, the generic `claude-taew` meta-agent:
    - Shifts focus from “Python vs TS details”
    - Towards “What is the application spec? What front/back distribution? What latency/SLA? What runtime best fits?”

So:

- Starting with `claude-taew-py` is not only pragmatic — it is actually *aligned* with the theories you’re referencing.
- 

## 5. Recommendation in one sentence



Yes, start with `claude-taew-py`. Design it so that its agent logic lives in a reusable core (or MCP server) that a future `claude-taew` plugin can wrap and orchestrate, instead of relying on marketplace-level plugin imports.

If you'd like, next step could be something very concrete, like:

- A proposed repo layout for `claude-taew-py` that keeps a clear “future generic core” in mind.
- Or a minimal taew-agent core API (the schema/functions that later `claude-taew` would also use).

You said:

We will start with `claude-taew-py` repository and adjust as we go. Please, suggest a short description to be supplied to github.

ChatGPT said:

Here are several concise GitHub-ready descriptions you can choose from, ranging from neutral to more expressive.

All are optimized for clarity, positioning, and future expansion.

---

## Option A — Clean & Direct (recommended)

Claude plugin for `taew-py`.

Provides AI-native scaffolding, project generation, and workflow automation for Python applications built with the Ports & Adapters (Hexagonal) architecture.

---

## Option B — Slightly Marketing-Oriented

AI-native development tools for `taew-py`.

A Claude plugin that generates ports, adapters, workflows, and complete MEP scaffolding using the taew (Elvish “socket”) architecture.

---

## Option C — Explicitly Python-First

Claude plugin for the taew-py framework.

Adds AI-assisted project initialization, code generation, and CLI workflows for building Python MEPs using Ports & Adapters.

---

## Option D — Emphasizing AI-Native Philosophy

AI-first tooling for taew-py.

This Claude plugin teaches the agent how to build evolvable Python applications using the taew Ports & Adapters architecture.

---

## Option E — Minimalist

Claude plugin providing AI-assisted scaffolding and development tooling for the taew-py Ports & Adapters framework.

---

## If you want my top recommendation: Option A.

It is:

- concise
- neutral
- future-proof
- understandable to contributors and early adopters
- clear that it's a Claude plugin and Python-specific

If you want, I can also generate:

- a longer “About” section,
- README structure,
- tags/topics list for GitHub discoverability.

ChatGPT can make mistakes. Check important info.