Agent Skills

Prototype: Table Structure Recognition

Agile SoDA

DOCUMENT AI 알고리즘팀

2026.01.19

Goal

Build Modular Agent Skills for a TSR Prototype

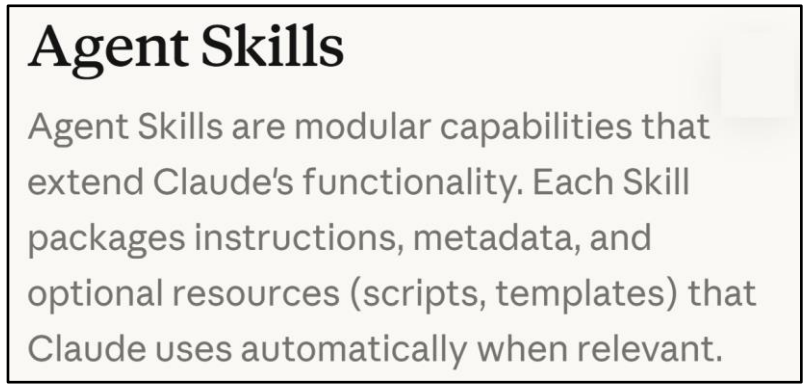
- Agent → Autonomous, minimizing human intervention
- Skills → Task specific modular
- TSR (Task)→ Table Structure Recognition
- Prototype → Simple, end-to-end functional, modular, and extensible

Content

- Introduction
- Proposed System
- Implementation
- Result
- Conclusion

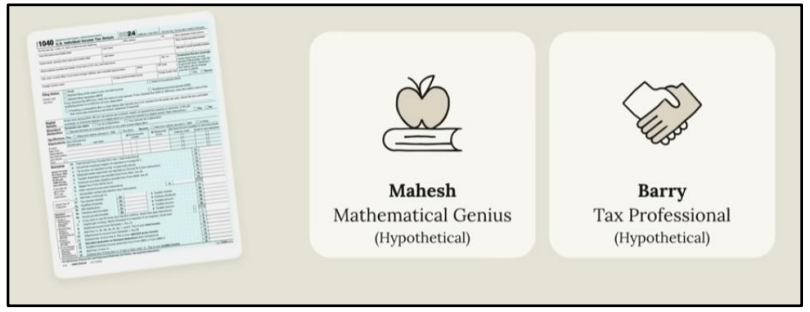
Introduction

- **What is Agent Skills?**
  - Task specific modular
  - Activates only when needed
  - E.g., TSR, classification, and summarization
  - Dev simplified: "Folder"



Claude Agent Skill

- **Why we need Agent Skills?**
  - Domain/Task expertise
  - Modular & Extensible



Claude Agent Skill

Introduction

- Available Agent Skills
  - Claude - Agent Skills
  - Google Antigravity – Agent Skills
  - Cursor – Agent Skills
  - OpenAI – CodeX – ExecPlan
  - Google Gemini - Gems

Skill Anatomy

```
skill-name/
├── SKILL.md (required)
│   ├── YAML frontmatter metadata (required) (Triggering & routing - When to use)
│   │   ├── name: (required)
│   │   └── description: (required)
│   └── MD instructions (required) (Instructional control - workflow, heuristic)
└── Bundled Resources (optional)
    ├── scripts/    - Executable code (Executable implementation - Python/Bash)
    ├── references/ - Doc loaded into context as needed (Contextual knowledge)
    └── assets/     - Files used in output (Output artifacts)
```

Claude Agent Skill

Each skill should be a folder containing a `SKILL.md` file:

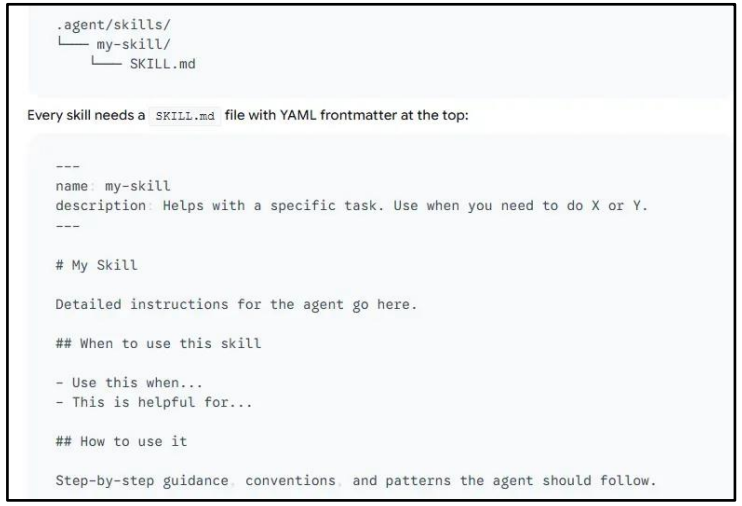
```
.cursor/
└── skills/
    └── my-skill/
        └── SKILL.md
```

SKILL.md file format

Each skill is defined in a `SKILL.md` file with YAML frontmatter:

```
---
name: my-skill
description: Short description of what this skill does and when to use it.
---

# My Skill

Detailed instructions for the agent.

## When to Use

- Use this skill when...
- This skill is helpful for...

## Instructions

- Step-by-step guidance for the agent
- Domain-specific conventions
- Best practices and patterns
```

Cursor – Agent Skills

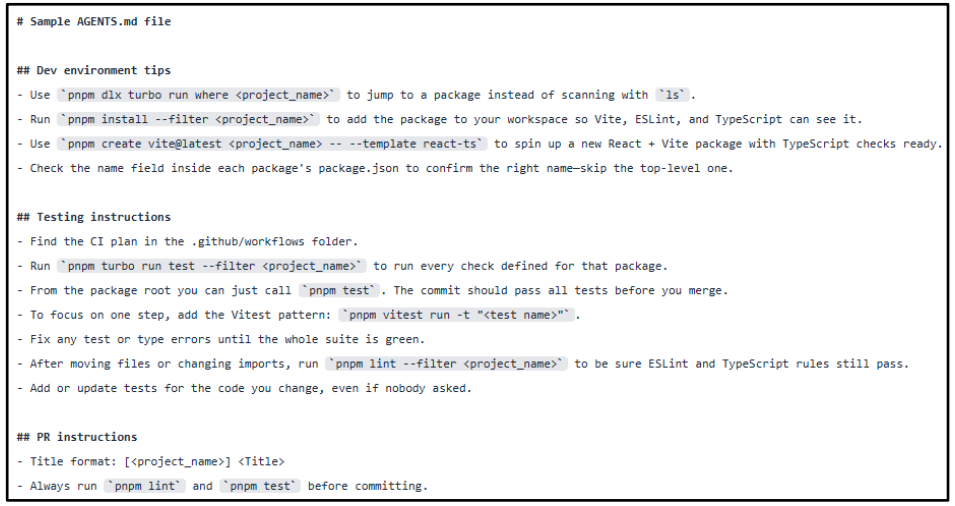


```
.agent/skills/
└── my-skill/
    └── SKILL.md
```

Every skill needs a `SKILL.md` file with YAML frontmatter at the top:

```
---
name: my-skill
description: Helps with a specific task. Use when you need to do X or Y.
---

# My Skill

Detailed instructions for the agent go here.

## When to use this skill

- Use this when...
- This is helpful for...

## How to use it

Step-by-step guidance, conventions, and patterns the agent should follow.
```

Google Antigravity – Agent Skills

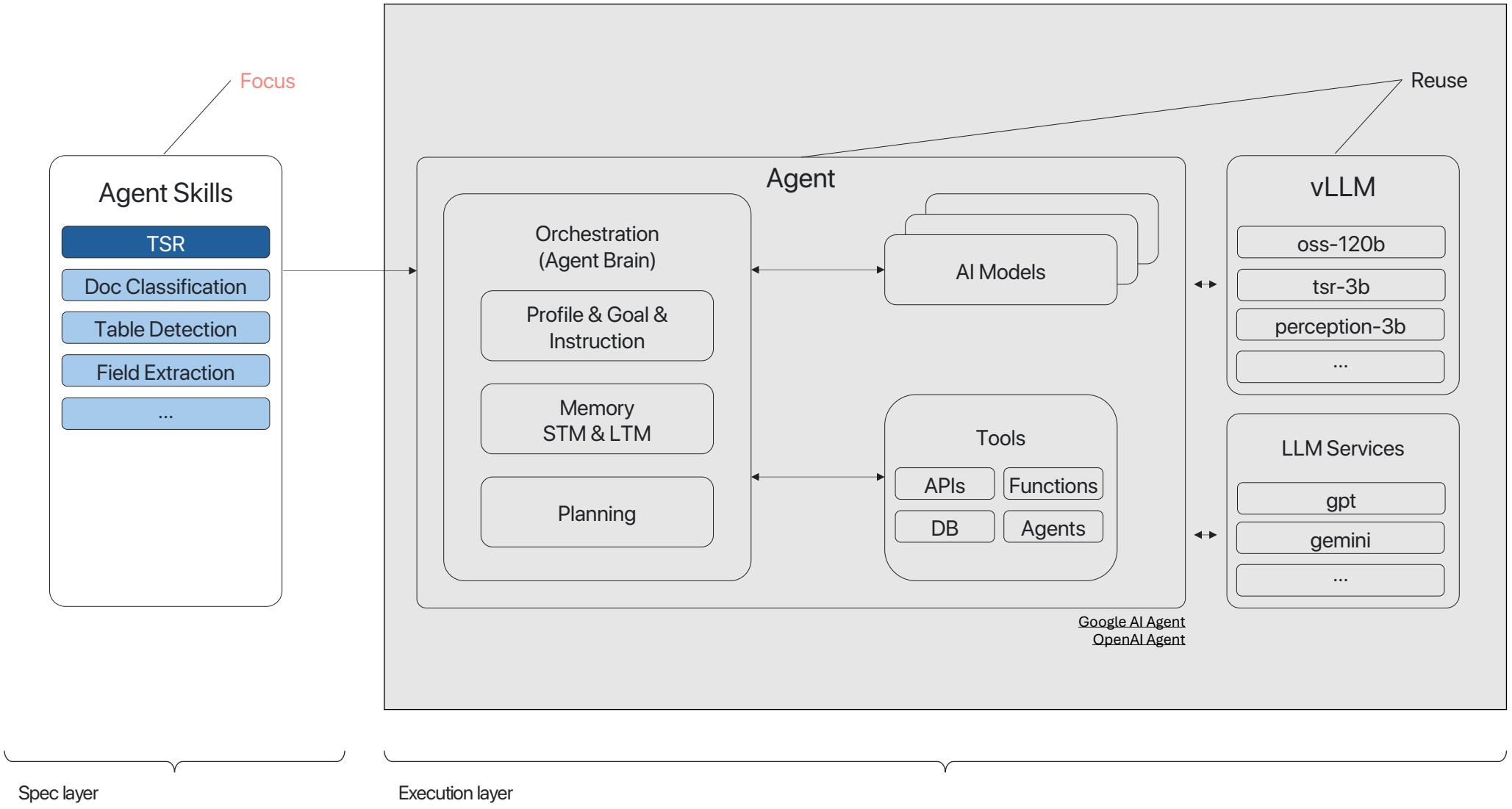


```
# Sample AGENTS.md file

## Dev environment tips
- Use `pnpm dlx turbo run where <project_name>` to jump to a package instead of scanning with `ls`.
- Run `pnpm install --filter <project_name>` to add the package to your workspace so Vite, ESLint, and TypeScript can see it.
- Use `pnpm create vite@latest <project_name> -- --template react-ts` to spin up a new React + Vite package with TypeScript checks ready.
- Check the name field inside each package's package.json to confirm the right name—skip the top-level one.

## Testing instructions
- Find the CI plan in the .github/workflows folder.
- Run `pnpm turbo run test --filter <project_name>` to run every check defined for that package.
- From the package root you can just call `pnpm test`. The commit should pass all tests before you merge.
- To focus on one step, add the Vitest pattern: `pnpm vitest run -t "<test name>"`.
- Fix any test or type errors until the whole suite is green.
- After moving files or changing imports, run `pnpm lint --filter <project_name>` to be sure ESLint and TypeScript rules still pass.
- Add or update tests for the code you change, even if nobody asked.

## PR instructions
- Title format: [<project_name>] <Title>
- Always run `pnpm lint` and `pnpm test` before committing.
```

OpenAI – CodeX – ExecPlan : AGENTS.md | PLAN.md

Proposed System

- **Agent Skills**
  - TSR
  - ...



Implementation

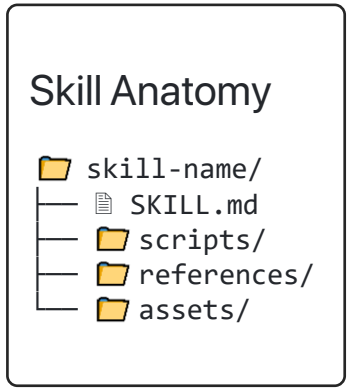
Re-use:

- vLLM
  - (OpenAI) oss-120b: LLM, Agent Brain
  - (AgileSoDA) tsr-3b: VLM, TSR as tool (note: Finetuned VLM TSR lost Reasoning capability)
  - (Nanonets) perception-3b: VLM, perception/ocr as tool

- Agent
  - (OpenAI) Agent (For quick prototyping)
  - (AgileSoDA) Agentic Framework (Daniel, from the scratch)

Implementation

- Agent Skills
  - (Claude) Agent Skill
    - Reuse: Concepts and Skill Anatomy
    - Implementation is private



Claude Agent Skill

Progressive Disclosure Design Principle

How Skills work

Skills are **model-invoked**: Claude decides which Skills to use based on your request. You don't need to explicitly call a Skill. Claude automatically applies relevant Skills when your request matches their description.

When you send a request, Claude follows these steps to find and use relevant Skills:

1 **Discovery**

At startup, Claude loads only the name and description of each available Skill. This keeps startup fast while giving Claude enough context to know when each Skill might be relevant.

2 **Activation**

When your request matches a Skill's description, Claude asks to use the Skill. You'll see a confirmation prompt before the full `SKILL.md` is loaded into context. Since Claude reads these descriptions to find relevant Skills, **write descriptions** that include keywords users would naturally say.

3 **Execution**

Claude follows the Skill's instructions, loading referenced files or running bundled scripts as needed.

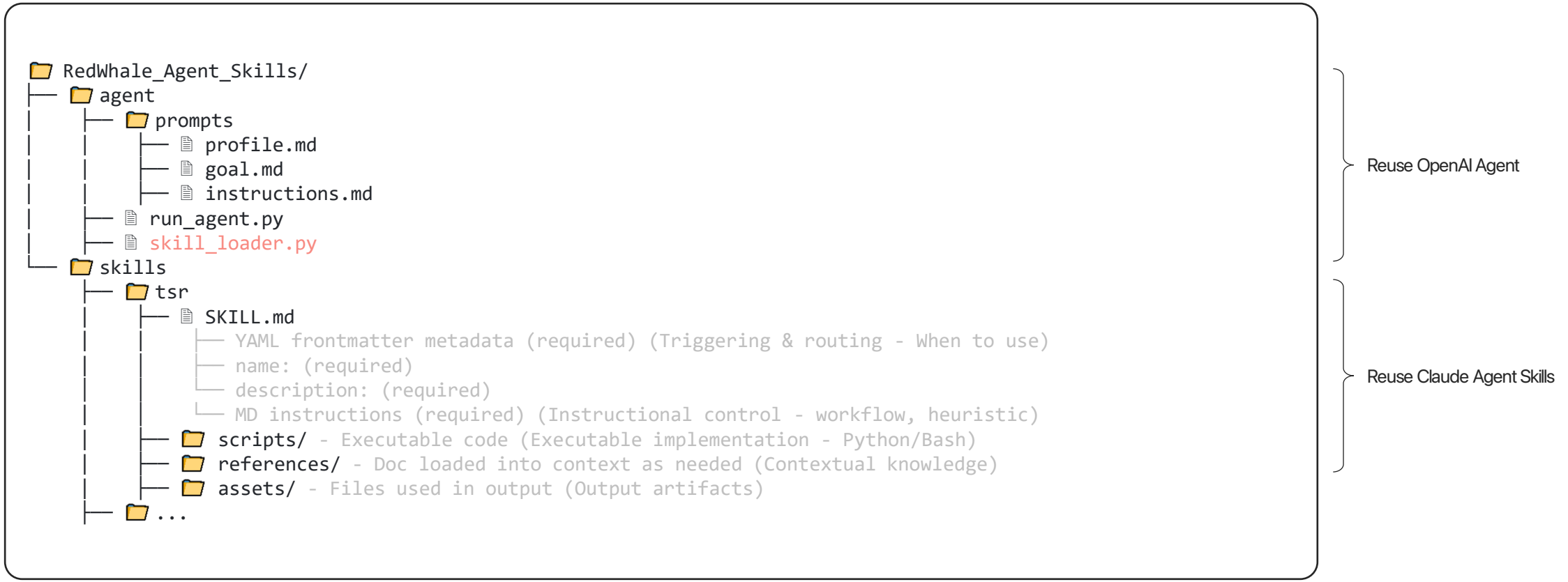
1. Which skill is needed

2. Load the required skill

3. Execute

Implementation

- **Project Structure**



Implementation

- **TSR Skill**

SKILL.md

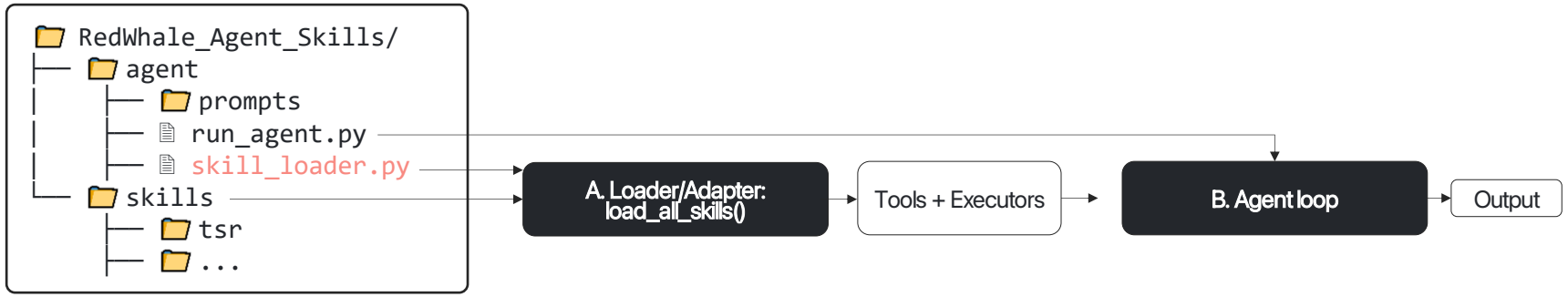
```
---
name: tsr
description: Extract structured tables from table images and return them as HTML and/or OTSL using TSR or
Perception VLM backends.
---

# TSR Skill

## Workflow

### 1) Extract table from an image
- Call: `tsr_extract(image_path, backend, output_format)`
- Backends:
  - **tsr** – dedicated TSR model (best for structure accuracy)
  - **perception** – general VLM (may return HTML directly)

### 2) Detect output format
- If result contains OTSL tags → treat as **OTSL**
- Otherwise → treat as **HTML**

### 3) Convert when needed
- OTSL → HTML using `otsl_to_html(otsl_text)`

### 4) Save result (optional)
- Save HTML file with `save_html(html, path)`

## Supported Outputs
- `html` – final structured table
- `otsl` – intermediate structured format
- `auto` – return both (recommended)

## Output Contract
- **tsr_extract** → raw text + detected format + OTSL/HTML
- **otsl_to_html** → HTML
- **save_html** → file path

## Prompting Rules
- Deterministic mode (`temperature=0`)
- High `max_tokens` for large tables
- Use TSR-focused prompts

## Debugging & Recovery
- Prefer **backend="tsr"** if quality is poor
- Increase max tokens if truncated
- Re-convert from OTSL when HTML is malformed
- Ensure input image is a proper table crop

## Notes
- Use tool calls for reliability
- Keep detailed prompts in reference files
```

scripts/tsr_tools.py

```
# Exposed TSR Tools – Interfaces Only

def tsr_extract(
    image_path: str,
    backend: str = "tsr",          # enum: ["tsr", "perception"]
    output_format: str = "auto",   # enum: ["auto", "otsl", "html"]
    max_tokens: int = 8192,
    prompt_text: str = "Extract table from this image."
) -> dict:
    """
    Extract table structure from a table image via VLM backend.

    Returns (JSON):
    {
      "raw_text": str,                # raw model output
      "detected_format": "otsl"|"html",
      "otsl": str,                    # present if detected_format=otsl (or output_format allows)
      "html": str,                    # present if detected_format=html OR converted from otsl (when
possible)
      "backend_used": "tsr"|"perception",
      "model": str,
      "base_url": str,
      # optional:
      "error": str
    }
    """
    pass


def otsl_to_html(otsl_text: str) -> dict:
    """
    Convert OTSL -> HTML.

    Returns (JSON):
    {
      "html": str,
      # optional:
      "error": str
    }
    """
    pass


def save_html(html: str, out_path: str) -> dict:
    """
    Save HTML string to file.

    Returns (JSON):
    {
      "out_path": str,
      # optional:
      "error": str
    }
    """
    pass
```

Implementation

- ## How Does It Work? (E2E Flow)



Input: path to skills/
Output:

- tools: a list of tool schemas to be passed into client.responses.create(...)
- runtime["executors"]: a dictionary {tool_name: python_callable} used to actually execute the tools

Mechanism:

- Iterate through each folder inside skills/
- Skip folders that are not valid skills (hidden folders, or those without SKILL.md)
- Parse SKILL.md → extract name and description
- Enter the scripts/ directory and import each .py file
- In each script, call the convention function get_registered_tools() to obtain a list of tools:

  {"name": ..., "description": ..., "parameters": ..., "fn": ...}

- Convert them into Responses API schema format:

  {"type": "function", "name": ..., "description": ..., "parameters": ..., "strict": true}

- Store fn into executors for local execution

Result:

- The OpenAI model sees "tools" as a list of functions with schemas
- Your runtime knows which Python function corresponds to each tool

Input: user text
Output: final text response

Loop mechanism (per iteration):

1. Call the model:

   resp = client.responses.create(model=..., tools=tools, input=input_list)

2. Append resp.output to input_list

   - Purpose: to maintain "trace items" (especially for reasoning models)
   - Without this, the model may lose context in the next iteration

3. Filter items with type == "function_call" in resp.output

4. If there is no tool call → end the loop and print resp.output_text

5. For each function_call:

   - Extract name, arguments, call_id
   - Execute the actual Python function:
     executors[name](**args)

   - Append a function_call_output item to return the result to the model:

     {"type": "function_call_output", "call_id": ..., "output": "{...json...}"}

6. Return to step (1) with the updated input_list containing tool outputs

7. The model reads the tool outputs and synthesizes them into the final answer