Agent Skills

Skill 1: Table Structure Recognition (TSR)

Agile SoDA

DOCUMENT AI 알고리즘팀

2026.01.21

Goal

Prototype Agent Skills - TSR

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

Content

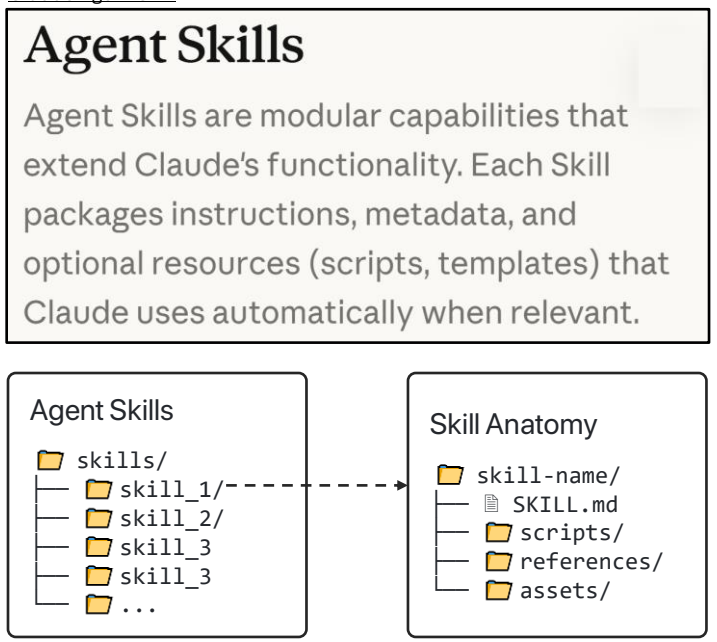
- Introduction
- Proposed System
- Implementation
- Demo
- Test & Results
- Conclusion

Introduction

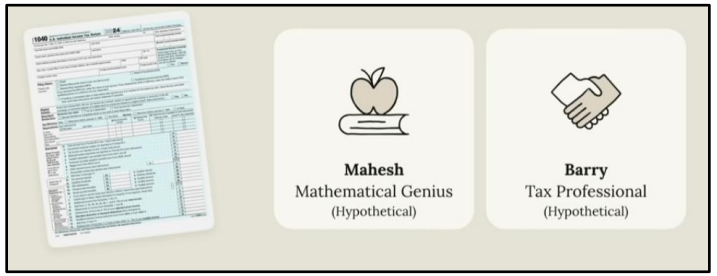
- What is Agent Skills?
  - Task specific modular
  - Agent activates only when needed

  - Examples: TSR, table detection, doc classification,
  - Dev simplified: “Folder”

Claude Agent Skill



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



Introduction

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

Claude Agent Skill

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)
```

Cursor – Agent Skills

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

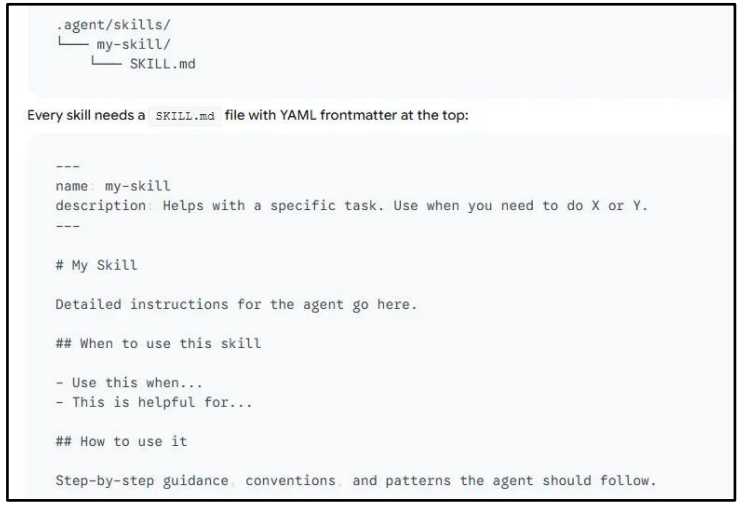
```
.cursor/
└── skills/
    └── my-skill/
        └── SKILL.md
```

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
```

Google Antigravity – Agent Skills

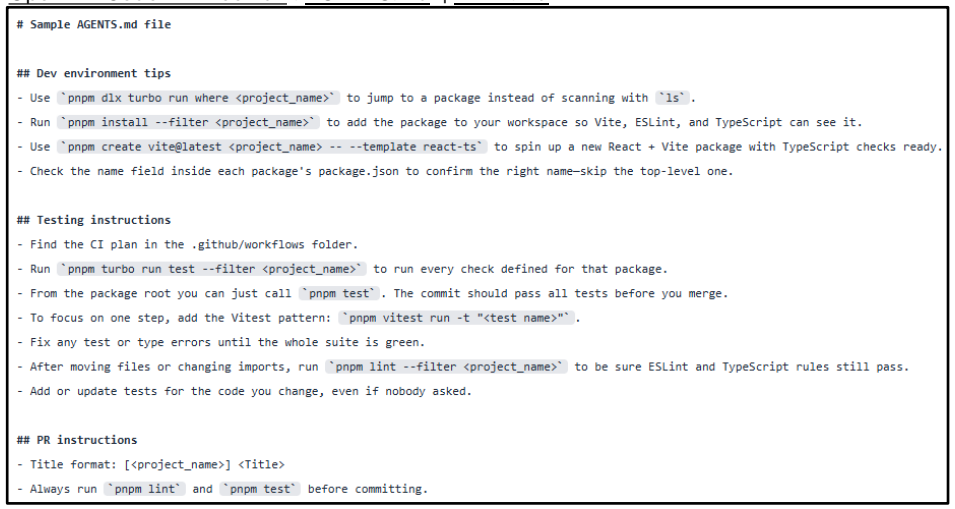


```
.agent/skills/
└── my-skill/
    └── SKILL.md
```

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.
```

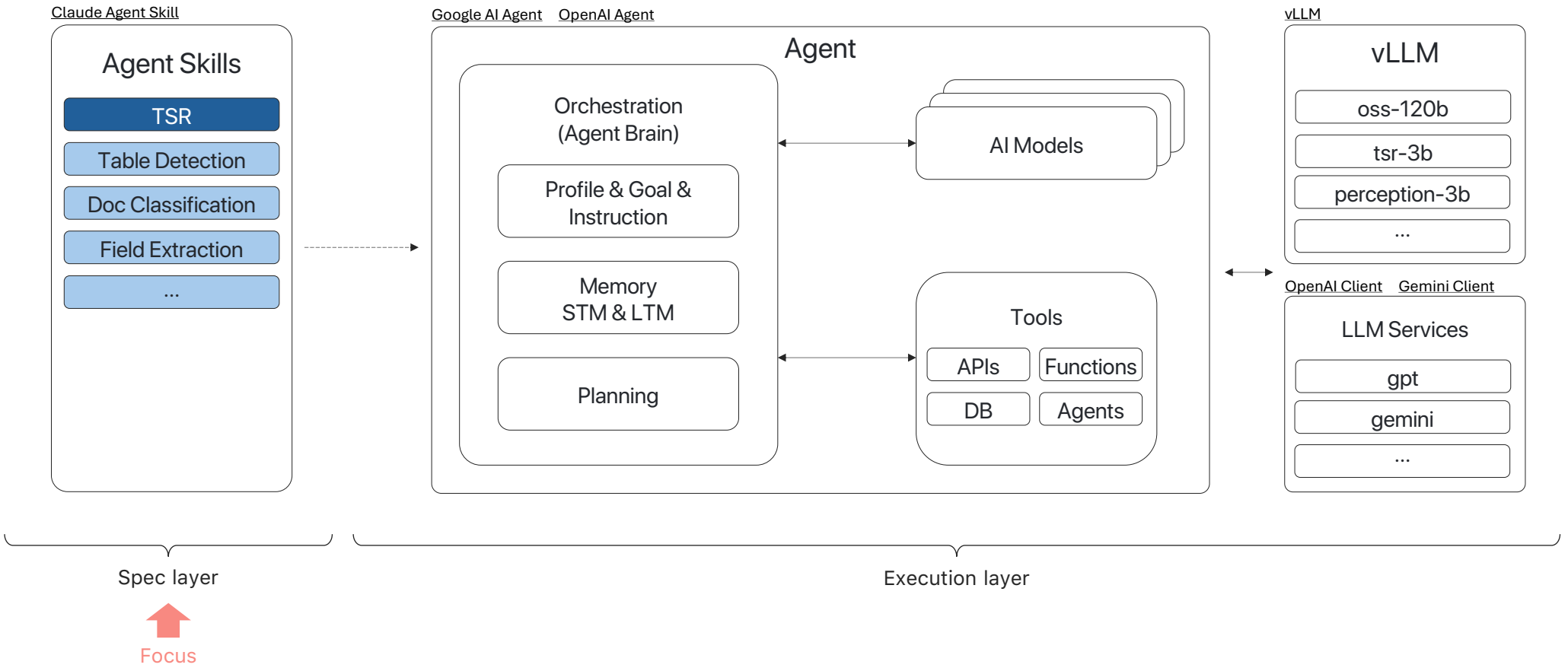
OpenAI – CodeX – ExecPlan : AGENTS.md | PLAN.md



```
# 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.
```

Proposed System

- Agent Skills
  - Skill 1: TSR
  - ...



Implementation

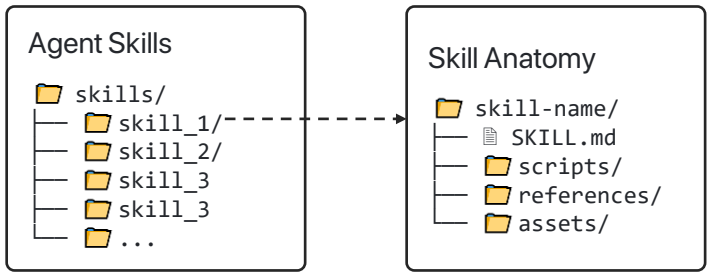
- vLLM
  - (OpenAI) oss-120b: LLM, Agent Brain
  - (AgileSoDA) tsr-3b: VLM, TSR as tool (Finetuned VLM TSR lost Reasoning capability)
  - (Nanonets) perception-3b: VLM, perception/ocr as tool

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

Implementation

- Agent Skills
  - (Claude) Agent Skill
    - Reuse: Skill Anatomy & Concepts
    - Private Implementation

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**

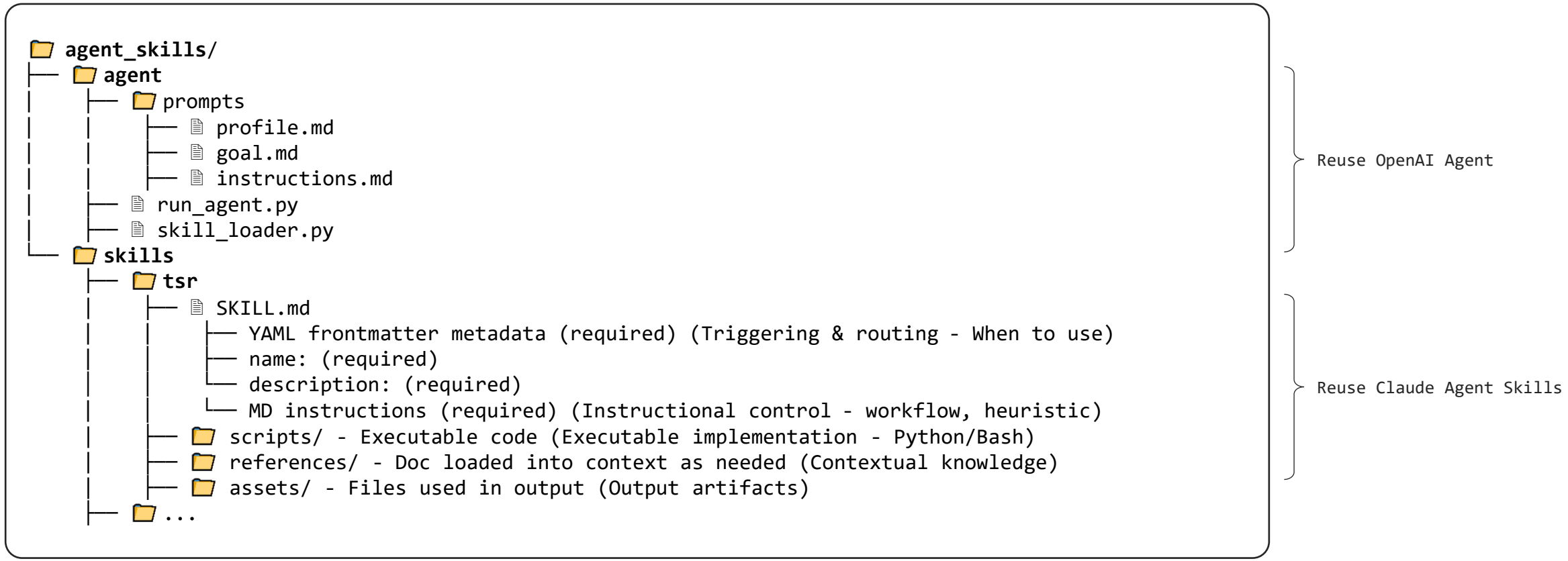
Claude follows the Skill's instructions, loading referenced files or running bundled scripts as needed.

1. Determine which skill is needed
2. Load skill
3. Execute skill

Implementation

- Project Structure

Agent Skills (github project)



Implementation

- **TSR Skill**

SKILL.md

YAML frontmatter

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

Markdown

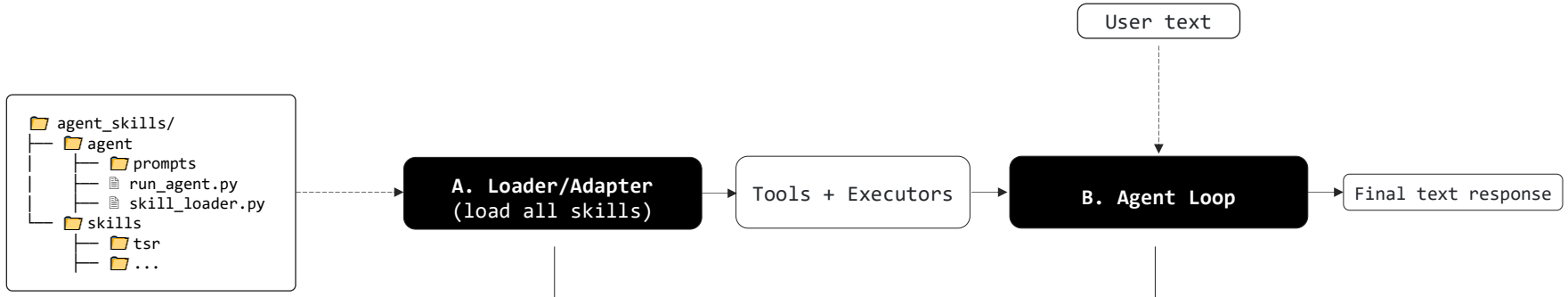
```markdown
# TSR Skill

## 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

```python
# 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: Overall flow (end-to-end flow)



A. Loader/Adapter: load_all_skills() → "turn Skill Folder into Tools + Executors"

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
- The runtime knows which Python function corresponds to each tool

B. Agent loop: run_agent.py → "model selects tool → runtime executes → returns result"

Input: user text

Output: final text response

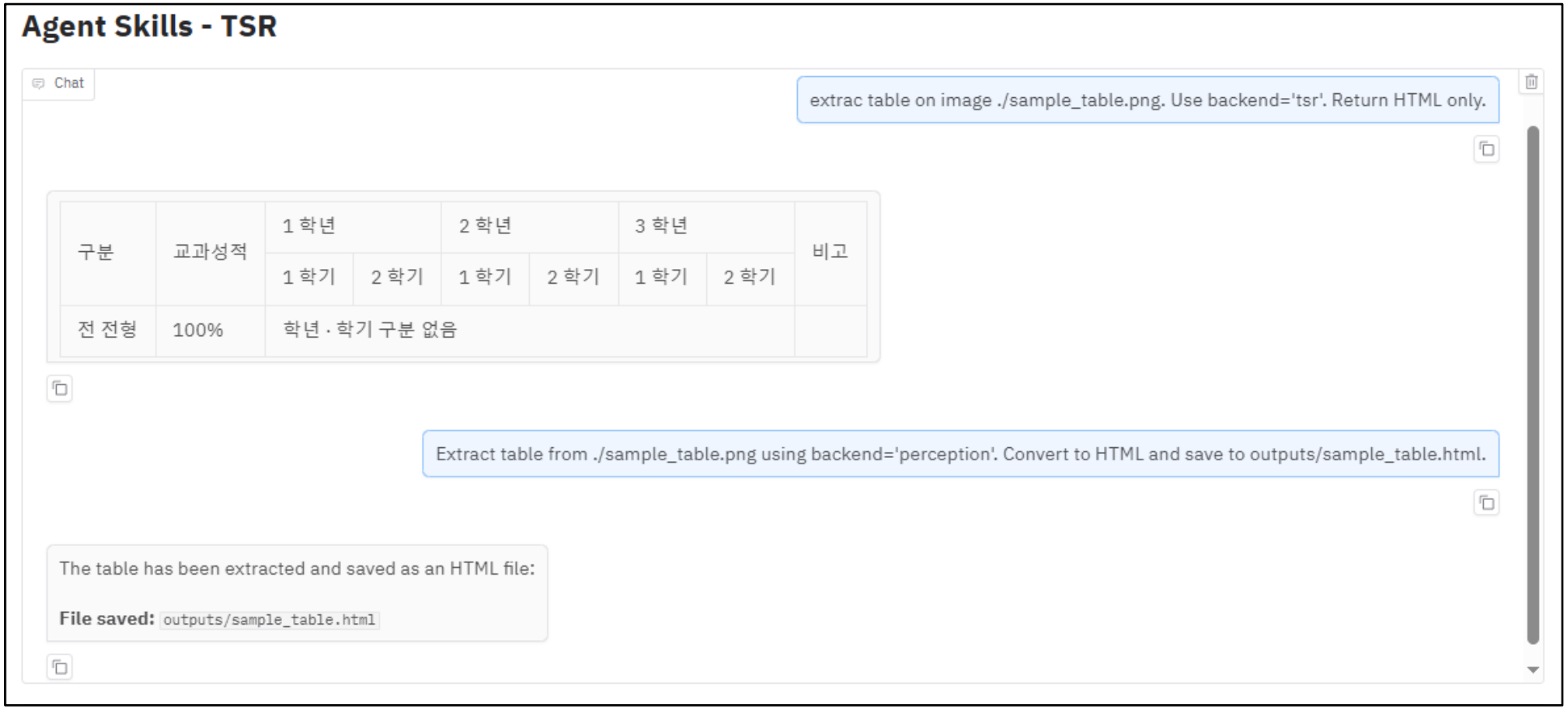
Loop mechanism (per iteration):

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

Demo

- Github Project: Agent Skills
- Gradio Demo: Agent Skills - TSR

구분	교과성적	1학년		2학년		3학년		비고
		1학기	2학기	1학기	2학기	1학기	2학기	
전 전형	100%	학년·학기 구분 없음						



Test & Results

- Settings

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

Skills	• TSR
	  • Exposed Tools:
	  • tsr_extract
	  • otsl_to_html
	  • save_html

Agent	• OpenAI Agent

Dataset	• Trivia
	  • notnt (460 files)
	  • tnt (65 files)

Prompt	• Extract table from $img_path using backend='$BACKEND'. Convert to HTML and save to $output_html.

- Results

Dataset	tsr-3b		perception-3b	
	TEDS (content)	TEDS-Struct	TEDS (content)	TEDS-Struct
BASE	???	???	NA	NA
notnt	0.9377	0.9646	0.6291	0.6864
tnt	0.9599	0.9710	0.4441	0.5038

Conclusion (1/2)

- Agent Skills provide a practical and scalable agent architecture.

  → The goal is not a single TSR pipeline, but a general, extensible framework

- TSR demonstrates the approach end-to-end

  → The Agent automatically selects Skills

  → Skills combine explicit workflows + executable tools

  → Autonomous, deterministic, and recoverable execution

- Clear separation of Spec and Execution layers

  → Spec (Skills): domain/task expertise

  → Execution (Agent): tool calling and agent loop

  → Enables reuse of agent infrastructure

Conclusion (2/2)

- Progressive Disclosure improves efficiency

  → Metadata → Instructions → References (on demand)

  → Lower cost, less noise, fewer hallucinations (in theorical)

- Skills are folder-based and highly extensible

  → Easy to add new capabilities without changing the Agent core

  → Designed for multi-skill, real-world Document AI systems

- Key takeaway

  → Agent Skills turn agents into modular, domain-expert systems.

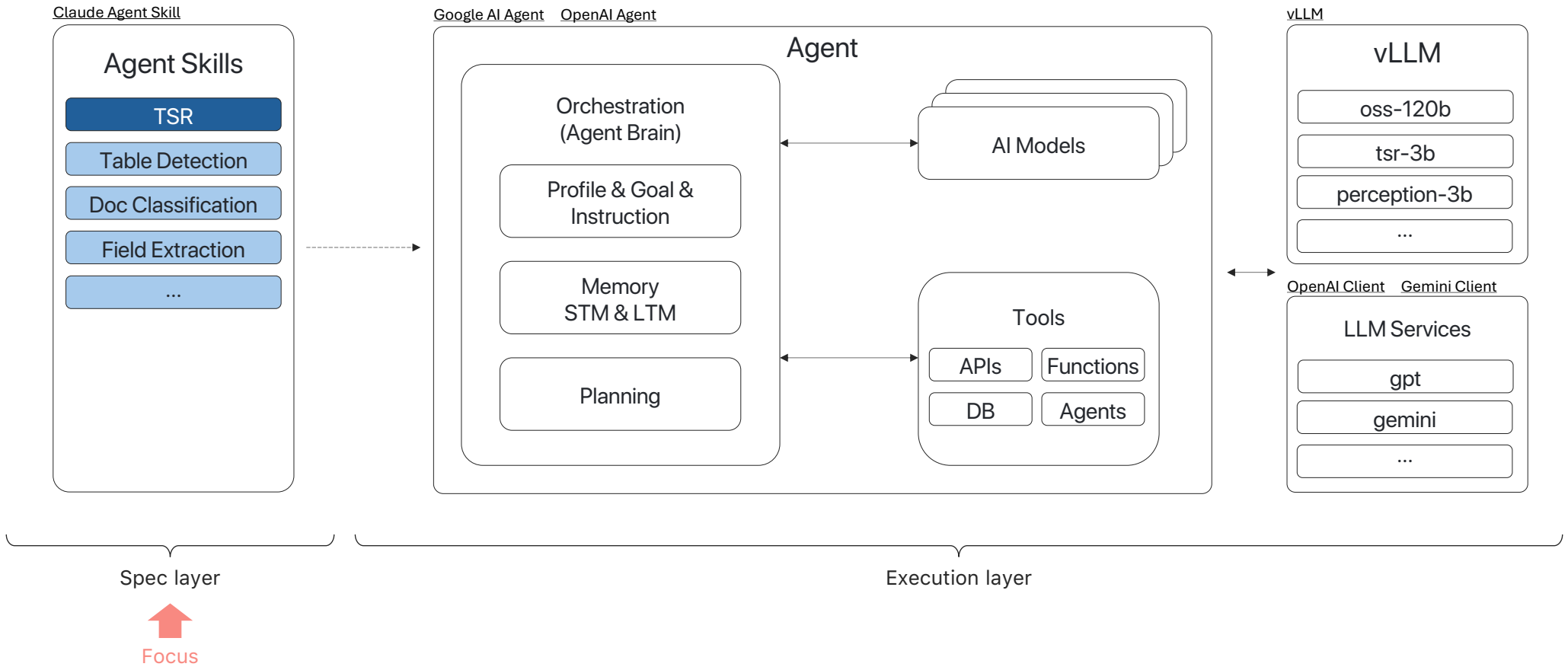
Future Work

- A. Expand Skill Library across Document AI tasks
  → Add skills for Table Detection (Subedi), Doc Classification (Jay), Field Extraction (Khan), Validation (James)
  → Intent: Validate scalability by composing multiple skills in a single agent

- B. Advanced TSR Skill
  → Add more tools and improve skill
  → Intent: Detection → TSR → Validation → Post-processing

- C. Towards Automatic Skill Creation
  → Explore semi-automatic generation of new Skills from task descriptions and examples
  → Intent: Agent creates Skill by Itself

- D. Optimization
  → Direction 1: Skill Quality Evaluation
  → Direction 2: Memory-Enhanced Skills
  → Direction 3: Skill Optimization (SERO)
  → Intent: self-evolving, self-improvement


Claude Agent Skill
Agent Skills
TSR
Table Detection
Doc Classification
Field Extraction
...
Google AI Agent
OpenAI Agent
Agent
Orchestration
(Agent Brain)
Profile & Goal &
Instruction
Memory
STM & LTM
Planning
AI Models
Tools
APIs
Functions
DB
Agents
vLLM
vLLM
oss-120b
tsr-3b
perception-3b
...
OpenAI Client
Gemini Client
LLM Services
gpt
gemini
...
Spec layer
Execution layer
Focus

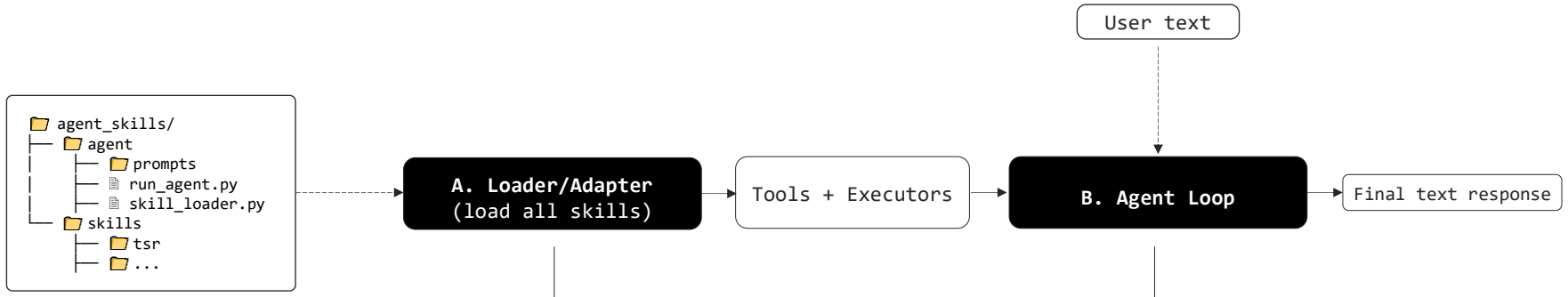
```
RedWhale_Agent_Skills/
├── agent
│   ├── prompts
│   │   ├── profile.md
│   │   ├── goal.md
│   │   └── instructions.md
│   ├── run_agent.py
│   └── skill_loader.py
└── skills
    ├── tsr
    │   ├── SKILL.md
    │   │   ├── YAML frontmatter metadata (required) (Triggering & routing - When to use)
    │   │   ├── name: (required)
    │   │   └── description: (required)
    │   │   └── MD instructions (required) (Instructional control - workflow, heuristic)
    │   ├── scripts/ - Executable code (Executable implementation - Python/Bash)
    │   ├── references/ - Doc loaded into context as needed (Contextual knowledge)
    │   └── assets/ - Files used in output (Output artifacts)
    └── ...
```

Agent Skills

```
skills/
├── skill_1/
├── skill_2/
├── skill_3/
├── skill_4/
└── ...
```

Skill Anatomy

```
skill-name/
├── SKILL.md
├── scripts/
├── references/
└── assets/
```



A. Loader/Adapter: load_all_skills() → “turn Skill Folder into Tools + Executors”

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
- The runtime knows which Python function corresponds to each tool

B. Agent loop: run_agent.py → “model selects tool → runtime executes → returns result”

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