Building and Applying AI Agents for Drug Discovery

Atsushi Yoshimori, Ph.D.
-Institute of Theoretical Medicine, Inc.
-Department of Life Science Informatics, B-IT
-Tokyo University of Science
-University of Shizuoka

Day 1: Developing foundational software tools for drug discovery.

Environment setup: Install software such as miniforge, jupyter lab, and RDKit.
Exercise_01: Visualization of SMILES
Exercise_02: Calculation of Molecular Properties
Exercise_03: Substructure Search
Exercise_04: Similarity Search

Day2: Developing MCP servers and integrating them with an LLM.

Environment setup: Install software LM Studio and Local LLM.
Exercise_01: Developing foundational MCP servers and integrating them with an LLM.
Exercise_02: Developing an MCP server for molecular properties and integrating it with an LLM.
Exercise_03: Integrating a ChEMBL MCP server (publicly available) with an LLM and accessing ChEMBL data via an LLM.
Exercises_02 build on the drug discovery tools developed on **Day 1.*

*** Python scripts for all Examples and Challenge exercises are available from the link below. Please try to solve the exercises on your own first. If you have difficulty finding a solution, you may refer to the scripts as needed.*
https://uni-bonn.sciebo.de/s/aJysxEGHzZqw8yQ

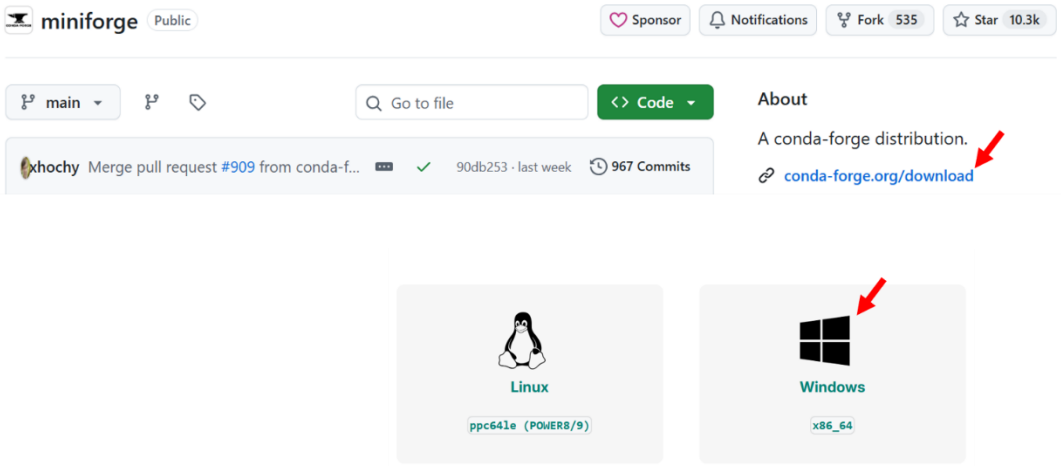
Day 1

Environment setup

Step 1: Instal Miniforge

Miniforge is a free and open-source installer for the Conda package manager. It provides a lightweight way to create and manage Python environments and install scientific computing packages.

Download **Miniforge3-Windows-x86_64.exe** from the following URL.
https://github.com/conda-forge/miniforge/

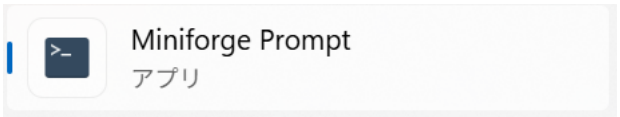


Double-click **Miniforge3-Windows-x86_64.exe**, then follow the on-screen instructions provided by the installer to complete the installation.

Type **"Miniforge"** into the Windows search box.



Then click **"Miniforge Prompt"** to launch and use Miniforge.



Step 2: Create a virtual environment using Conda

$ conda create -n mcp python=3.10 -y
$ conda activate mcp
$ pip install rdkit==2026.9.1
$ pip install slearn==0.3.0
$ pip install jupyterlab==4.6.4
$ pip install xgboost==3.2.0
$ pip install mcp==2.3.0

Step 3: JupyterLab Setup

Create the following folders on your Desktop or in any other preferred location.

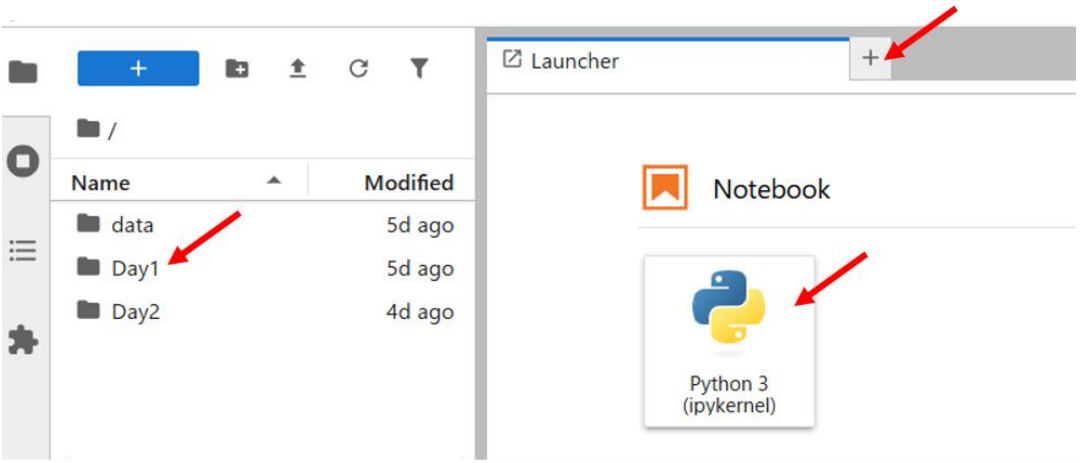
AI_Agent/
- **Day1/**
- **Day2/**
- **data/**

In the **Miniforge Prompt**, navigate to the following directory.
(base) $ cd [PATH TO **AI_Agent** folder]

Launch **JupyterLab**
(base) $ conda activate mcp
(mcp) $ jupyter lab

--User Guide for JupyterLab--

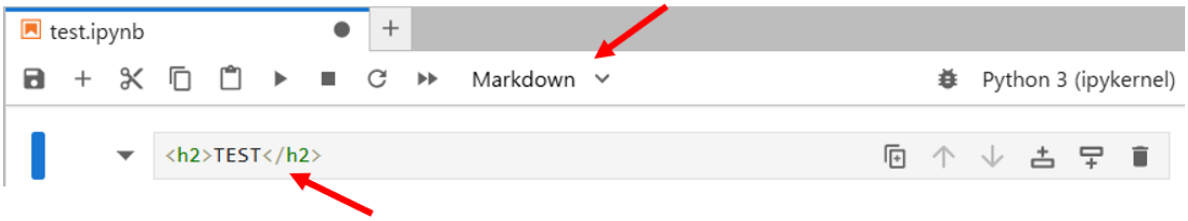
Step 3-1: Click Day1 folder and then Click Notebook



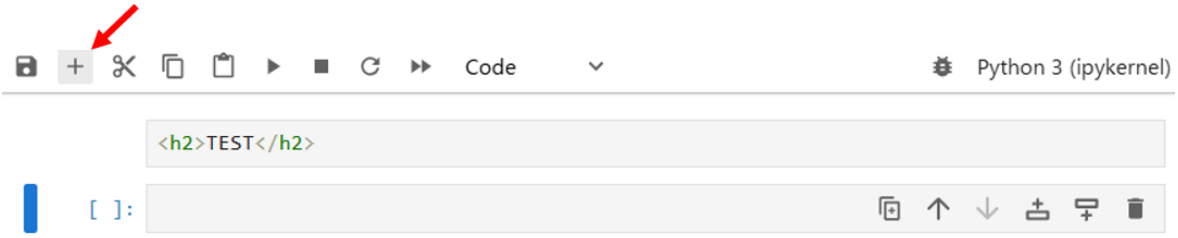
If the "**Launcher**" tab is not available, click the "**+**" button to create a new tab.

Step 3-2: [File]→[Rename Notebook]→test.ipynb

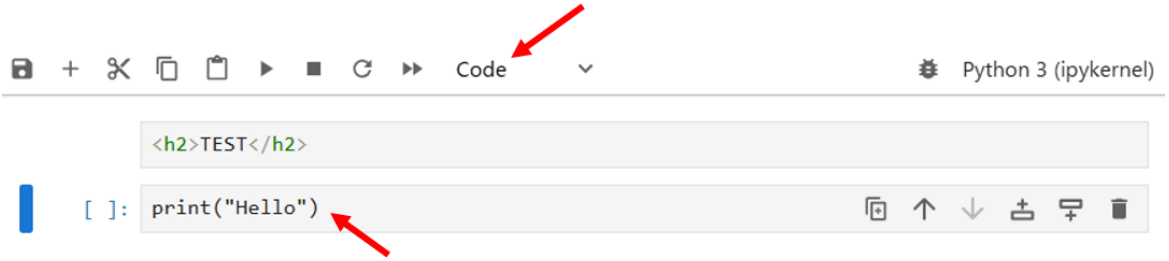
Step 3-3: To add a title or comments, select "**Markdown**" or "**Raw**" and enter the content in the cell using HTML.



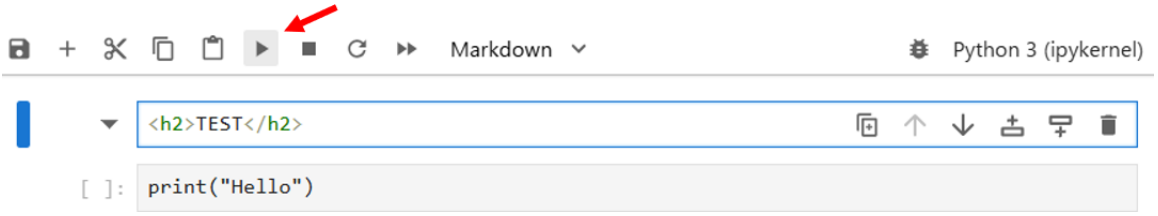
Step 3-4: Clicking the "**+**" button adds a new cell.



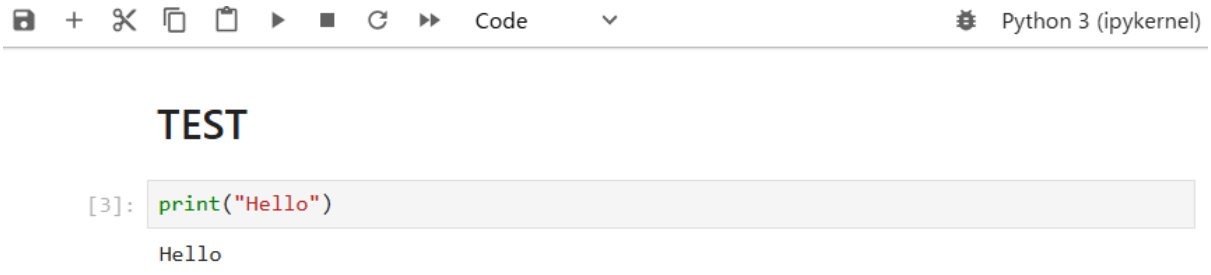
Step 3-5: Select "**Code**" and write a Python script.



Step 3-6: Select the first cell and click the "(▶)" button to execute it.



Step 3-7: After running all cells, the following output will be displayed.



Exercise 01: Visualization of SMILES

Enter the following **Example 01** into **JupyterLab** and run it.
Save the file as "**Example_01.ipynb**" under the Day1 folder.
The following shows the output after execution.

Example 01: Visualization of SMILES

[1]:
```
from rdkit import Chem
from rdkit.Chem import Draw, AllChem
```

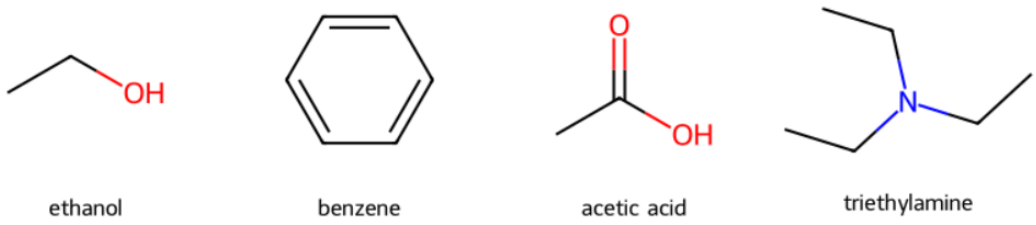
From SMILES to Molecule

[2]:
```
smiles_list = [
    "CCO",          # ethanol
    "c1ccccc1",     # benzene
    "CC(=O)O",      # acetic acid
    "CCN(CC)CC"     # triethylamine
]
legends = ["ethanol", "benzene", "acetic acid", "triethylamine"]
```

Molecular Visualization

[3]:
```
mols = [Chem.MolFromSmiles(smi) for smi in smiles_list]
Draw.MolsToGridImage(mols, molsPerRow=4, legends=legends, subImgSize=(200, 200))
```

[3]:



Molecular Visualization with Partial Atomic Charges

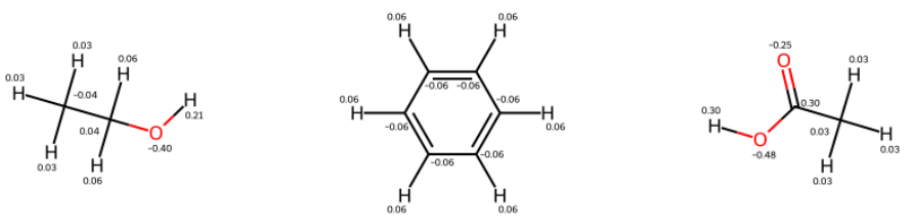
[4]:
```
mols = []
for smi in smiles_list:
    mol = Chem.MolFromSmiles(smi)
    mol_h = Chem.AddHs(mol)
    AllChem.ComputeGasteigerCharges(mol_h)

    for atom in mol_h.GetAtoms():
        q = float(atom.GetProp("_GasteigerCharge"))
        atom.SetProp("atomNote", f"{q:.2f}")

    mols.append(mol_h)
```

[5]:
```
Draw.MolsToGridImage(mols, molsPerRow=4, legends=legends, subImgSize=(300, 300))
```

[5]:



Challenge 01: Visualization of SMILES

Write and run a Python script in JupyterLab to visualize the following 10 molecules with partial atomic charges.

```
c1ccccc1O     # Phenol
Nc1ccccc1     # Aniline
C1CCCCC1      # Cyclohexane
c1ccncc1      # Pyridine
Cc1ccccc1     # Toluene
CCOC(=O)C     # Ethyl acetate
CC(=O)N       # Acetamide
CC(=O)C       # Acetone
O=C(O)c1ccccc1      # Benzoic acid
Cn1cnc2n(C)c(=O)n(C)c(=O)c12 # Caffeine
```

Save the file as "**Challenge_01.ipynb**" under the Day1 folder.

Exercise 02: Calculation of Molecular Properties

Enter the following **Example 02** into **JupyterLab** and run it.
Save the file as "**Example_02.ipynb**" under the Day1 folder.

The following shows the output after execution.

Example 02: Calculation of Molecular Properties

[2]:
```
from rdkit import Chem
from rdkit.Chem import Descriptors, Crippen
```

Function for MW/LogP calculation

[3]:
```
def calc_mw_logp(smiles: str) -> dict[str, float]:
    """
    Calculate molecular weight (MW) and LogP from a SMILES string.

    Parameters
    ----------
    smiles : str
        SMILES representation of a molecule.

    Returns
    -------
    dict[str, float]
        Dictionary containing:
        - MW: Molecular weight
        - LogP: Wildman-Crippen LogP
    """

    mol = Chem.MolFromSmiles(smiles)

    if mol is None:
        raise ValueError(f"Invalid SMILES: {smiles}")

    return {
        "MW": Descriptors.MolWt(mol),
        "LogP": Crippen.MolLogP(mol),
    }
```

(Continued)

(Continued)

MW/LogP calculation

[5]:
```
smiles_list = [
    "CCO",          # ethanol
    "c1ccccc1",     # benzene
    "CC(=O)O",      # acetic acid
    "CCN(CC)CC"     # triethylamine
]
```

[14]:
```
for smiles in smiles_list:
    res = calc_mw_logp(smiles)
    print(f"{smiles},MW:{res['MW']:.2f},LogP:{res['LogP']:.2f}")
```

```
CCO,MW:46.07,LogP:-0.00
c1ccccc1,MW:78.11,LogP:1.69
CC(=O)O,MW:60.05,LogP:0.09
CCN(CC)CC,MW:101.19,LogP:1.35
```

Challenge 02: Calculate of Moleculer Properties

Write and run a Python script in JupyterLab to calculate **MW**, **LogP**, **TPSA**, and **Fsp3**.

Hint:

Mw, logP, TPSA, and Fsp3 can be calculated using the following functions.

```
from rdkit.Chem import Descriptors, Crippen, rdMolDescriptors
# Molecular weight
mw = Descriptors.MolWt(mol)
# Wildman-Crippen LogP
logp = Crippen.MolLogP(mol)
# Topological polar surface area
tpsa = rdMolDescriptors.CalcTPSA(mol)
# Fraction of sp3-hybridized carbons
fsp3 = rdMolDescriptors.CalcFractionCSP3(mol)
```

Save the file as "**Challenge_02.ipynb**" under the Day1 folder.

Exercise 03: Substructure Search

Enter the following **Example 03** into **JupyterLab** and run it.
Save the file as "**Example_03.ipynb**" under the Day1 folder.

Example 03: Substructure Search

[1]:
```
from rdkit import Chem
from rdkit.Chem import Draw
```

List of molecules

[2]:
```
smiles_list = [
    "c1ccccc1O",        # phenol
    "CCO",              # ethanol
    "c1ccncc1",         # pyridine
    "c1ccccc1C(=O)O"    # benzoic acid
]
```

Query substructure (benzene ring)

[3]:
```
query = Chem.MolFromSmiles("c1ccccc1")
```

Substructure search

[4]:
```
print("=== Substructure Search: Does it contain a benzene ring? ===")
for s in smiles_list:
    mol = Chem.MolFromSmiles(s)
    # Check if the molecule contains the query substructure
    if mol.HasSubstructMatch(query):
        print(f"{s} → contains benzene ring")
    else:
        print(f"{s} → does NOT contain benzene ring")
```

```
=== Substructure Search: Does it contain a benzene ring? ===
c1ccccc1O → contains benzene ring
CCO → does NOT contain benzene ring
c1ccncc1 → does NOT contain benzene ring
c1ccccc1C(=O)O → contains benzene ring
```

(Continued)

(Continued)

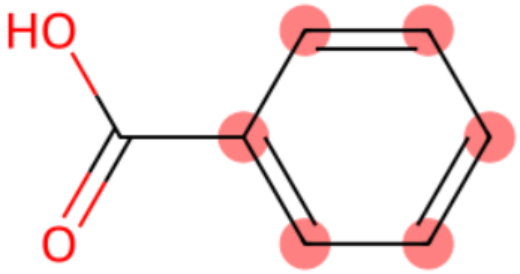
Visualization of Matched Atoms

```
[5]: mol = Chem.MolFromSmiles("c1ccccc1C(=O)O")
     query = Chem.MolFromSmiles("c1ccccc1")

     matches = mol.GetSubstructMatches(query)
     print("matched atoms")
     print(matches)
     highlight_atoms = matches[0]
     Draw.MolToImage(mol, highlightAtoms=highlight_atoms,size=(300, 300))
```

```
matched atoms
((0, 1, 2, 3, 4, 5),)
```

[5]:



Challenge 03: Substructure Search

Write and run a Python script in JupyterLab to search for the carboxylic acid group "C(=O)O" within the following SMILES strings.

CC(=O)O
c1ccccc1O
c1ccccc1C(=O)O
CCN
COC(=O)O

Save the file as "**Challenge_03.ipynb**" under the Day1 folder.

Exercise 04: Similarity Search

Enter the following **Example 04** into **JupyterLab** and run it.
Save the file as "**Example_04.ipynb**" under the Day1 folder.

Example 04: Similarity Search

[1]:
```
from rdkit import Chem
from rdkit.Chem import rdFingerprintGenerator, DataStructs
```

Compute fingerprints of database molecules

[2]:
```
# MorganGenerator for fingerprint generation
morgan_gen = rdFingerprintGenerator.GetMorganGenerator(radius=2, fpSize=2048)

# List of molecules for the similarity search database
smiles_list = ["CCO", "CCCO", "CCOC", "CCN", "COC"]
mols = [Chem.MolFromSmiles(s) for s in smiles_list]

# Generate Morgan fingerprints using the generator
fps = [morgan_gen.GetFingerprint(m) for m in mols]
```

Compute fingerprint of a query molecule

[3]:
```
# Define the query molecule (ethanol)
query = Chem.MolFromSmiles("CCO")
query_fp = morgan_gen.GetFingerprint(query)
```

Similarity Search

[4]:
```
# Compute Tanimoto similarity between query and each molecule
results = []
for smi, fp in zip(smiles_list, fps):
    sim = DataStructs.TanimotoSimilarity(query_fp, fp)
    results.append((smi, sim))

# Sort results by similarity (descending order)
results_sorted = sorted(results, key=lambda x: x[1], reverse=True)

# Print results
for smi, sim in results_sorted:
    print(f"{smi}: {sim:.3f}")
```

```
CCO: 1.000
CCCO: 0.556
CCN: 0.333
CCOC: 0.273
COC: 0.111
```

Challenge 04-1: Similarity Search

Write and run a Python script in JupyterLab to retrieve the 10 most similar molecules to the query molecule from the database molecules.

Read the database molecules from **CHEMBL2599_SYK.csv**
Use the following script as a reference for loading the database molecules:

```
import pandas as pd
df = pd.read_csv("CHEMBL2599_SYK.csv", sep=",")
smiles_db = df["SMILES"].tolist()
mols_db = [Chem.MolFromSmiles(s) for s in smiles_db]
fps_db = [morgan_gen.GetFingerprint(m) for m in mols_db]
```

Query molecule is "COc1cccc(-c2cc3ncccc3c(NCCCN)n2)c1".

*__CHEMBL2599_SYK.csv__ can be downloaded from the following link and is located in the data folder of the downloaded files.
https://uni-bonn.sciebo.de/s/aJysxEGHzZqw8yQ

Save the file as "**Challenge_04-1.ipynb**" under the Day1 folder.

Challenge 04-2: Similarity Search and Property Calculation

Please add a feature to "**Challenge_04-1.ipynb**" that displays the structures of the 10 similar compounds along with their similarity scores and molecular weights.

The similarity scores and molecular weights should be displayed using the legends parameter of the following function:

```
Draw.MolsToGridImage(
        mols,
        molsPerRow=5,
        legends=legends,
        subImgSize=(300, 300))
```

Save the file as "**Challenge_04-2.ipynb**" under the Day1 folder.

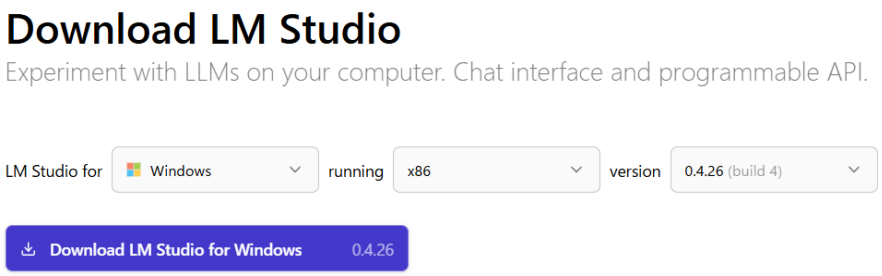
Day 2

Environment setup

Step 1: Instal LMStudio

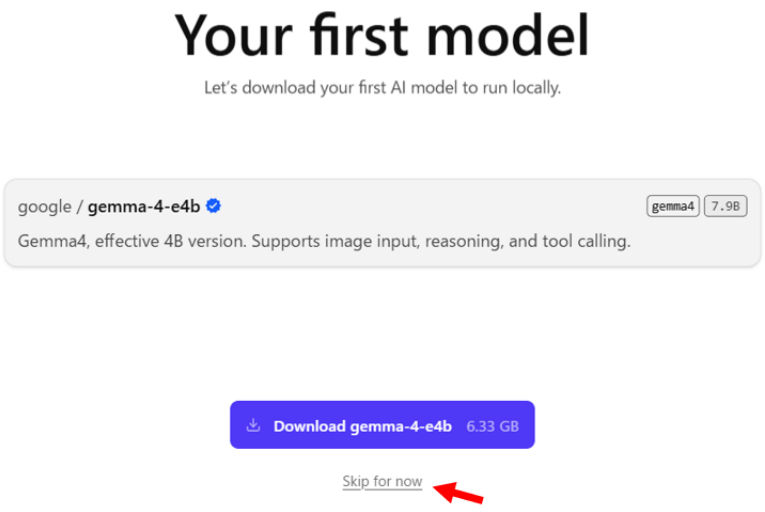
LM Studio is a desktop application that lets you run and interact with large language models (LLMs) locally on your own computer. It provides an easy-to-use interface for downloading, managing, and chatting with AI models, and it can also expose them through an OpenAI-compatible API for integration with other applications.

Download **LM-Studio-0.4.26-4-x64.exe** from the following URL.
https://lmstudio.ai/download

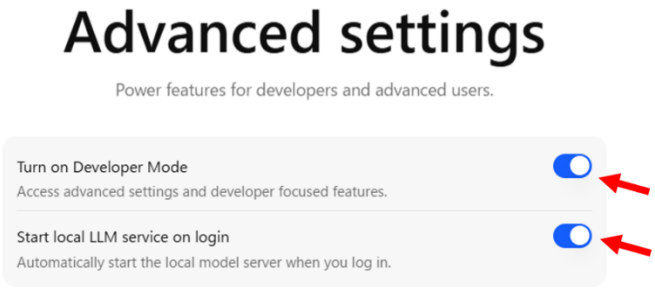


Double-click **LM-Studio-0.4.26-4-x64.exe**, then follow the on-screen instructions provided by the installer to complete the installation.

On the "**Your first model**" screen, select "**Skip for now**".



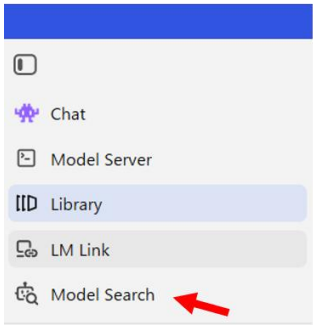
Enable all options in the "**Advanced Settings**" section.

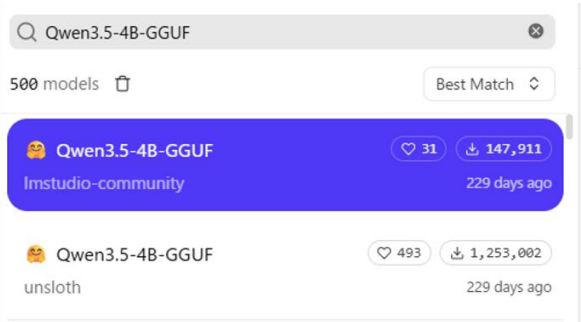


Step 2: LMStudio Setup

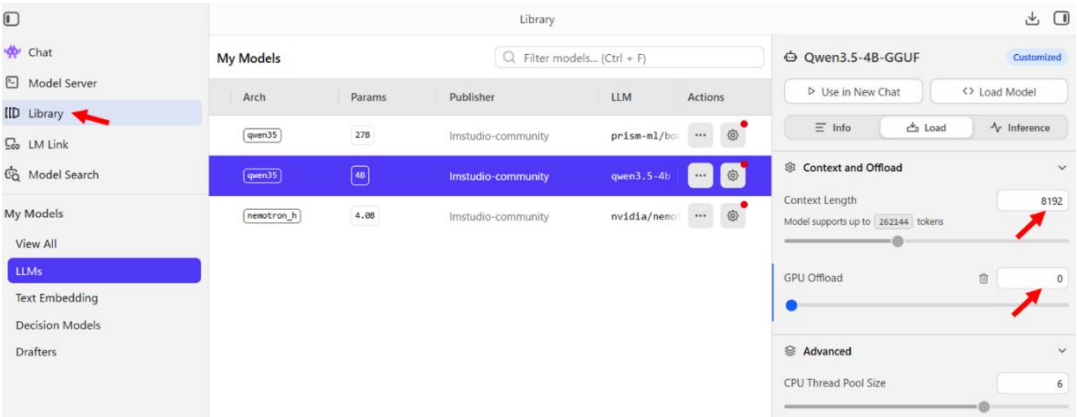
Step 2-1: Double-click the **LM Studio icon** on the desktop to launch LM Studio.

Step 2-2: Select "**Model Search**", enter "**Qwen3.5-4B-GGUF**" (LLM) in the search box, select the model from the search results, and download it.





Step 2-3: Configure the parameters for "**Qwen3.5-4B-GGUF**". Select "**Library**" and then choose "Qwen3.5-4B-GGUF" to access the parameter settings. Set "**Context Length**" to **8192** and "**GPU Offload**" to **0**. Depending on your PC's GPU performance, you may need to adjust the parameters.



Step 2-4: Repeat the same steps above to configure the LLM model "**Nemotron 3 Nano 4B**".

*During the download of LLM, you may carry out **Steps 1** through **6** of **Exercise_01**.

Exercise 01: Developing foundational MCP servers and integrating them with an LLM.

Step 1: Setting Up a Python Development Environment in JupyterLab

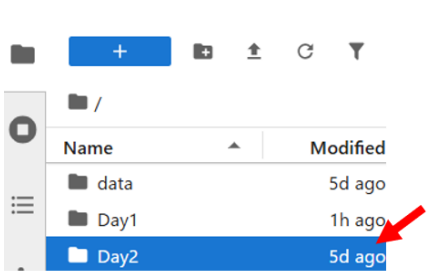
In the **Miniforge Prompt**, navigate to the following directory.

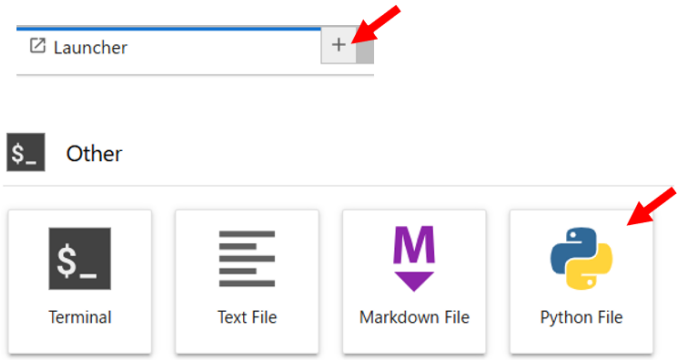
(base) $ cd [PATH TO **AI_Agent** folder]

Launch **JupyterLab**

(base) $ conda activate mcp

(mcp) $ jupyter lab





If the "**Launcher**" tab is not available, click the "**+**" button to create a new tab.

The following screen will be displayed. From the menu, select **[File]** → **[Rename Python File]**, and rename the file to "**mcp_server_01.py**".



Step 2: Enter the following Python script.

Line 5: This line configures the **MCP server**.

Line 7: By adding the **@mcp.tool decorator**, **calc_molecular_formula** is exposed as a tool through the MCP server.

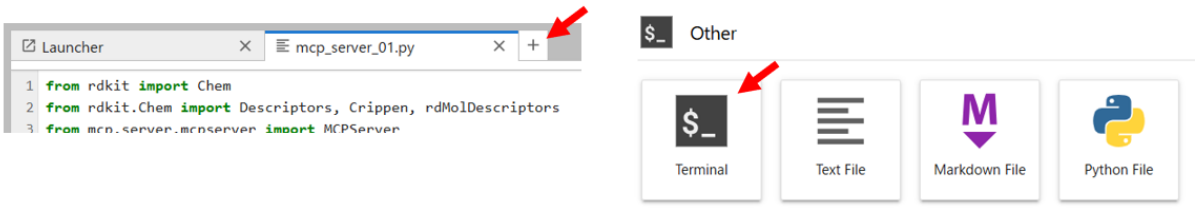
Lines 9 to 26 contain the docstring. This section describes the tool's functionality, its arguments, and its return values. This information is provided to the LLM. Based on the docstring, the LLM determines whether to use the tool and how to execute it.

Line 34: This line starts the MCP server and exposes it on port 8100.

Launcher | mcp_server_01.py

```
1  from rdkit import Chem
2  from rdkit.Chem import Descriptors, Crippen, rdMolDescriptors
3  from mcp.server.mcpserver import MCPServer
4
5  mcp = MCPServer("mcp_server")
6
7  @mcp.tool()
8  def calc_molecular_formula(smiles: str) -> str:
9      """
10     Compute the molecular formula from a SMILES string using RDKit.
11
12     Parameters
13     ----------
14     smiles : str
15         Input SMILES representation of the molecule.
16
17     Returns
18     -------
19     str
20         Molecular formula (e.g., "C2H6O").
21
22     Raises
23     ------
24     ValueError
25         If the SMILES string cannot be parsed by RDKit.
26     """
27     mol = Chem.MolFromSmiles(smiles)
28     if mol is None:
29         raise ValueError(f"Invalid SMILES: {smiles}")
30     return rdMolDescriptors.CalcMolFormula(mol)
31
32
33 if __name__ == "__main__":
34     mcp.run(transport="streamable-http", host="0.0.0.0", port=8100)
```

Step 3: Click the "+" button to open a new Launcher tab, and then select "Terminal".

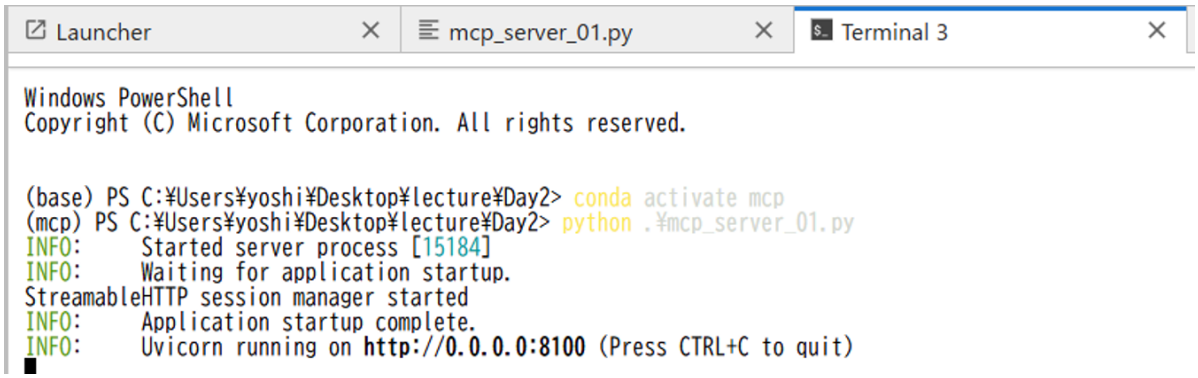


Step 4: Run the following commands in the Terminal:

> conda activate mcp

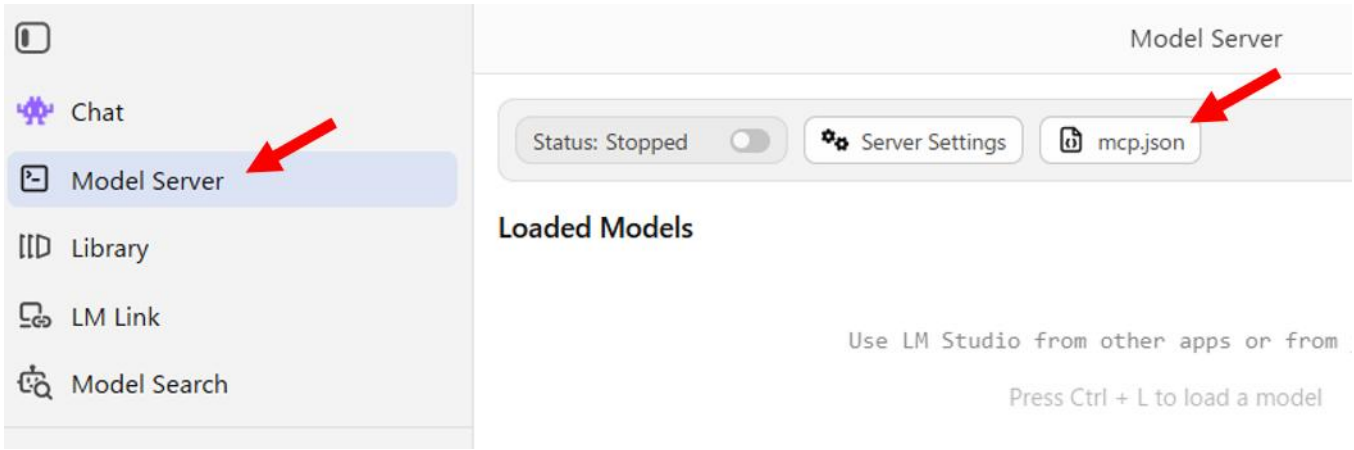
> python mcp_server_01.py

If the Python script runs successfully, the MCP server will start on port 8100.



To stop the server, press "**Ctrl + C**" in the Terminal.

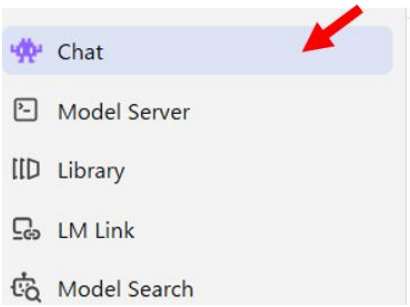
Step 5: The connection between the "**MCP server**" and the LLM is configured in the "**mcp.json**". Click "**Model Server**", then click the "**mcp.json**" button to open the configuration file.

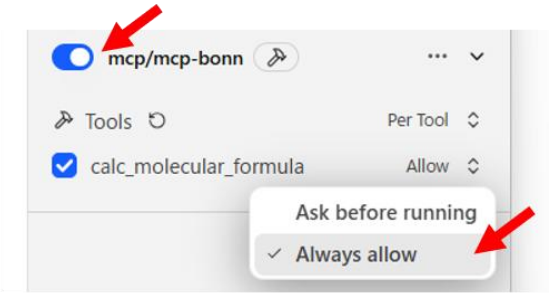


Step 6: Add the MCP server running on port 8010 to the “**mcp.json**” configuration file. Once you have added the MCP server, click "Save".

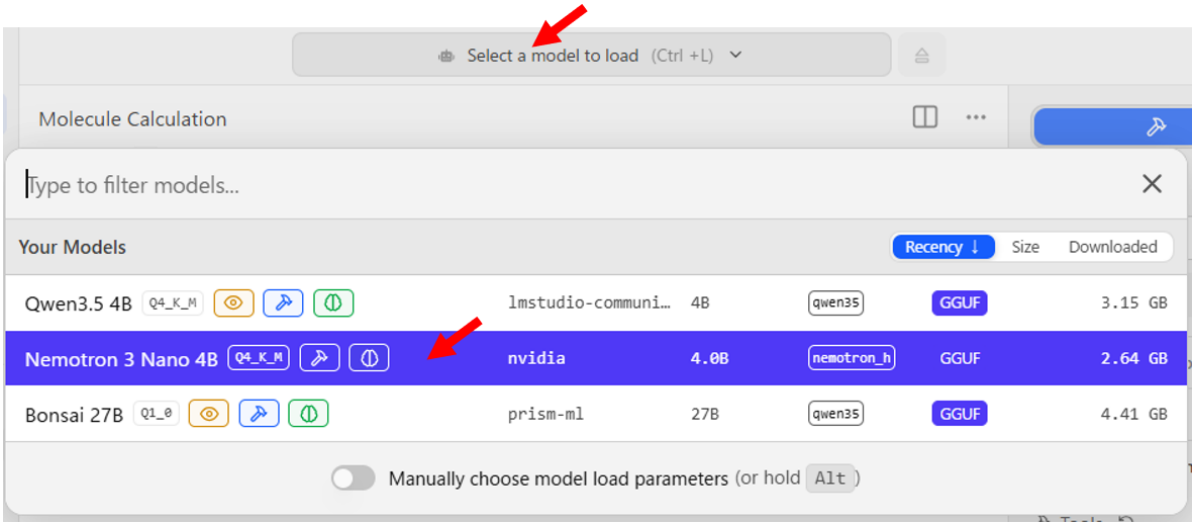


Select the "**Chat**" tab and enable the "**mcp/mcp-bonn**" button. Then set the execution condition for calc_molecular_formula to "**Always allow**".



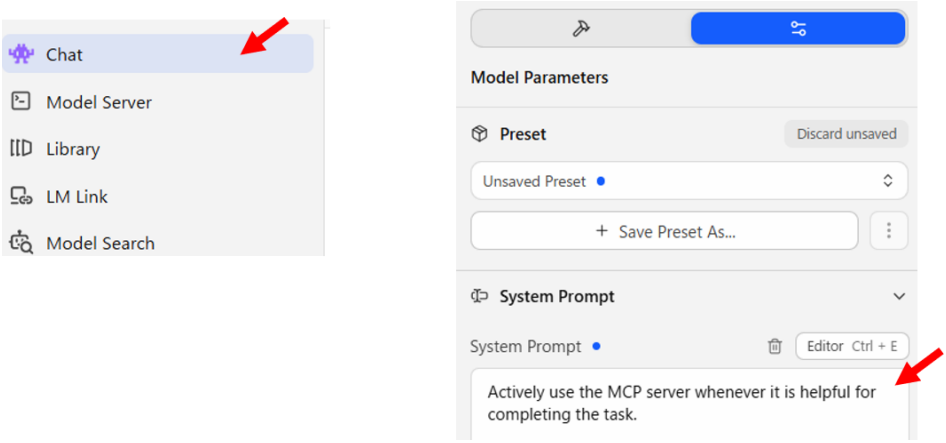


Step 7: Click "**Select a model to load**", then choose "**Nemotron 3 Nano 4B**" from the list of available models.

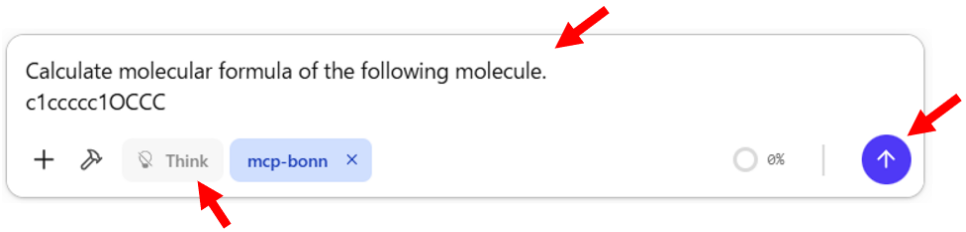


Step 8: Select the "**Chat**" tab and enter the following text in the "**System Prompt**" field.

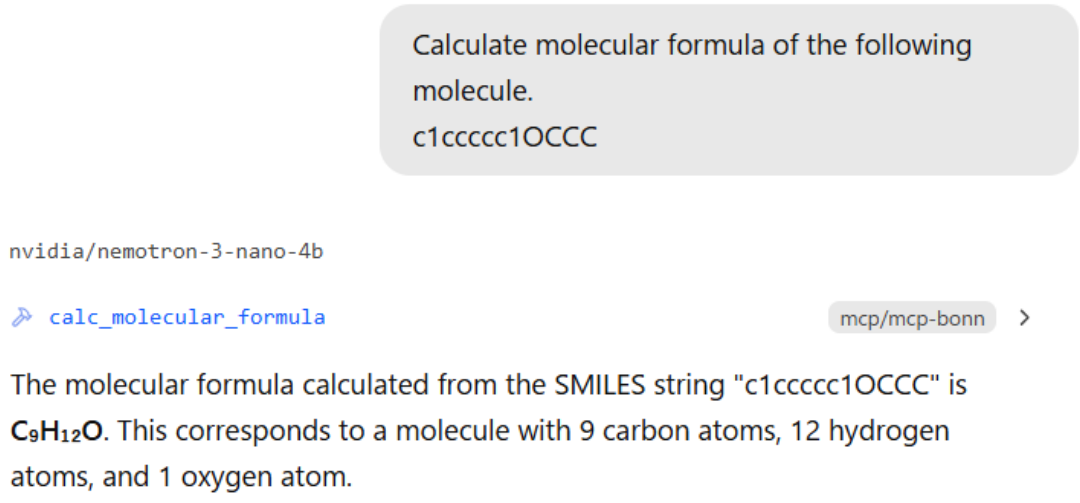
"Actively use the MCP server whenever it is helpful for completing the task."



Step 9: Enter the following **prompt**. Then disable "**Think** Mode" and click the **Run** button to execute it.



As shown below, the AI agent invoked the "**calc_molecular_formula**" tool to calculate the molecular formula.



Tips: Prior to starting the chat, press the button below to clear the context and set it to zero.



The current context usage can be monitored with the indicator below.



Step 10: Test the tool by entering various molecules in SMILES format and reviewing the results.

Exercise 02: Developing an MCP server for molecular properties and integrating it with an LLM.

The following script contains functions for an MCP server that calculate **molecular properties**. Add the function to mcp_server_01.py, then save the modified script as "**mcp_server_02.py**".

```python
@mcp.tool()
def calculate_properties(smiles: str) -> dict[str, float]:
    """
    Calculate molecular properties from a SMILES string.

    Parameters
    ----------
    smiles : str
        Input molecule represented as a SMILES string.

    Returns
    -------
    dict[str, float]
        A dictionary containing the following molecular properties:

        - Mw : Molecular weight
        - logP : Wildman-Crippen LogP
        - TPSA : Topological polar surface area
        - Fsp3 : Fraction of sp3-hybridized carbon atoms

    Raises
    ------
    ValueError
        Raised when the input SMILES string cannot be parsed into a valid molecule.

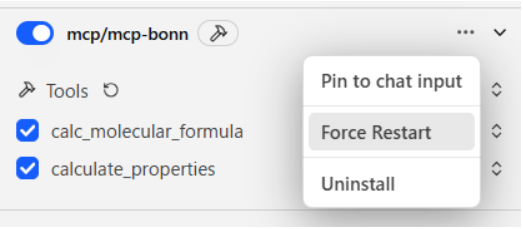
    Examples
    --------
    >>> calculate_properties("CCO")
    {
        "Mw": 46.069,
        "logP": -0.001,
        "TPSA": 20.23,
        "Fsp3": 1.0
    }
    """

    mol = Chem.MolFromSmiles(smiles)

    if mol is None:
        raise ValueError(f"Invalid SMILES: {smiles}")

    return {
        "Mw": Descriptors.MolWt(mol),
        "logP": Crippen.MolLogP(mol),
        "TPSA": rdMolDescriptors.CalcTPSA(mol),
        "Fsp3": rdMolDescriptors.CalcFractionCSP3(mol),
    }
```

Stop “**mcp_server_01.py**” using “**Ctrl + C**”, then run the MCP server using “**mcp_server_02.py**” by following the same procedure as in **Exercise 01**.
By performing a "**Force Restart**" on mcp/mcp-bonn, **calculate_properties** will be recognized and reflected as an MCP server.

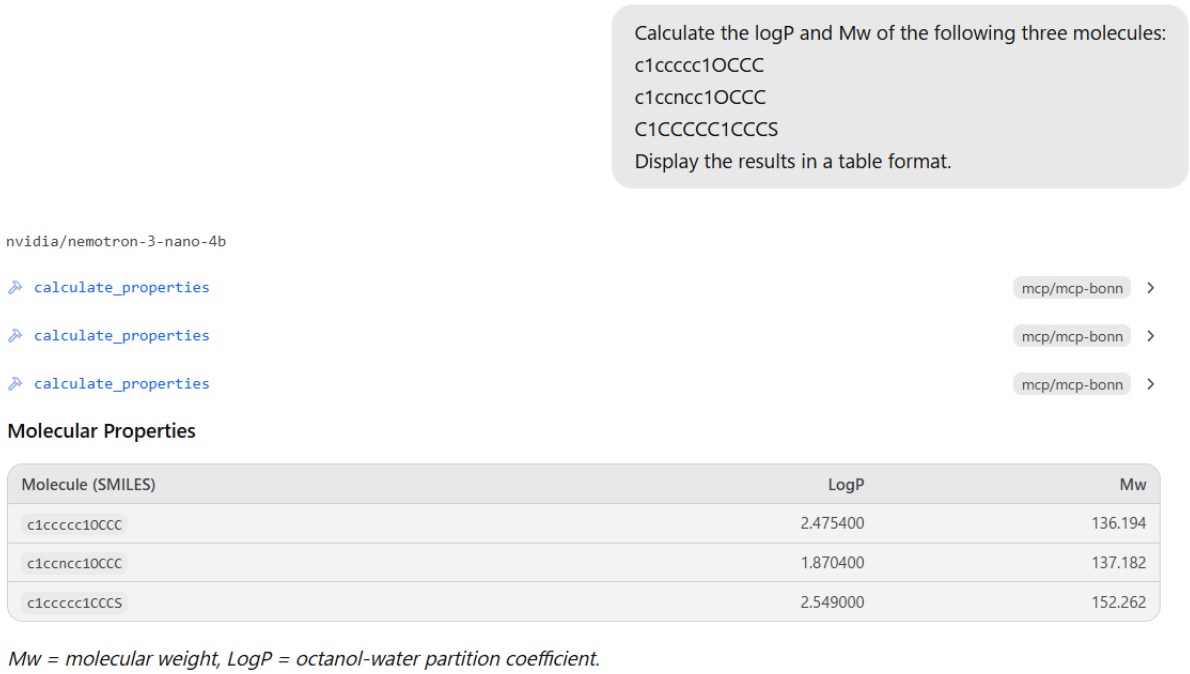


After that, enter the following prompt in LM Studio and check the results.

Prompt:
Calculate the logP and Mw of the following three molecules:
c1ccccc1OCCC
c1ccncc1OCCC
C1CCCCC1CCCS
Display the results in a table format.

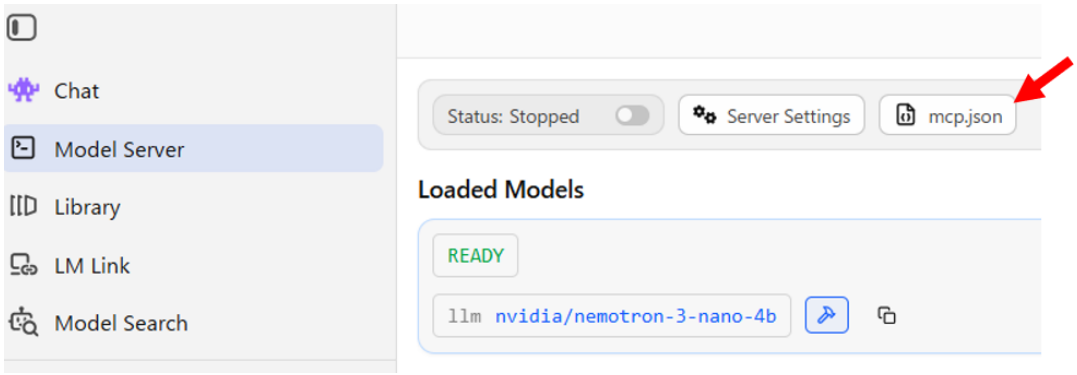
If the response is similar to the example below, the system is functioning correctly.

Response:



Exercise_03: Integrating a ChEMBL MCP server (publicly available) with an LLM and accessing ChEMBL data via an LLM.

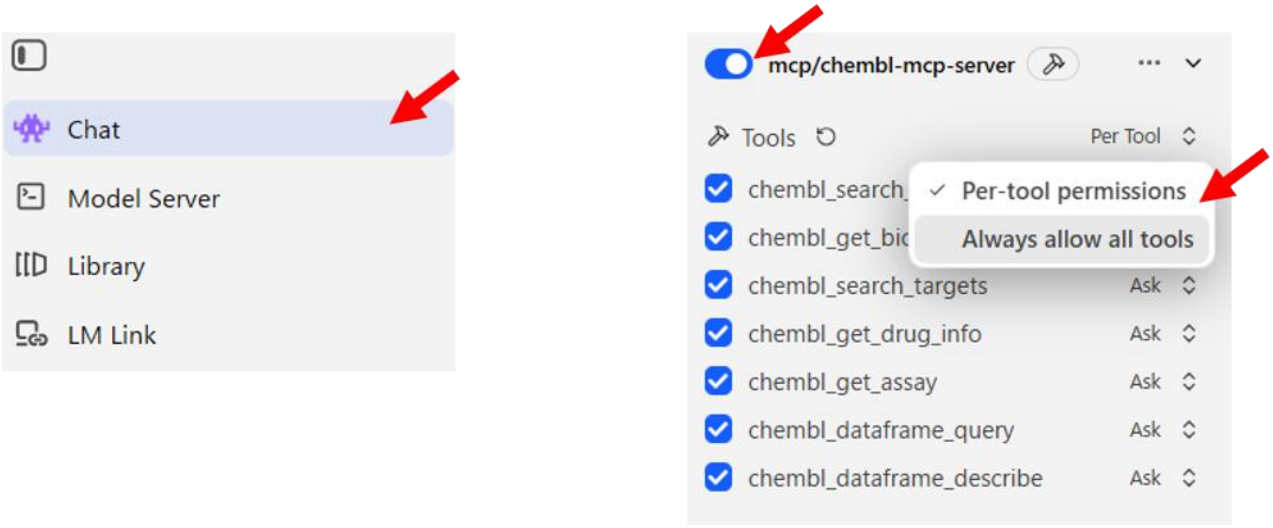
Step 1: To register the “**ChEMBL MCP server**”, select "**Model Server**" and then click the "**mcp.json**" button.



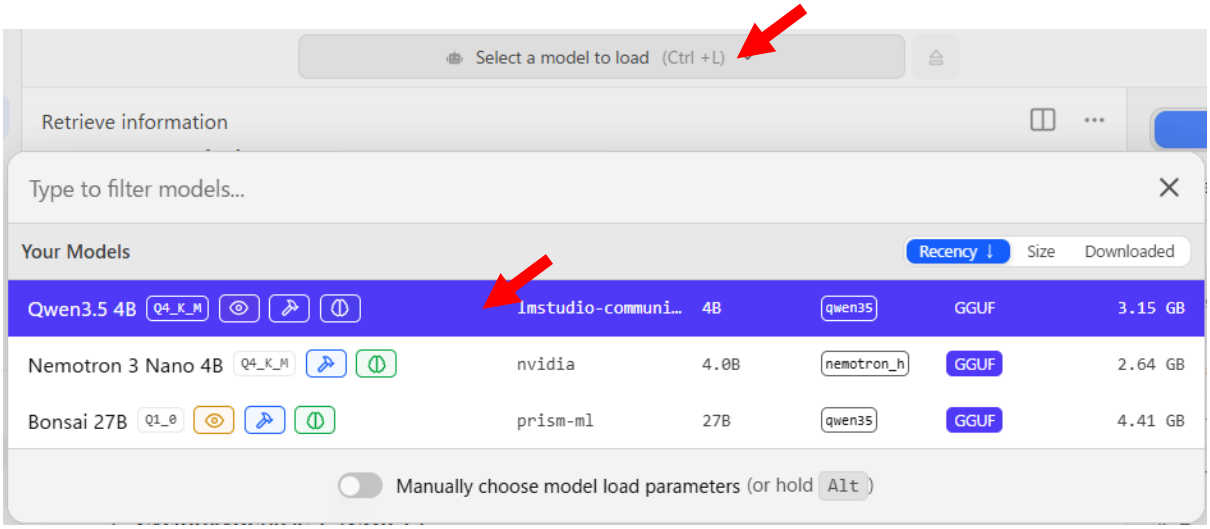
Step 2: Add the “**chembl-mcp-server**” entry to your “**mcp.json**” as shown below, then click the Save button.



Select the "**Chat**" tab and enable the "**mcp/mcp/chembl-mcp-server**" button. Then set the execution condition for calc_molecular_formula to "**Always allow**".



Step 3: Click "**Select a model to load**", then choose "**Qwen3.5 4B**" from the list of available models.



Step 4: Please input each of the following prompts related to **aspirin** and check the results. It is recommended that you reset the context to zero before executing each prompt.
*** If you encounter an error due to an insufficient context length, please refer to **Step 2-3 of the Environment Setup** and increase the **Context Length** setting (for example, to 50,000).

Prompt 1:
Retrieve information on aspirin from ChEMBL.

Prompt 2:
Retrieve the bioassay data for aspirin from the ChEMBL database.
Limit the search results to a maximum of three entries.

Prompt 3:
Retrieve the drug information of aspirin from ChEMBL.
Limit the search results to a maximum of three entries.

Prompt 4:
Retrieve all ChEMBL assay records for aspirin with a confidence score of 9.
Limit the search results to a maximum of three entries.

Step 5: Please use the **chembl-mcp-server** to look up information on **ibuprofen**.