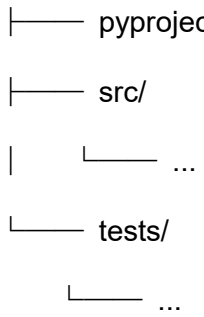
# MCP PA Execution Instructions and Test Results

## Execution Instructions

1.Extract the downloaded archive. You will see the following directory structure:
CS550MCP/



2.Next, make sure your Python version is ⩾ 3.10. Then, install uv using the following command:

**pip install uv**

3.In the directory containing pyproject.toml, run:

**uv venv**

**uv sync**

This will install the required packages: fastapi, uvicorn, and pytest.

4.Activate the uv virtual environment and start the FastAPI server with the following command:

**uvicorn src.server:app --reload**

By default, the server listens on: http://127.0.0.1:8000/mcp

If the server runs successfully, you should see output similar to the figure shown below.

```
INFO:     Will watch for changes in these directories: ['D:\\DEV\\CS550MCP']
INFO:     Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
INFO:     Started reloader process [27236] using StatReload
INFO:     Started server process [11632]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
INFO:     Shutting down
INFO:     Waiting for application shutdown.
INFO:     Application shutdown complete.
INFO:     Finished server process [11632]
INFO:     Stopping reloader process [27236]
```

5.Use pytest to verify the server's behavior:

**pytest -s**

This will run all test files under the tests/ directory and display the server responses.

## Server-side Code Description

server.py serves as the unified listening interface, while the core functionality is implemented in mcp_handlers.py.

```python
# src/server.py

from fastapi import FastAPI, Request
from fastapi.responses import JSONResponse
from src.mcp_handlers import handle_mcp_request

app = FastAPI()

@app.post("/mcp")
async def mcp_entrypoint(request: Request):
    """
    accept MCP JSON-RPC 2.0 request。
    """
    try:
        data = await request.json()
    except Exception:
        return JSONResponse(
            status_code=400,
            content={
                "jsonrpc": "2.0",
                "error": {"code": -32700, "message": "Parse error: Invalid JSON"},
                "id": None
            }
        )

    response = handle_mcp_request(data)
    return JSONResponse(content=response)
```

mcp_handlers.py includes two core functionalities and two additional features.

In addition, handle_mcp_request serves as the main entry point for handling MCP requests. It is responsible for parsing the JSON-RPC request data, identifying the method name, and dispatching the request to the corresponding handler function.

1.handle_mcp_request(data: Dict[str, Any]) → Dict[str, Any]

Function:

Main entry point for processing MCP requests. It parses the JSON-RPC request data, identifies the method name, and dispatches the request to the corresponding handler function.

Key Responsibilities:

Validate the JSON-RPC protocol version

Dispatch the request to either mcp/listResources or mcp/callTool

Catch runtime exceptions and wrap them into standard JSON-RPC error responses

Ensure each response includes both the "jsonrpc" and "id" fields

```python
def handle_mcp_request(data: Dict[str, Any]) -> Dict[str, Any]:
    response = {
        "jsonrpc": "2.0",
        "id": data.get("id")
    }

    if data.get("jsonrpc") != "2.0":
        response["error"] = _make_error(-32600, "Invalid JSON-RPC version")
        return response

    method = data.get("method")
    params = data.get("params", {})

    if method == "mcp/listResources":
        response["result"] = handle_list_resources()
    elif method == "mcp/callTool":
        try:
            response["result"] = handle_call_tool(params)
        except Exception as e:
            response["error"] = _make_error(-32000, f"Tool error: {str(e)}")
    else:
        response["error"] = _make_error(-32601, f"Method '{method}' not found")

    return response
```

2.handle_list_resources() → list

Function:

Returns a simulated list of resources, representing the “discoverable resources” currently supported by the server.

Returned Items Include:

A directory for simulating HDF5 files

A mock Arxiv API endpoint

A hypothetical data compression tool

A parallel file system path

```python
def handle_list_resources() -> list:
    return [
        {
            "name": "SimHDF5Files",
            "type": "filesystem",
            "path": "./mock_data/hdf5"
        },
        {
            "name": "MockArxivFetcher",
            "type": "external-api",
            "endpoint": "arxiv.org"
        },
        {
            "name": "CompressionUtility",
            "type": "tool",
            "description": "Simulates compressing log files using gzip"
        },
        {
            "name": "ParallelFSProjectDir",
            "type": "parallel-fs",
            "path": "/pfs/project_x"
        }
    ]
```

3.handle_call_tool(params: Dict[str, Any]) → Dict[str, Any]

Function:

Invokes the corresponding simulated tool execution function (such as the Slurm or HDF5 tool) based on the provided tool name. Example Parameter Structure:

```json
{
  "tool": "slurm",
  "params": {
    "script": "run.sh",
    "cores": 4
  }
}
```

Error Handling: If the tool name is not supported, a ValueError is raised and wrapped by the caller into a JSON-RPC error response.

```python
def handle_call_tool(params: Dict[str, Any]) -> Dict[str, Any]:
    tool_name = params.get("tool")
    tool_params = params.get("params", {})

    if tool_name == "slurm":
        return _simulate_slurm_tool(tool_params)
    elif tool_name == "hdf5":
        return _simulate_hdf5_tool(tool_params)
    else:
        raise ValueError(f"Unsupported tool '{tool_name}'")
```

4._simulate_slurm_tool(params: Dict[str, Any]) → Dict[str, Any]

Function:

Simulates a Slurm job submission operation, generating a virtual job ID and returning task status information.

Supported Parameters:

script: Name of the job script file (optional; defaults to "unnamed.sh")

cores: Number of CPU cores requested (defaults to 1)

Returned Fields:

job_id: A randomly generated job identifier

message: A brief message describing the submission result

status: Always "success"

```python
def _simulate_slurm_tool(params: Dict[str, Any]) -> Dict[str, Any]:
    """
    Simulates Slurm job submission.
    Example parameters:
      { "script": "run.sh", "cores": 4 }
    """
    script = params.get("script", "unnamed.sh")
    cores = params.get("cores", 1)

    # Simulate printed message and generate fake job_id
    job_id = random.randint(10000, 99999)
    message = f"Job '{script}' submitted using {cores} core(s)."
    print(f"[Slurm] {message} -> Job ID: {job_id}")

    return {
        "status": "success",
        "job_id": job_id,
        "message": message
    }
```

5._simulate_hdf5_tool(params: Dict[str, Any]) → Dict[str, Any]

Function:

Simulates wildcard searching for .hdf5 files in the local file system and returns a list of matching file paths.

Processing Logic:

Extract the base directory and glob pattern from the pattern parameter

Use pathlib.Path.glob() to find files matching the pattern

Raise an exception if the path does not exist or the pattern is invalid

Returned Fields:

matches: A list of absolute paths that matched the pattern

pattern: The original pattern provided

status: Always "success"

```python
def _simulate_hdf5_tool(params: Dict[str, Any]) -> Dict[str, Any]:
    """
    Simulates searching for HDF5 file paths.
    Example parameters:
      { "pattern": "./mock_data/hdf5/**/*.hdf5" }
    """
    pattern = params.get("pattern")
    if not pattern:
        raise ValueError("Missing 'pattern' in hdf5 params")

    # Split the path into base directory and glob pattern
    pattern_path = pathlib.Path(pattern)
    parts = pattern_path.parts

    # Find the first part containing a wildcard
    for i, part in enumerate(parts):
        if "*" in part or "?" in part:
            base_dir = pathlib.Path(*parts[:i])
            glob_pattern = str(pathlib.Path(*parts[i:]))
            break
    else:
        # No wildcard found, raise error
        raise ValueError("Pattern must contain a glob expression like *.hdf5")

    base_dir = base_dir.resolve()
    if not base_dir.exists():
        raise ValueError(f"Base path does not exist: {base_dir}")

    matched_files = [str(p.resolve()) for p in base_dir.glob(glob_pattern)]

    print(f"[HDF5] Matching from '{base_dir}' with pattern '{glob_pattern}': {len(matched_files)} file(s)")

    return {
        "status": "success",
        "pattern": pattern,
        "matches": matched_files
    }
```

6._make_error(code: int, message: str) → Dict[str, Any]

Function:

Constructs a standardized JSON-RPC error response object for consistent error

handling and formatting.

```
def _make_error(code: int, message: str) -> Dict[str, Any]:
    return {
        "code": code,
        "message": message
    }
```

## Test Description and Results

The test covers the responses to listResources as well as mcp/callTool requests targeting the "slurm" and "hdf5" tools.

The specific request contents used in the test files are listed below.

1.Correct cases

1) test_valid_list_resources

```
def test_valid_list_resources():
    request = {
        "jsonrpc": "2.0",
        "method": "mcp/listResources",
        "params": {},
        "id": "rsc_1"
    }
    response = handle_mcp_request(request)
    print("\n[listResources OK]:", response)

    assert "result" in response
    assert isinstance(response["result"], list)
    assert len(response["result"]) >= 3
```

2) test_valid_slurm_tool

```python
def test_valid_slurm_tool():
    request = {
        "jsonrpc": "2.0",
        "method": "mcp/callTool",
        "params": {
            "tool": "slurm",
            "params": {
                "script": "train.sh",
                "cores": 4
            }
        },
        "id": "slurm_1"
    }
    response = handle_mcp_request(request)
    print("\n[slurm OK]:", response)

    assert "result" in response
    assert response["result"]["status"] == "success"
    assert "job_id" in response["result"]
```

3) test_valid_hdf5_tool

```python
def test_valid_hdf5_tool(tmp_path):
    hdf5_dir = tmp_path / "mock_data" / "hdf5"
    hdf5_dir.mkdir(parents=True)
    (hdf5_dir / "a.hdf5").touch()
    (hdf5_dir / "b.hdf5").touch()

    pattern = str(hdf5_dir / "*.hdf5")

    request = {
        "jsonrpc": "2.0",
        "method": "mcp/callTool",
        "params": {
            "tool": "hdf5",
            "params": {
                "pattern": pattern
            }
        },
        "id": "hdf5_1"
    }
    response = handle_mcp_request(request)
    print("\n[hdf5 OK]:", response)

    assert "result" in response
    assert response["result"]["status"] == "success"
    assert len(response["result"]["matches"]) == 2
```

2.Wrong cases

1) test_invalid_method_name

Called an MCP method name not implemented by the server: "mcp/unknown"

```python
def test_invalid_method_name():
    request = {
        "jsonrpc": "2.0",
        "method": "mcp/unknown",
        "params": {},
        "id": "fail_1"
    }
    response = handle_mcp_request(request)
    print("\n[method FAIL]:", response)

    assert "error" in response
    assert response["error"]["code"] == -32601


def test_invalid_slurm_missing_script():
    request = {
        "jsonrpc": "2.0",
        "method": "mcp/callTool",
        "params": {
            "tool": "slurm",
            "params": {
                # "script" is missing
                "cores": 2
            }
        },
        "id": "fail_2"
    }
    response = handle_mcp_request(request)
    print("\n[slurm param FAIL]:", response)

    assert "result" in response  # still success but using default script name
    assert response["result"]["status"] == "success"
    assert "job_id" in response["result"]
```

2) test_invalid_slurm_missing_script

The "script" field is missing in the parameters; this is the required task name for the Slurm tool.

```python
def test_invalid_slurm_missing_script():
    request = {
        "jsonrpc": "2.0",
        "method": "mcp/callTool",
        "params": {
            "tool": "slurm",
            "params": {
                # "script" is missing
                "cores": 2
            }
        },
        "id": "fail_2"
    }
    response = handle_mcp_request(request)
    print("\n[slurm param FAIL]:", response)

    assert "result" in response  # still success but using default script name
    assert response["result"]["status"] == "success"
    assert "job_id" in response["result"]
```

3) test_invalid_hdf5_no_match

The provided pattern includes a non-existent directory path.

```python
def test_invalid_hdf5_no_match():
    request = {
        "jsonrpc": "2.0",
        "method": "mcp/callTool",
        "params": {
            "tool": "hdf5",
            "params": {
                "pattern": "./nonexistent_dir/**/*.hdf5"
            }
        },
        "id": "fail_3"
    }
    response = handle_mcp_request(request)
    print("\n[hdf5 not found FAIL]:", response)

    assert "error" in response
    assert response["error"]["code"] == -32000
```

## Test result：

After running all test cases, the results are shown as follows. A total of six test results were returned, each including the server's response.

```
(.venv) D:\DEV\CS550MCP>pytest -s
============================================================================================= test session starts
=============================================================================================
platform win32 -- Python 3.13.3, pytest-8.3.5, pluggy-1.5.0
rootdir: D:\DEV\CS550MCP
configfile: pyproject.toml
plugins: anyio-4.9.0
collected 6 items

test\test_mcp_handlers.py
[listResources OK]: {'jsonrpc': '2.0', 'id': 'rsc_1', 'result': [{'name': 'SimHDF5Files', 'type': 'filesystem', 'p
ath': './mock_data/hdf5'}, {'name': 'MockArxivFetcher', 'type': 'external-api', 'endpoint': 'arxiv.org'}, {'name':
 'CompressionUtility', 'type': 'tool', 'description': 'Simulates compressing log files using gzip'}, {'name': 'Par
allelFSProjectDir', 'type': 'parallel-fs', 'path': '/pfs/project_x'}]}
.[Slurm] Job 'train.sh' submitted using 4 core(s). -> Job ID: 22399

[slurm OK]: {'jsonrpc': '2.0', 'id': 'slurm_1', 'result': {'status': 'success', 'job_id': 22399, 'message': "Job '
train.sh' submitted using 4 core(s)."}}
.[HDF5] Matching from 'C:\Users\Admin\AppData\Local\Temp\pytest-of-Admin\pytest-2\test_valid_hdf5_tool0\mock_data\
hdf5' with pattern '*.hdf5': 2 file(s)

[hdf5 OK]: {'jsonrpc': '2.0', 'id': 'hdf5_1', 'result': {'status': 'success', 'pattern': 'C:\\Users\\Admin\\AppDat
a\\Local\\Temp\\pytest-of-Admin\\pytest-2\\test_valid_hdf5_tool0\\mock_data\\hdf5\\*.hdf5', 'matches': ['C:\\Users
\\Admin\\AppData\\Local\\Temp\\pytest-of-Admin\\pytest-2\\test_valid_hdf5_tool0\\mock_data\\hdf5\\a.hdf5', 'C:\\Us
ers\\Admin\\AppData\\Local\\Temp\\pytest-of-Admin\\pytest-2\\test_valid_hdf5_tool0\\mock_data\\hdf5\\b.hdf5']}}
.
[method FAIL]: {'jsonrpc': '2.0', 'id': 'fail_1', 'error': {'code': -32601, 'message': "Method 'mcp/unknown' not f
ound"}}
.[Slurm] Job 'unnamed.sh' submitted using 2 core(s). -> Job ID: 94573

[slurm param FAIL]: {'jsonrpc': '2.0', 'id': 'fail_2', 'result': {'status': 'success', 'job_id': 94573, 'message':
 "Job 'unnamed.sh' submitted using 2 core(s)."}}
.
[hdf5 not found FAIL]: {'jsonrpc': '2.0', 'id': 'fail_3', 'error': {'code': -32000, 'message': 'Tool error: Base p
ath does not exist: D:\\DEV\\CS550MCP\\nonexistent_dir'}}
.

============================================================================================= 6 passed in 0.03s =
=============================================================================================
```

The first result confirms that the server correctly responded to the mcp/listResources request and returned a list of registered simulated resources in the system.

The second result shows that the server successfully simulated a task submission using the Slurm tool and returned a virtual Job ID along with status information.

The third result indicates that the server successfully recognized and matched the provided HDF5 file path pattern, returning a list of matching files.

The fourth result shows that the client requested an undefined MCP method, and the server returned a “method not found” error according to the JSON-RPC specification.

The fifth result indicates that although the required script name was missing from the parameters, the server used the default name "unnamed.sh" to simulate the submission, which completed successfully.

The sixth result shows that the path pattern provided to the HDF5 tool included a non-existent directory, causing the simulated search to fail and returning a standard tool error.