    def \_parse\_and\_validate\_tool\_calls(

        self, tool\_or\_tools: Union[ToolCall, ToolChain], instruction: str

    ) -> ToolChain:

        """

        Parses and validates one or more tool calls for execution.

        Accepts either a single ToolCall (representing a one-step tool plan)

        or a ToolChain (multi-step plan containing a list of ToolCalls), both as

        Pydantic models.

        For each tool call:

            - Verifies that the tool name exists in the current tool registry.

            - Filters parameters to include only those defined in the registry schema.

            - Logs and falls back to a default response if any tool is unknown or if no

                valid steps remain.

        Args:

            tool\_or\_tools (Union[ToolCall, ToolChain]):

                A ToolCall instance (for a single-step plan) or a ToolChain instance

                (for a multi-step plan).

            instruction (str):

                The original user instruction, used for logging and fallback response

                    generation.

        Returns:

            ToolChain: A validated multi-step plan. If validation fails, returns

                a fallback ToolChain.

        """

        # Normalize to a list of ToolCall for processing

        steps = []

        # Accept raw dicts/lists for extra robustness

        if isinstance(tool\_or\_tools, ToolCall):

            steps = [tool\_or\_tools]

        elif isinstance(tool\_or\_tools, ToolChain):

            steps = tool\_or\_tools.steps

        elif isinstance(tool\_or\_tools, dict):

            steps = [ToolCall(\*\*tool\_or\_tools)]

        elif isinstance(tool\_or\_tools, list):

            # Each item should be dict or ToolCall

            for item in tool\_or\_tools:

                if isinstance(item, ToolCall):

                    steps.append(item)

                elif isinstance(item, dict):

                    steps.append(ToolCall(\*\*item))

                else:

                    logger.error(

                        f"[Planner] Unknown item type in tool list: {type(item)}"

                    )

                    return self.\_fallback\_llm\_response(instruction)

        else:

            logger.error(

                f"[Planner] input is not ToolCall, ToolChain, dict, or list: got {type(tool\_or\_tools)}"

            )

            return self.\_fallback\_llm\_response(instruction)

        validated\_tools = []

        for i, step in enumerate(steps):

            tool\_name = step.tool

            params = step.params

            # Check tool exists

            if tool\_name not in self.tool\_registry.tools:

                logger.error(

                    f"[Planner] Unknown tool at step {i}: '{tool\_name}'. Fallback."

                )

                return self.\_fallback\_llm\_response(instruction)

            # Filter/normalize params for registry

            valid\_keys = set(

                self.tool\_registry.tools[tool\_name]

                .get("parameters", {})

                .get("properties", {})

                .keys()

            )

            filtered\_params = {k: v for k, v in params.items() if k in valid\_keys}

            if len(filtered\_params) != len(params):

                logger.info(

                    f"[Planner] Extra params filtered for tool '{tool\_name}': {set(params) - valid\_keys}"

                )

            validated\_tools.append(ToolCall(tool=tool\_name, params=filtered\_params))

        if not validated\_tools:

            logger.warning(

                "[Planner] All tool calls invalid or filtered out. Fallback."

            )

            return self.\_fallback\_llm\_response(instruction)

        logger.info(

            f"[Planner] Returning ToolChain with {len(validated\_tools)} validated tool call(s)."

        )

        return ToolChain(steps=validated\_tools)

Code Walk Through:  
The provided Python code, \_parse\_and\_validate\_tool\_calls, is a function designed to **parse and validate tool call requests** that are part of an automated planning system. The function takes a user instruction and a potential tool call or a series of tool calls and processes them to ensure they are valid and can be executed.

**Function Purpose**

The primary purpose of this function is to act as a **gatekeeper** for tool execution requests. It sanitizes and standardizes the input, ensuring that the requested tools and their parameters are well-formed and exist within a predefined registry. If any part of the request is invalid, the function logs the error and falls back to a safe default response.

**Step-by-Step Walkthrough**

**1. Input Normalization**

The function begins by taking the input tool\_or\_tools, which can be a single ToolCall, a ToolChain (a list of ToolCalls), or even raw Python dicts or lists. The first block of if/elif statements handles this variety of input types, normalizing all of them into a single format: a **list of ToolCall objects** stored in the steps variable.

* if isinstance(tool\_or\_tools, ToolCall): If the input is a single ToolCall object, it's wrapped in a list.
* elif isinstance(tool\_or\_tools, ToolChain): If the input is a ToolChain object, the function extracts the list of steps from it.
* elif isinstance(tool\_or\_tools, dict): If a raw dictionary is provided, it's assumed to be a single tool call and is converted into a ToolCall Pydantic model.
* elif isinstance(tool\_or\_tools, list): If a list is provided, the function iterates through each item. It converts any dictionary in the list into a ToolCall object and appends existing ToolCall objects to the steps list.
* else: If the input type is not recognized, an error is logged, and the function immediately calls \_fallback\_llm\_response, which likely returns a safe, non-tool-related response to the user.

**2. Validation Loop**

After normalizing the input into a list of ToolCalls, the function enters a for loop to validate each tool call individually. A new list, validated\_tools, is initialized to store the successfully validated calls.

* **Tool Name Check**: For each step (a ToolCall object), the code extracts the tool\_name from step.tool. It then checks if this tool\_name exists as a key in self.tool\_registry.tools. The tool\_registry is an internal object that holds information about all available tools. If the tool name is not found, an error is logged, and the function falls back to a default response, as the request cannot be fulfilled.
* **Parameter Filtering**: If the tool name is valid, the code proceeds to validate the parameters. It retrieves the schema for the tool from the tool\_registry, specifically the names of the valid parameters (valid\_keys). It then creates a new dictionary, filtered\_params, which only includes parameters from the original request (step.params) that match the valid\_keys in the schema. This step is crucial for security and robustness, as it prevents unexpected or unauthorized parameters from being passed to a tool.
* **Append to Validated List**: Finally, a **new ToolCall object** is created with the validated tool\_name and the filtered\_params. This new, sanitized object is then appended to the validated\_tools list.

**3. Final Check and Return**

After the loop completes, the function checks if the validated\_tools list is empty.

* if not validated\_tools: If the list is empty, it means all the original tool calls were either invalid or had their parameters filtered out completely. In this case, a warning is logged, and the function again calls the fallback response. This prevents the system from trying to execute a plan with no valid steps.
* else: If the validated\_tools list contains one or more items, the function constructs and returns a new **ToolChain object** containing the successfully validated steps. This ToolChain object is the final, executable plan. A confirmation message is logged, indicating the number of validated tool calls.

# tools/tool\_models.py

class ToolTransformation(BaseModel):

source\_tool: str

target\_tool: str

transform\_fn: str # e.g., "tools.transformations.dir\_to\_files"

[

{

"source\_tool": "find\_dir\_structure",

"target\_tool": "read\_files",

"transform\_fn": "tools.transformations.dir\_to\_files"

},

{

"source\_tool": "find\_files\_by\_keyword",

"target\_tool": "read\_files",

"transform\_fn": null # Direct match, no transformation needed

}

]

# tools/tool\_registry.py

class ToolRegistry:

def \_\_init\_\_(self, registry\_path=TOOL\_REGISTRY, transform\_path=TOOL\_TRANSFORMATIONS):

self.registry\_path = Path(registry\_path)

self.transform\_path = Path(transform\_path) if transform\_path else None

self.tools: Dict[str, ToolSpec] = self.\_load\_and\_validate\_tools()

self.transformations: List[ToolTransformation] = self.\_load\_transformations()

def \_load\_transformations(self) -> List[ToolTransformation]:

if not self.transform\_path or not self.transform\_path.exists():

return []

data = json.loads(self.transform\_path.read\_text())

return [ToolTransformation.model\_validate(item) for item in data]

def get\_transform\_fn(self, source\_tool: str, target\_tool: str) -> Callable[[Any], Any] | None:

for transform in self.transformations:

if transform.source\_tool == source\_tool and transform.target\_tool == target\_tool:

return self.\_load\_transform\_fn(transform.transform\_fn) if transform.transform\_fn else None

return None

def match\_and\_convert\_output(

self,

output: ToolOutput,

target\_tool: str,

transform\_fn: Optional[Callable[[Any], Any]] = None

) -> BaseModel:

spec = self.tools.get(target\_tool)

if not spec:

raise ValueError(f"Unknown target tool: {target\_tool}")

input\_model = self.\_load\_model(spec.input\_model)

if not input\_model:

raise ValueError(f"Input model for tool '{target\_tool}' not found or invalid")

# Use transform\_fn from caller or transformations

if not transform\_fn and hasattr(output, "tool"):

transform\_fn = self.get\_transform\_fn(output.tool, target\_tool)

payload = output.result

if transform\_fn:

try:

payload = transform\_fn(payload)

except Exception as e:

raise ValueError(f"Transformation failed: {e}")

try:

if isinstance(payload, BaseModel):

payload = payload.model\_dump()

return input\_model.model\_validate(payload if isinstance(payload, dict) else {"result": payload})

except ValidationError as e:

raise ValueError(f"Output does not match {input\_model.\_\_name\_\_}: {e}")

# tools/transformations.py

def dir\_to\_files(node: DirectoryNode) -> dict:

files = []

def traverse(n: DirectoryNode):

if n.type == "file":

files.append(n.name)

elif n.children:

for child in n.children:

traverse(child)

traverse(node)

return {"path": files} # Matches ReadFilesInput.path

class ToolOutput(BaseModel):

status: StepStatus

message: Optional[str] = None

result: Any

tool: Optional[str] = None # Source tool name

Update the executor to set tool in ToolOutput:

python

result = tool\_fn(\*\*params)

previous\_output = ToolOutput.model\_validate({\*\*result, "tool": tool\_name})

**Usage Example**:

registry = ToolRegistry(transform\_path="tool\_transformations.json")

dir\_output = DirStructureResult(

status=StepStatus.SUCCESS,

result=DirectoryNode(

name="root", type="directory",

children=[

DirectoryNode(name="file1.txt", type="file"),

DirectoryNode(name="sub", type="directory", children=[

DirectoryNode(name="file2.txt", type="file")

])

]

),

tool="find\_dir\_structure"

)

read\_files\_input = registry.match\_and\_convert\_output(

output=dir\_output,

target\_tool="read\_files"

)

print(read\_files\_input) # ReadFilesInput(path=['file1.txt', 'file2.txt'])