```python
import os
from typing import List, Dict, Any, Optional, Callable, get_type_hints
from dataclasses import dataclass, field
import json
from datetime import datetime
import inspect
import typing
from typing import Union
from swarms import Agent
from swarm_models import OpenAIChat


@dataclass
class ToolDefinition:
    name: str
    description: str
    parameters: Dict[str, Any]
    required_params: List[str]
    callable: Optional[Callable] = None


def extract_type_hints(func: Callable) -> Dict[str, Any]:
    """Extract parameter types from function type hints."""
    return typing.get_type_hints(func)
```

```
def extract_tool_info(func: Callable) -> ToolDefinition:
    """Extract tool information from a callable function."""
    # Get function name
    name = func.__name__

    # Get docstring
    description = inspect.getdoc(func) or "No description available"

    # Get parameters and their types
    signature = inspect.signature(func)
    type_hints = extract_type_hints(func)

    parameters = {}
    required_params = []

    for param_name, param in signature.parameters.items():
        # Skip self parameter for methods
        if param_name == "self":
            continue

        param_type = type_hints.get(param_name, Any)

        # Handle optional parameters
        is_optional = (
            param.default != inspect.Parameter.empty
            or getattr(param_type, "__origin__", None) is Union
            and type(None) in param_type.__args__
        )

        if not is_optional:
            required_params.append(param_name)

        parameters[param_name] = {
            "type": str(param_type),
            "default": (
                None
                if param.default is inspect.Parameter.empty
                else param.default
            ),
            "required": not is_optional,
        }

    return ToolDefinition(
        name=name,
        description=description,
        parameters=parameters,
        required_params=required_params,
        callable=func,
    )


@dataclass
```

```
class FunctionSpec:
    """Specification for a callable tool function."""

    name: str
    description: str
    parameters: Dict[
        str, dict
    ]  # Contains type and description for each parameter
    return_type: str
    return_description: str


@dataclass
class ExecutionStep:
    """Represents a single step in the execution plan."""

    step_id: int
    function_name: str
    parameters: Dict[str, Any]
    expected_output: str
    completed: bool = False
    result: Any = None


@dataclass
class ExecutionContext:
    """Maintains state during execution."""

    task: str
    steps: List[ExecutionStep] = field(default_factory=list)
    results: Dict[int, Any] = field(default_factory=dict)
    current_step: int = 0
    history: List[Dict[str, Any]] = field(default_factory=list)


def func():
    pass

hints = get_type_hints(func)


class ToolAgent:
    def __init__(
        self,
        functions: List[Callable],
        openai_api_key: str,
        model_name: str = "gpt-4",
        temperature: float = 0.1,
    ):
        self.functions = {func.__name__: func for func in functions}
        self.function_specs = self._analyze_functions(functions)
        self.model = OpenAIChat(
            openai_api_key=openai_api_key,
            model_name=model_name,
            temperature=temperature,
        )

        self.system_prompt = self._create_system_prompt()
        self.agent = Agent(
            agent_name="Tool-Agent",
            system_prompt=self.system_prompt,
            llm=self.model,
            max_loops=1,
            verbose=True,
        )

    def _analyze_functions(
        self, functions: List[Callable]
    ) -> Dict[str, FunctionSpec]:
        """Analyze functions to create detailed specifications."""
        specs = {}
        for func in functions:
            hints = get_type_hints(func)
            sig = inspect.signature(func)
            doc = inspect.getdoc(func) or ""

            # Parse docstring for parameter descriptions
```

```
param_descriptions = {}

current_param = None

for line in doc.split("\n"):

    if ":param" in line:

        param_name = (

            line.split(":param")[1].split(":")[0].strip()

        )

        desc = line.split(":", 2)[-1].strip()

        param_descriptions[param_name] = desc

    elif ":return:" in line:

        return_desc = line.split(":return:")[1].strip()


# Build parameter specifications

parameters = {}

for name, param in sig.parameters.items():

    param_type = hints.get(name, Any)

    parameters[name] = {

        "type": str(param_type),

        "type_class": param_type,

        "description": param_descriptions.get(name, ""),

        "required": param.default == param.empty,

    }


specs[func.__name__] = FunctionSpec(

    name=func.__name__,

    description=doc.split("\n")[0],
                parameters=parameters,
                return_type=str(hints.get("return", Any)),
                return_description=(
                    return_desc if "return_desc" in locals() else ""
                ),
            )

        return specs

    def _create_system_prompt(self) -> str:
        """Create system prompt with detailed function specifications."""
        functions_desc = []
        for spec in self.function_specs.values():
            params_desc = []
            for name, details in spec.parameters.items():
                params_desc.append(
                    f"    - {name}: {details['type']} - {details['description']}"
                )

            functions_desc.append(
                f"""
Function: {spec.name}
Description: {spec.description}
Parameters:
{chr(10).join(params_desc)}
Returns: {spec.return_type} - {spec.return_description}
```

```
            """
            )

        return f"""You are an AI agent that creates and executes plans using available functions.

Available Functions:
{chr(10).join(functions_desc)}

You must respond in two formats depending on the phase:

1. Planning Phase:
{{
    "phase": "planning",
    "plan": {{
        "description": "Overall plan description",
        "steps": [
            {{
                "step_id": 1,
                "function": "function_name",
                "parameters": {{
                    "param1": "value1",
                    "param2": "value2"
                }},
                "purpose": "Why this step is needed"
            }}
        ]
    }}
}}

2. Execution Phase:
{{
    "phase": "execution",
    "analysis": "Analysis of current result",
    "next_action": {{
        "type": "continue|request_input|complete",
        "reason": "Why this action was chosen",
        "needed_input": {{}} # If requesting input
    }}
}}

Always:
- Use exact function names
- Ensure parameter types match specifications
- Provide clear reasoning for each decision
"""

    def _execute_function(
        self, spec: FunctionSpec, parameters: Dict[str, Any]
    ) -> Any:
        """Execute a function with type checking."""
        converted_params = {}
        for name, value in parameters.items():
            param_spec = spec.parameters[name]

            try:

                # Convert value to required type

                param_type = param_spec["type_class"]

                if param_type in (int, float, str, bool):

                    converted_params[name] = param_type(value)

                else:

                    converted_params[name] = value

            except (ValueError, TypeError) as e:

                raise ValueError(

                    f"Parameter '{name}' conversion failed: {str(e)}"

                )


        return self.functions[spec.name](**converted_params)


    def run(self, task: str) -> Dict[str, Any]:

        """Execute task with planning and step-by-step execution."""

        context = ExecutionContext(task=task)

        execution_log = {

            "task": task,

            "start_time": datetime.utcnow().isoformat(),

            "steps": [],

            "final_result": None,

        }


        try:
```

```
# Planning phase
plan_prompt = f"Create a plan to: {task}"
plan_response = self.agent.run(plan_prompt)
plan_data = json.loads(
    plan_response.replace("System:", "").strip()
)

# Convert plan to execution steps
for step in plan_data["plan"]["steps"]:
    context.steps.append(
        ExecutionStep(
            step_id=step["step_id"],
            function_name=step["function"],
            parameters=step["parameters"],
            expected_output=step["purpose"],
        )
    )

# Execution phase
while context.current_step < len(context.steps):
    step = context.steps[context.current_step]
    print(
        f"\nExecuting step {step.step_id}: {step.function_name}"
    )

    try:
```

```
# Execute function
spec = self.function_specs[step.function_name]
result = self._execute_function(
    spec, step.parameters
)
context.results[step.step_id] = result
step.completed = True
step.result = result

# Get agent's analysis
analysis_prompt = f"""
Step {step.step_id} completed:
Function: {step.function_name}
Result: {json.dumps(result)}
Remaining steps: {len(context.steps) - context.current_step - 1}

Analyze the result and decide next action.
"""


analysis_response = self.agent.run(
    analysis_prompt
)
analysis_data = json.loads(
    analysis_response.replace(
        "System:", ""
    ).strip()
    )

    execution_log["steps"].append(
        {
            "step_id": step.step_id,
            "function": step.function_name,
            "parameters": step.parameters,
            "result": result,
            "analysis": analysis_data,
        }
    )

    if (
        analysis_data["next_action"]["type"]
        == "complete"
    ):
        if (
            context.current_step
            < len(context.steps) - 1
        ):
            continue
        break

    context.current_step += 1

except Exception as e:
        print(f"Error in step {step.step_id}: {str(e)}")
        execution_log["steps"].append(
            {
                "step_id": step.step_id,
                "function": step.function_name,
                "parameters": step.parameters,
                "error": str(e),
            }
        )
        raise

# Final analysis
final_prompt = f"""
Task completed. Results:
{json.dumps(context.results, indent=2)}

Provide final analysis and recommendations.
"""

final_analysis = self.agent.run(final_prompt)
execution_log["final_result"] = {
    "success": True,
    "results": context.results,
    "analysis": json.loads(
        final_analysis.replace("System:", "").strip()
    ),
            }

        except Exception as e:
            execution_log["final_result"] = {
                "success": False,
                "error": str(e),
            }

        execution_log["end_time"] = datetime.utcnow().isoformat()
        return execution_log


def calculate_investment_return(
    principal: float, rate: float, years: int
) -> float:
    """Calculate investment return with compound interest.

    :param principal: Initial investment amount in dollars
    :param rate: Annual interest rate as decimal (e.g., 0.07 for 7%)
    :param years: Number of years to invest
    :return: Final investment value
    """
    return principal * (1 + rate) ** years


agent = ToolAgent(
    functions=[calculate_investment_return],
    openai_api_key=os.getenv("OPENAI_API_KEY"),
)

result = agent.run(
    "Calculate returns for $10000 invested at 7% for 10 years"
)
```