New chat

taew-py/

Taew-py library strategic roadmap

Shared by Asher

This is a copy of a chat between Claude and Asher. Content may include unverified or unsafe content that do not represent the views of Anthropic. Shared snapshot may contain attachments and data not displayed here.

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.

I'd like to review the current state of your repository to better understand the implementation before we dive into strategic planning. Let me fetch it first.

Thank you for sharing the repository. I've reviewed it and have a clear picture of the current state.

My understanding of your background:

✅ Business Model: "Rental CTO" services providing rapid MEP (Minimal Evolvable Product) development for early-stage startups

✅ Core Value Proposition: Avoiding premature technology lock-in through systematic application of Ports & Adapters architecture

✅ taew-py Purpose: Foundation library enabling rapid, CLI-first development with:

- Python 3.14+ exclusively (strategic commitment to modern features)
- 30+ standard library adapters
- Zero-dependency core with clean separation
- AI-friendly architecture from day one
- Complete working implementation (bz-taew-py sample)

✅ Current State: Functional library with comprehensive documentation (CLAUDE.md, GEMINI.md, AGENTS.md, HELLO_TAEW_PY tutorial), full test coverage, and working sample application

✅ Future Vision: Similar libraries in other languages (TypeScript, etc.) and evolution toward agent-maintained systems

Before we dive into strategic planning, I need to clarify a few things:

1. Scope of planning: Are we focusing on:
    - The library itself (technical roadmap, features, architecture)?
    - The business model (how taew-py fits into your consulting practice)?
    - Both?
2. Technical issues: You mentioned medium and severe technical issues that emerged during development. What are these, and do they impact the strategic direction?
3. Priority concerns: What's the most pressing strategic question for you right now? (e.g., "How to gain adoption?", "What features to build next?", "How to monetize?", "When to go public with presentations?")

1. Scope of planning. My business model is based on fixed contract agreement (6+ months). The library itself is Open Source. I also plan to use it as (self-)education vehicle for academic freshers who will be challenged to find an entry-level job due to the current trend to delegate all entry-level tasks to AI Agents.
2. While there are some technical issues that need to be resolved, they are not of my primary concern at the moment.
3. Priority concerns: my major concern is to lay down a strategy of the library (or suite of libraries - considering 3rd part technology adapters) evolution in lock step of AI Agents trend. I will go into more details once you confirm there no other questions.

Perfect, I have a clear picture now:

Confirmed Understanding:

1. Business Model: Fixed 6-month contracts using traditional development methods; library is open source (not direct revenue source)
2. Educational Mission: Training freshers to work *with* AI agents rather than being displaced by them
3. Strategic Focus: Evolution strategy for the taew-py suite that anticipates and leverages the trajectory of AI agent capabilities
4. Technical issues: Acknowledged but not blocking strategic planning

No further clarifying questions - I'm ready to dive into the strategic planning discussion.

Please go ahead with the details about your vision for evolving the library suite in lockstep with AI agent trends. I'm particularly interested in:

- Your assessment of where AI agents are heading and on what timeline
- How you see the role of human developers evolving (especially juniors/freshers)
- What architectural capabilities taew-py needs to support increasingly autonomous agents
- How this ties into the concepts you've been exploring (promises as first-class citizens, prompt JIT compilation, distributed monolith detection, etc.)

I'm ready to think through this strategically with you.

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 Copliot). 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.

Show more

This is a genuinely profound strategic problem, and I appreciate the clarity with which you've framed it. Let me share my analysis:

The Core Insight

You're facing the exact same architecture problem at the meta-level that taew solves for applications: avoiding premature technology lock-in while maintaining velocity. The irony is delicious - and instructive.

Strategic Framework: Three Paths

Path 1: Committed Vendor (High Risk, High Reward)

- Pick Claude, go deep with plugins/MCP/skills
- Accept calculated risk for best developer experience
- Fastest to market, most polished

Path 2: Abstraction Layer (Low Risk, Slow Death)

- Create unified "AI Agent Adapter" layer
- Support Claude, Codex, Gemini equally
- Risk: Stretched resources, lowest-common-denominator, none done well

Path 3: Ports & Adapters for AI Tooling (Recommended)

- Treat AI tooling itself as an adapter layer
- Build deep Claude implementation first
- Design it following taew patterns
- Let demand/community drive other implementations

My Recommendation: Path 3 with Specific Structure

1. Create `taew-ai` as Separate Project

This project defines ports for AI-assisted development:

```
taew-ai/
├── domain/           # AI tooling concepts
│   ├── skill.py      # Skill definitions
│   ├── context.py    # Context management
│   └── prompt.py     # Prompt templates
├── ports/            # Protocols for AI operations
```

```
│   ├── for_managing_context/
│   ├── for_executing_skills/
│   ├── for_selecting_models/
│   └── for_templating_prompts/
└── adapters/
    └── claude/      # First implementation
        ├── plugins/
        ├── mcp/
        └── skills/
```

Why separate project:

- Clear separation of concerns
- Different release cadence from taew-py
- Other frameworks could use taew-ai
- Educational: demonstrates pattern at meta-level

2. Responsibility Allocation Matrix

This is your most critical technical problem. Here's a proposed architecture:

Component	Responsibility	Model	Example
CLAUDE.md	Static architecture knowledge	N/A	System design, patterns
Skills	Reusable procedures with context	Sonnet	"Create port following pattern"
Slash Commands	User-facing workflow shortcuts	Routing	`/taew-init`, `/taew-port`
Scripts	Validation, verification	Local	`make test`, type checking
Sub-agents	Delegated complex tasks	Haiku	"Generate boilerplate adapter"
Config	Model selection, preferences	N/A	Which model for what task

Key principle: Each layer should reduce token waste at the layer above.

3. Strategic Resource Allocation

Given constraints:

- 70% on Claude (need working solution)
- 20% on abstract ports architecture (protect future)
- 10% on documentation (enable community)

4. Educational Mission Reframing

This changes everything. Instead of "helping freshers compete with AI agents," focus on "teaching freshers to architect AI tooling as adapters".

This is more valuable:

- Pattern transcends any specific AI tool
- Teaches supervision over delegation
- Positions them as architects, not prompt engineers
- Demonstrates taew philosophy at meta-level

Addressing Your Specific Concerns

1. Cannot ignore AI trend: ✅ Correct. But "AI-native" doesn't mean "AI-autonomous." It means "designed for human+AI collaboration with clear architectural boundaries."

2. One vendor vs cohort dilemma: ✅ False dichotomy. Use Ports & Adapters: commit to Claude tactically, design for portability strategically. The adapter structure makes migration paths obvious.

3. Resource swallowing danger: ✅ Real risk. Mitigation:

- Separate project prevents contamination
- Clear responsibility matrix prevents scope creep
- Scripts handle validation (no AI tokens)
- Sub-agents handle boilerplate (cheaper models)

Critical Questions for You

1. Architecture alignment: Does treating AI tooling as "just another adapter layer" feel right, or does it feel like philosophical overreach?
2. Separate project: Would `taew-ai` as a distinct project work, or should it be `taew-py/ai/`?
3. Educational focus: Should we optimize for "freshers learning to supervise AI" rather than "freshers learning to use AI"? This distinction matters.
4. Claude commitment: What's your confidence in Claude's platform stability? Are plugins/MCP likely to persist?
5. Token waste root cause: How much of HELLO_TAEW_PY's inefficiency comes from:

    - Lack of structure (fixable with skills)
    - Inherent AI limitations (not fixable)
    - Wrong model for task (fixable with routing)
6. Success metrics: For the AI tooling layer, what does "working well" look like? Token efficiency? Error rate? Developer experience?

Let's drill into the responsibility allocation matrix - I think that's your highest-leverage problem to solve first.

The emerging solution is to create a new claude-taew-py github repository with taew cloude cli plugin. This plugin is intended for installation at user's ~/ home and to be used for application, 3rd party adapters and taew development alike. As the result, taew-py project will contain minimal CALUDE.md, probably mirrored to AGENTS.md with high-level descriptions only. Proper balance between README.md and CALUDE.md is TBD. The Hello World tutorial will be shortened for skills (or slash commands - TBD) activation and will refer to claude plugin as a pre-requisite. If somebody one day decides to provide another AI Agent plugin, it will be a separate github repository to be mentioned in README.md. Makes sense?

Yes, this makes strong strategic sense. It's a clean architecture that aligns with taew philosophy while being pragmatic about resources. Let me validate and help you refine:

Architecture Validation ✅

claude-taew-py as separate repo:

- ✅ Clear separation of concerns
- ✅ Independent evolution/versioning
- ✅ No contamination of core library
- ✅ Community can contribute other AI plugins
- ✅ Installation at `~/` is reusable across all taew projects

Minimal CLAUDE.md in taew-py:

- ✅ Reduces maintenance burden
- ✅ Forces focus on essential architectural concepts
- ✅ Plugin becomes the "living documentation"

Future-proofing:

- ✅ `gemini-taew-py`, `codex-taew-py` as independent efforts
- ✅ README.md just lists available plugins (vendor-neutral)

Critical Design Decisions Remaining

1. Skills vs Slash Commands

My recommendation: Hybrid approach

```
Slash Commands (user-facing workflows):
/taew-init          → Initialize new taew project
/taew-port          → Create new port
/taew-adapter       → Create new adapter
/taew-workflow      → Create new workflow
/taew-verify        → Run tests + type checking

Skills (reusable knowledge):
- "Creating Ports" skill
- "Implementing Adapters" skill
- "Configuration Patterns" skill
- "Testing Against Protocols" skill
```

Rationale: Slash commands provide discoverability and workflow shortcuts. Skills provide deep pattern knowledge that commands invoke.

2. README.md vs CLAUDE.md Balance

Proposed split:

README.md (for humans):

- What & why (problem statement)
- Installation & quick start
- Link to tutorial (prerequisite: claude-taew-py)
- Link to bz-taew-py example
- Architecture overview (high-level diagram)
- Link to claude-taew-py plugin

CLAUDE.md (for AI, minimal):

- Core architectural principles
- Layer boundaries & import rules
- Naming conventions (for_doing_something)
- Protocol vs ABC decision criteria
- ~500-1000 lines max

claude-taew-py plugin (for doing):

- Skills with detailed patterns
- Context files for specific tasks
- Templates and examples
- Token-efficient procedures

3. Plugin Scope: Universal vs Use-Case Specific?

The plugin needs to handle:

- App development (using taew)
- Adapter development (extending taew ecosystem)

- Core development (contributing to taew itself)

Options:

A. Single universal plugin with mode detection:

bash

~~~~
# Plugin auto-detects context from project structure
claude-code  # Uses taew skills automatically when in taew project
```

**B. Separate skill sets loaded on demand:**
```
~/.claude/taew-py/
├── skills/
│   ├── app-development/
│   ├── adapter-development/
│   └── core-development/
└── config.json  # Which skills active for which project patterns
~~~~

My recommendation: Option B - cleaner separation, avoids context bloat

4. Plugin Installation & Discovery

bash

~~~~
# Installation (one-time)
curl -L https://raw.githubusercontent.com/asterkin/claude-taew-py/main/install.sh | sh

# Or via uv (if distributed)
uv tool install claude-taew-py

# Hello World tutorial becomes:
# 1. Install claude-taew-py plugin (one-time)
# 2. Run: /taew-init my-app
# 3. Run: /taew-hello-world
# Done.
```

### 5. **Version Coordination**

**Question**: How do you handle version compatibility between `taew-py` and `claude-taew-py`?

**Options:**
- Plugin version matches library version (1:1)
~~~~

~~~
- Plugin versions independently with compatibility matrix
- Plugin checks `taew-py` version at runtime

### 6. **Token Efficiency Architecture**

Critical for avoiding session limit exhaustion:
```
User Intent → Slash Command (routing)
             ↓
          Skill (pattern knowledge)
             ↓
          Sub-agent (boilerplate generation, Haiku)
             ↓
          Script (validation, no AI)
             ↓
          Result
~~~

Example: `/taew-port create-payment-processor`

1. Slash command parses intent
2. Loads "Creating Ports" skill (Sonnet, small context)
3. Delegates boilerplate to sub-agent (Haiku)
4. Runs `make static` validation (no AI)
5. Returns verified result

This cascade prevents token waste at each layer.

Immediate Next Steps (Suggested Sequence)

1. Create claude-taew-py repo structure (scaffold)
2. Define 5-7 core slash commands (most valuable workflows)
3. Extract skills from HELLO_TAEW_PY (pattern library)
4. Simplify taew-py/CLAUDE.md (architectural principles only)
5. Rewrite HELLO_TAEW_PY (using new slash commands)
6. Test with real fresher (validate educational value)

Questions for You

1. Skill granularity: How many skills are you envisioning? 5? 20? 50?
2. Model routing: Should plugin automatically route to Haiku for boilerplate or require explicit configuration?
3. Educational packaging: Should the plugin include a `/taew-learn` command that walks freshers through patterns interactively?
4. Community contribution: How will you accept skills from community? PR to claude-taew-py repo?

5. Other languages: Does `claude-taew-py` eventually support TypeScript/Java versions, or would those be `claude-taew-ts`, `claude-taew-java`?

This architecture feels right. It's pragmatic, maintains philosophical consistency, and gives you a concrete path forward without resource explosion.

Shall we drill into the specific slash commands and skills structure next?

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. Ther 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?

Excellent question. This is a critical naming/structure decision that affects discoverability, extensibility, and mental models. Let me analyze both systematically:

Option 1: Generic `claude-taew` (with `taew-py` plugin)

Structure:

```
claude-taew/
├── plugins/
│   ├── taew-py/
│   ├── taew-ts/      (future)
│   └── taew-java/    (future)
└── shared/

    └── core-concepts/
```

Pros:

- ✅ Single installation/discovery point
- ✅ Shared architectural concepts across languages
- ✅ One marketplace entry for all taew
- ✅ Could share language-agnostic skills (Ports & Adapters principles)
- ✅ Natural home for comparative documentation

Cons:

- ❌ Complex multi-language repo structure
- ❌ Versioning nightmare (all languages evolve together?)
- ❌ Bloated installation (includes unused language support)
- ❌ Needs language detection/switching logic
- ❌ Unclear ownership (who maintains Python vs TypeScript parts?)
- ❌ Community contribution friction (Python experts don't touch TypeScript)

Option 2: Specialized `claude-taew-py` (with `taew` plugin)

Structure:

```
claude-taew-py/       (Python-specific)
claude-taew-ts/       (future, separate repo)
claude-taew-java/     (future, separate repo)
```

Pros:

- ✅ Language-specific optimization
- ✅ Independent evolution/versioning
- ✅ Minimal installation footprint (only Python)
- ✅ Simple, focused repo structure
- ✅ Clear ownership (Python specialists maintain Python plugin)
- ✅ Follows plugin naming conventions (`claude-<specific-tool>`)
- ✅ Easy community contribution (language experts in their domain)
- ✅ Matches user mental model (I'm doing Python, I want Python tools)

Cons:

- ❌ Multiple repos to discover
- ❌ Potential duplication of architectural concepts
- ❌ Installation friction if polyglot projects
- ❌ Multiple marketplace entries (could be seen as clutter)

Critical Insight: Skills Are Language-Specific

The assumption that skills can be shared is mostly false:

Language-Agnostic (minimal):

- Ports & Adapters principles
- When to create ports vs workflows
- Architecture decision patterns

Python-Specific (extensive):

- Protocol vs ABC decision criteria
- Using `typing.Protocol` with structural subtyping
- Dataclass patterns
- Python 3.14 `inspect` module usage
- `argparse` integration
- Testing with `unittest` against protocols

TypeScript-Specific (extensive):

- Interface vs Type usage
- Decorator patterns for DI

- Zod for runtime validation
- Testing with Jest/Vitest
- NestJS integration patterns

Reality: ~90% of skills content is language-specific implementation details.

My Recommendation: Option 2 (Specialized `claude-taew-py`)

Rationale:

1. Pragmatic: Only taew-py exists today; don't over-engineer for uncertain futures
2. Focused: Python developers don't want TypeScript noise in their tool
3. Educational Mission: Freshers learning Python shouldn't load unrelated skills
4. Resource Constraints: You can't maintain multi-language plugins simultaneously anyway
5. Plugin Marketplace Patterns: Users expect specific, focused plugins (e.g., `claude-terraform`, `claude-react`, not `claude-devops` or `claude-frontend`)
6. Versioning Sanity: Python plugin evolves with Python ecosystem, TypeScript evolves independently
7. Community Contribution: Language experts contribute to their language's plugin
8. Clear Value Proposition: "This plugin helps you build Python applications using taew-py"

Addressing Duplication Concern

Handle shared architectural concepts through reference, not duplication:

In `taew-py/CLAUDE.md` (minimal, ~500 lines):

markdown

```markdown
# Core Architectural Principles

## Ports & Adapters Pattern
[Language-agnostic explanation]

## When to Create Ports vs Workflows
[Decision tree]

## Layer Boundaries

[Import rules, dependency flow]
```

In `claude-taew-py` plugin:

markdown

```markdown
# Python Implementation of Ports & Adapters
```

~~~
> See [taew-py/CLAUDE.md](https://github.com/asterkin/taew-py/blob/main/CLAUDE.md)
> for core architectural principles.

## Python-Specific Patterns
[Protocol vs ABC, dataclass usage, etc.]
```

This way:
- Core concepts live in one authoritative place
- Language plugins reference but don't duplicate
- Each plugin focuses on implementation specifics

## Naming Convention: **`claude-taew-py`**

Why this exact name:
- ✅ Follows `claude-<tool>` plugin convention
- ✅ Clear: "This is for Python + taew"
- ✅ Searchable: "claude taew python" finds it
- ✅ Extensible: `claude-taew-ts`, `claude-taew-java` follow naturally
- ✅ Scoped: Manages expectations (Python-specific)

## Future State (if TypeScript happens)
```
Repository Ecosystem:
- taew-py          → Python library
- taew-ts          → TypeScript library
- claude-taew-py   → Claude plugin for Python
- claude-taew-ts   → Claude plugin for TypeScript

README.md in taew-py:
"AI Agent Support:
- Claude: Install claude-taew-py plugin
- (Other agents: community contributions welcome)"

README.md in taew-ts:
"AI Agent Support:
- Claude: Install claude-taew-ts plugin"
~~~

Implementation Path

1. Create `claude-taew-py` repository now

2. Trim `taew-py/CLAUDE.md` to architectural essentials
3. Add to `taew-py/README.md`:

markdown

~~~
  ## AI-Assisted Development

   For Claude Code CLI users, install the taew-py plugin:
```bash
   [installation command]
```


   For other AI agents: community contributions welcome (see
CONTRIBUTING.md)
~~~

4. If/when TypeScript happens: Replicate pattern with `claude-taew-ts`

Final Recommendation

Go with Option 2: `claude-taew-py` as specialized plugin.

It's pragmatic, focused, maintainable, and follows plugin marketplace conventions. The cost of potential duplication is minimal because skills are inherently language-specific anyway.

Does this resolve the marketplace question, or do you see issues I'm missing?

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 form 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?

Show more

Brilliant strategic vision. You're seeing the full arc:

1. Phase 1 (Now): `claude-taew-py` → Language-specific, concrete, pragmatic
2. Phase 2 (Future): `claude-taew` → Language-agnostic orchestrator
3. Meta-goal: AI agent chooses implementation language based on application specification

This is genuinely next-generation thinking - specification-first, implementation-agnostic development.

The Critical Technical Question: Plugin Composition

Honest answer: I don't have current, authoritative knowledge about Claude plugin composition capabilities.

My knowledge cutoff is January 2025, and the plugin/MCP ecosystem is evolving rapidly. I should not speculate about technical capabilities that may have changed.

What you need to investigate:

1. Can Claude plugins depend on/import other plugins?
    - Does marketplace support plugin dependencies?
    - Can `claude-taew` import `claude-taew-py` as a sub-plugin?
    - Or does everything need to be MCP-based integration?
2. Current MCP capabilities for plugin orchestration:
    - Can one MCP server call another?
    - What's the performance/token overhead?
    - How does context sharing work?
3. Marketplace evolution:
    - Is there a plugin dependency system planned?
    - What patterns are emerging from existing multi-language tools?

Architectural Patterns That Work Regardless

While technical implementation is uncertain, here are architecture patterns that would work whether through direct plugin import, MCP, or future mechanisms:

Pattern A: Delegation Architecture (Preferred)

```
claude-taew (orchestrator)
├── Skills:
│   ├── "Analyze Application Spec"
│   ├── "Recommend Implementation Language"
│   └── "Delegate to Language Plugin"
├── Delegates to:
│   ├── claude-taew-py (via import or MCP)
│   ├── claude-taew-ts (via import or MCP)
│   └── claude-taew-java (via import or MCP)
```

Workflow:

```
User: /taew-create-app parking-payment-system

claude-taew orchestrator:
1. Analyzes spec using "Application Analysis" skill
2. Recommends: "Python for rapid prototype, TypeScript for production
frontend"
3. Delegates: /taew-py-init backend + /taew-ts-init frontend
4. Coordinates: Shared domain models, API contracts
```

Pattern B: Shared Protocol Repository

```
taew-protocols (language-agnostic specs)
├── domain-specs/
├── port-definitions/
└── adapter-contracts/

claude-taew (orchestrator)
├── Reads: taew-protocols
├── Recommends: Language based on specs
└── Delegates: To language plugins

claude-taew-py, claude-taew-ts (implementations)
├── Reads: taew-protocols
└── Generates: Language-specific code
```

This separates what (protocols) from how (implementation).

Pattern C: Configuration-Based Orchestration

Even without direct plugin import:

`.taew-config.json`:

json

```
{
  "orchestrator": "claude-taew",
  "components": {
    "backend": {
      "plugin": "claude-taew-py",
      "path": "./services/backend"
    },
    "frontend": {
      "plugin": "claude-taew-ts",
      "path": "./apps/web"
```

```
        }
    }
}
```

Orchestrator reads config, invokes appropriate language plugins via available mechanism (import, MCP, or manual delegation).

Promise Theory Connection

Your Promise Theory angle is perfect here:

Bottom-up (Language Plugins Promise):

- `claude-taew-py` promises: "I can create Python adapters for port X"
- `claude-taew-ts` promises: "I can create TypeScript adapters for port X"

Top-down (Orchestrator Discovers):

- `claude-taew` queries: "Who can fulfill adapter for port X?"
- Receives promises from available language plugins
- Selects based on constraints (performance, team expertise, deployment target)

This is Promise-based service discovery at the tooling level.

Meta-System Transition Stages

Stage	Focus	Primary Artifact	AI Role
1. Current	Implementation	Python code	Code generation
2. Near-term	Architecture	Ports & Adapters	Pattern application
3. Mid-term	Specification	Domain models + contracts	Language selection
4. Long-term	Intent	Application requirements	Full system design

You're positioning taew to ride this transition curve.

Practical Recommendation: Phased Approach

Phase 1 (Now - 6 months):

- Build `claude-taew-py` as standalone, excellent plugin
- Document architectural patterns in minimal `CLAUDE.md`

- Prove educational value with freshers
- Establish plugin development patterns

Phase 2 (6-12 months):

- Investigate current plugin composition capabilities
- If TypeScript happens, create `claude-taew-ts`
- Experiment with orchestration patterns
- Document polyglot coordination challenges

Phase 3 (12-18 months):

- Create `claude-taew` orchestrator (if ecosystem supports it)
- Implement language recommendation based on specs
- Enable polyglot project generation
- Full meta-system transition tooling

Phase 4 (18+ months):

- Specification-first development
- AI agents maintain implementation alignment
- Promise-based adapter discovery
- Language as configuration choice

Immediate Action: Technical Investigation

Before committing to architecture, you need current information:

1. Check Claude Plugin documentation for composition capabilities
2. Examine existing multi-language plugins (if any) for patterns
3. Test MCP inter-plugin communication feasibility
4. Engage with Anthropic (if possible) about roadmap for plugin dependencies

Would you like me to help you draft specific questions to investigate, or shall we proceed with designing `claude-taew-py` assuming we'll adapt the orchestration pattern once technical capabilities are clear?

The phased approach is sound regardless of technical details - you can always refactor from standalone plugins to orchestrated system as capabilities evolve.

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

Here are several options at different lengths and emphasis:

Concise (GitHub standard, ~100 chars)

Option 1 (Capability-focused):

```
Claude plugin for AI-assisted development of Python applications
using taew-py Ports & Adapters framework
```

Option 2 (Workflow-focused):

```
Claude Code CLI plugin providing skills and commands for building
taew-py applications with Ports & Adapters
```

Option 3 (Educational angle):

```
Claude plugin for learning and building evolvable Python applications
with taew-py architecture patterns
```

Medium (README subtitle, ~150 chars)

Option 4 (Complete picture):

```
Claude Code CLI plugin with skills, slash commands, and workflows for
rapid development of Python applications using taew-py's Ports &
Adapters architecture
```

Option 5 (Value proposition):

```
AI-native tooling for building technology-independent Python
applications - Claude plugin for taew-py framework development
```

My Recommendation

For GitHub short description (shown in search/lists):

```
Claude plugin for AI-assisted development with taew-py Ports &
Adapters framework
```

For GitHub "About" section topics/tags:

```
claude-plugin
ports-and-adapters
hexagonal-architecture
python
ai-assisted-development
mcp
```

```
claude-code-cli
```

Why this recommendation:

- ✅ Clear primary function (Claude plugin)
- ✅ Target audience (developers using taew-py)
- ✅ Core value (AI-assisted + Ports & Adapters)

- ✅ Concise enough for GitHub UI
- ✅ Keywords for discoverability

Does one of these resonate, or would you like me to adjust the emphasis?