```
import concurrent.futures
from typing import Callable, Any, Dict, List
from swarms.utils.loguru_logger import initialize_logger


logger = initialize_logger(log_folder="func_calling_executor")


# def openai_tool_executor(
#     tools: List[Dict[str, Any]],
#     function_map: Dict[str, Callable],
#     verbose: bool = True,
#     return_as_string: bool = False,
#     *args,
#     **kwargs,
# ) -> Callable:
#     """
#     Creates a function that dynamically and concurrently executes multiple functions based on parameters specified
#     in a list of tool dictionaries, with extensive error handling and validation.

#     Args:
#         tools (List[Dict[str, Any]]): A list of dictionaries, each containing configuration for a tool, including parameters.
#         function_map (Dict[str, Callable]): A dictionary mapping function names to their corresponding callable functions.
#         verbose (bool): If True, enables verbose logging.
#         return_as_string (bool): If True, returns the results as a concatenated string.
```

```
#     Returns:
#         Callable: A function that, when called, executes the specified functions concurrently with the
parameters given.

#     Examples:
#     >>> def test_function(param1: int, param2: str) -> str:
#     ...     return f"Test function called with parameters: {param1}, {param2}"

#     >>> tool_executor = openai_tool_executor(
#     ...     tools=[
#     ...         {
#     ...             "type": "function",
#     ...             "function": {
#     ...                 "name": "test_function",
#     ...                 "parameters": {
#     ...                     "param1": 1,
#     ...                     "param2": "example"
#     ...                 }
#     ...             }
#     ...         }
#     ...     ],
#     ...     function_map={
#     ...         "test_function": test_function
#     ...     },
#     ...     return_as_string=True
```

```
#     ... )
#     >>> results = tool_executor()
#     >>> print(results)
#     """

#     def tool_executor():
#         # Prepare tasks for concurrent execution
#         results = []
#         logger.info(f"Executing {len(tools)} tools concurrently.")
#         with concurrent.futures.ThreadPoolExecutor() as executor:
#             futures = []
#             for tool in tools:
#                 if tool.get("type") != "function":
#                     continue  # Skip non-function tool entries

#                 function_info = tool.get("function", {})
#                 func_name = function_info.get("name")
#                 logger.info(f"Executing function: {func_name}")

#                 # Check if the function name is mapped to an actual function
#                 if func_name not in function_map:
#                     error_message = f"Function '{func_name}' not found in function map."
#                     logger.error(error_message)
#                     results.append(error_message)
#                     continue
```

```
#             # Validate parameters
#             params = function_info.get("parameters", {})
#             if not params:
#                 error_message = f"No parameters specified for function '{func_name}'."
#                 logger.error(error_message)
#                 results.append(error_message)
#                 continue

#             # Submit the function for execution
#             try:
#                 future = executor.submit(
#                     function_map[func_name], **params
#                 )
#                 futures.append((func_name, future))
#             except Exception as e:
#                 error_message = f"Failed to submit the function '{func_name}' for execution: {e}"
#                 logger.error(error_message)
#                 results.append(error_message)

#         # Gather results from all futures
#         for func_name, future in futures:
#             try:
#                 result = future.result()  # Collect result from future
#                 results.append(f"{func_name}: {result}")
#             except Exception as e:
#                 error_message = f"Error during execution of function '{func_name}': {e}"
```

```
#                     logger.error(error_message)
#                     results.append(error_message)

#         if return_as_string:
#             return "\n".join(results)

#         logger.info(f"Results: {results}")

#         return results

#     return tool_executor


def openai_tool_executor(
    tools: List[Dict[str, Any]],
    function_map: Dict[str, Callable],
    verbose: bool = True,
    return_as_string: bool = False,
    *args,
    **kwargs,
) -> Callable:
    def tool_executor():
        results = []
        logger.info(f"Executing {len(tools)} tools concurrently.")
        with concurrent.futures.ThreadPoolExecutor() as executor:
            futures = []
```

```
for tool in tools:
    if tool.get("type") != "function":
        continue

    function_info = tool.get("function", {})
    func_name = function_info.get("name")
    logger.info(f"Executing function: {func_name}")

    if func_name not in function_map:
        error_message = f"Function '{func_name}' not found in function map."
        logger.error(error_message)
        results.append(error_message)
        continue

    params = function_info.get("parameters", {})
    if not params:
        error_message = f"No parameters specified for function '{func_name}'."
        logger.error(error_message)
        results.append(error_message)
        continue

    if (
        "name" in params
        and params["name"] in function_map
    ):
        try:
```

```
            result = function_map[params["name"]](
                **params
            )
            results.append(f"{params['name']}: {result}")
        except Exception as e:
            error_message = f"Failed to execute the function '{params['name']}': {e}"
            logger.error(error_message)
            results.append(error_message)
        continue

    try:
        future = executor.submit(
            function_map[func_name], **params
        )
        futures.append((func_name, future))
    except Exception as e:
        error_message = f"Failed to submit the function '{func_name}' for execution: {e}"
        logger.error(error_message)
        results.append(error_message)

for func_name, future in futures:
    try:
        result = future.result()
        results.append(f"{func_name}: {result}")
    except Exception as e:
        error_message = f"Error during execution of function '{func_name}': {e}"
```

```
                logger.error(error_message)
                results.append(error_message)

        if return_as_string:
            return "\n".join(results)

        logger.info(f"Results: {results}")

        return results

    return tool_executor


# function_schema = {
#     "name": "execute",
#      "description": "Executes code on the user's machine **in the users local environment** and returns the output",
#     "parameters": {
#         "type": "object",
#         "properties": {
#             "language": {
#                 "type": "string",
#                         "description": "The programming language (required parameter to the `execute` function)",
#                 "enum": [
#                     # This will be filled dynamically with the languages OI has access to.
```

```
#                 ],
#             },
#             "code": {
#                 "type": "string",
#                 "description": "The code to execute (required)",
#             },
#         },
#         "required": ["language", "code"],
#     },
# }



# def execute(language: str, code: str):
#     """
#     Executes code on the user's machine **in the users local environment** and returns the output

#     Args:
#         language (str): The programming language (required parameter to the `execute` function)
#         code (str): The code to execute (required)

#     Returns:
#         str: The output of the code execution
#     """
#     # This function will be implemented by the user
#     return "Code execution not implemented yet"
```

```
# # Example execution
# out = openai_tool_executor(
#     tools=[function_schema],
#     function_map={
#         "execute": execute,
#     },
#     return_as_string=True,
# )
# print(out)
```