Use LM Studio to build automatic tool calling with Granite

In this step-by-step tutorial, you will use LM Studio with the open source IBM® Granite® 3.3-8b instruct modelon your local machine. Initially, you will test the local model as is and then write Python functions the model can use for automatic tool calling. Finally, you will develop more functions to play a game of chess with an AI agent. This tutorial can also be found in the IBM Granite Community's Granite Snack Cookbook GitHub in the form of a Jupyter Notebook.

LM Studio

LM Studio is an application for working with local large language models (LLMs). You can use any open source model with LM Studio, such as Mistral AI models, Google's Gemma, Meta's Llama or DeepSeek's R1 series.

Using LM Studio, beginner to more advanced users can run LLMs with either their computer's CPU or even a GPU. LM Studio offers a chat-like interface to interact with local LLMs similar to ChatGPT's chat.

With a local AI model, you can fine-tune, inference and more without having to worry about external API calls (like OpenAI or IBM watsonx.ai application programming interfaces, or APIs) or token usage. LM Studio also lets users locally and privately "chat with documents." A user can attach a document to a chat session and ask questions about the document. In cases where the document is long, LM Studio will set up a retrieval augmented generation (RAG) system for querying.

Tool calling

While LLMs excel at understanding and generating human-like text, they often face limitations when tasks require precise computation, access to real-time external data or the execution of specific, well-defined procedures. By implementing tool calling, we equip LLMs with a set of "tools"—external functions—that they can choose to call that can significantly extend their capabilities. This tutorial will demonstrate how to define these tools and integrate them, enabling the LLM to perform a wider range of tasks with greater reliability.

Steps

Step 1. Install LM Studio

Before installing LM Studio, check that your local machine meets the minimum system requirements.

Next, download the appropriate installer for your computer's operating system (Windows, macOS or Linux®). Then, follow these instructions to download the models to your local machine.

We will be using the Granite 3.3-8b instruct model for this recipe, but feel free to use any LLM of your choice. If you're using the Granite model, you can search for the specific user/model stringibm-granite/granite-3.3-8b-instruct-GGUF in theSelect a model to load space in LM Studio.

Next, start the LM Studio local server by navigating to the greenDeveloper icon in the upper left of LM Studio. Toggle theStatus bar on the upper left toRunning
.

Step 2. Install dependencies

We first need to install the necessary libraries, including the LM Studio SDK and chess library.

```
%pip install git+https://github.com/ibm-granite-community/utils \
    lmstudio \
    chess
import lmstudio as lms
```

Step 3. Load the model

We'll specify the model we want to use in this recipe next. In our case, it will be the model we downloaded in LM Studio, Granite 3.3-8b instruct.

We'll also start chatting with it by calling `model.respond()` with an initial message.

```
model = lms.llm("ibm-granite/granite-3.3-8b-instruct-GGUF")

print(model.respond("Hello Granite!"))
```

Step 4. Perform a calculation without tools

Let's start by asking the model to do a straightforward calculation.

```
print(model.respond("What is 26.97 divided by 6.28? Don't round."))
```

While the model might be able to provide a close approximation, it won't return the exact answer because it can't calculate the quotient on its own.

Step 5. Create calculation tools with Python functions

To solve this problem, we will provide the model with tools. Tools are Python functions that we provide to the model at inference. The model can choose to call one or more of these tools to answer the user's query.

Review the LM Studio Docs for more information on how to write tools. In general, you should make sure that your tooling functions have an appropriate name, defined input and output types and a description that explains the purpose of the tool. All this information is passed to the model and can help it select the correct tool to answer your query.

We will write several simple math functions for the model to use as tools:

```
def add(a: float, b:float):
    """Given two numbers a and b, return a + b."""
    return a + b

def subtract(a: float, b:float):
    """Given two numbers a and b, return a - b."""
    return a - b

def multiply(a: float, b: float):
    """Given two numbers a and b, return a * b."""
    return a * b

def divide(a: float, b: float):
    """Given two numbers a and b, return a / b."""
    return a / b

def exp(a: float, b:float):
    """Given two numbers a and b, return a^b"""
    return a ** b
```

Now, we can rerun the same query but provide the model some tools to help it answer. We'll use the `model.act()` call for automatic tool calling and indicate to the model that it can use the functions we created.

```
model.act(
  "What is 26.97 divided by 6.28? Don't round.",
  [add, subtract, multiply, divide, exp],
  on_message=print,
)
```

We can see that the model was able to select the correct tool under `ToolCallRequest`, `name`, used the appropriate inputs under `arguments` (the arguments to pass to the function) and avoided using the irrelevant tools. Finally, the response under `AssistantResponse`, `content`, `text` shows the response from the model, an exact answer to the question.

How many Rs in strawberry?

A very simple question that stumps even the smartest language models. Almost every single LLM with a training cutoff before 2024 answers that there are only 2 Rs in the word "strawberry." As a bonus, it might even hallucinate incorrect positions for the letters.

Nowadays, LLMs tend to get this specific question right, purely because its virality landed it in most training datasets. However, LLMs still commonly fail on similar letter counting tasks.

```
print(model.respond("How many Bs are in the word 'blackberry'?"))
```

Let's write a tool to help the model do a better job.

```
def get_letter_frequency(word: str) -> dict:
    """Takes in a word (string) and returns a dictionary containing the counts of each letter that appears in the word. """

    letter_frequencies = {}

    for letter in word:
        if letter in letter_frequencies:
            letter_frequencies[letter] += 1
        else:
            letter_frequencies[letter] = 1

    return letter_frequencies
```

Now we can pass the tool to the model and rerun the prompt.

```
model.act(
  "How many Bs are in the word 'blackberry'?",
  [get_letter_frequency],
  on_message=print,
)
```

Using the `get_letter_frequency()` tool, the model was able to accurately count the number of b's in the word 'blackberry'.

Step 6. Implement automatic tool calling for an agent

One of the best use-cases of this automatic tool-calling workflow is to give your model the ability to interact with its external environment. Let's build an agent that uses tools to play chess!

While language models can have strong conceptual knowledge of chess, they aren't inherently designed to understand a chess board. If you try to play a game of chess with an online chatbot, it will often derail after several turns, making illegal or irrational moves.

We are providing the model several tools that help it understand and interact with the board.

- **legal_moves()**: provides a list of all legal moves in the current position
- **possible_captures()**: provides a list of all possible captures in the current position
- **possible_checks()**: provides a list of all possible checks in the current position
- **get_move_history()**: provides a list of all moves played so far
- **get_book_moves()**: provides a list of all book moves
- **make_ai_move()**: an interface to let the model input its move

It's not a lot, but it is enough for the model to play a full game of chess without hallucinating, and use some intelligent reasoning to base its decisions.

```
import chess
import chess.polyglot
from IPython.display import display, SVG, clear_output
import random
import os, requests, shutil, pathlib

board = chess.Board()
ai_pos = 0

# Download book moves
RAW_URL   = ("https://raw.githubusercontent.com/"
             "niklasf/python-chess/master/data/polyglot/performance.bin")
DEST_FILE = "performance.bin"

if not os.path.exists(DEST_FILE):
    print("Downloading performance.bin …")
    try:
        with requests.get(RAW_URL, stream=True, timeout=15) as r:
            r.raise_for_status()
            with open(DEST_FILE, "wb") as out:
                shutil.copyfileobj(r.raw, out, 1 << 16)  # 64 KB chunks
    except requests.exceptions.RequestException as e:
        raise RuntimeError(f"Download failed: {e}")



def legal_moves() -> list[str]:
    """
    Returns a list of legal moves in standard algebraic notation.
    """
    return [board.san(move) for move in board.legal_moves]

def possible_captures() -> list[dict]:
    """
    Returns all legal captures with metadata:
    - san: SAN notation of the capture move.
    - captured_piece: The piece type being captured ('P','N','B','R','Q','K').
    - is_hanging: True if the captured piece was undefended before the capture.
    """
    result = []
    for move in board.generate_legal_captures():
        piece = board.piece_at(move.to_square)
        piece_type = piece.symbol().upper() if piece else "?"
        # Check defenders of the target square
        defenders = board.attackers(not board.turn, move.to_square)
        is_hanging = len(defenders) == 0  # no defenders => hanging

        result.append({
            "san": board.san(move),
            "captured_piece": piece_type,
            "is_hanging": is_hanging
        })
    return result

def possible_checks() -> list[dict]:
    """
    Returns all legal checking moves with metadata:
    - san: SAN notation of the checking move.
    - can_be_captured: True if after the move, the checking piece can be captured.
    - can_be_blocked: True if the check can be legally blocked.
    - can_escape_by_moving_king: True if the king can move out of check.
    """
    result = []
    for move in board.legal_moves:
        if not board.gives_check(move):
            continue
        temp = board.copy()
        temp.push(move)

        can_capture = any(
            temp.is_capture(reply) and reply.to_square == move.to_square
            for reply in temp.legal_moves
        )

        # King escapes by moving
        king_sq = temp.king(not board.turn)
        can_escape = any(
            reply.from_square == king_sq for reply in temp.legal_moves
        )

        # Blocking: legal non-capture, non-king move that resolves check
        can_block = any(
            not temp.is_capture(reply)
            and reply.from_square != king_sq
            and not temp.gives_check(reply)
            for reply in temp.legal_moves
        )

        result.append({
            "san": board.san(move),
            "can_be_captured": can_capture,
            "can_be_blocked": can_block,
            "can_escape_by_moving_king": can_escape
        })
    return result
```

```
def get_move_history() -> list[str]:
    """
    Returns a list of moves made in the game so far in standard algebraic notation.
    """
    return [board.san(move) for move in board.move_stack]

def get_book_moves() -> list[str]:
    """
    Returns a list of book moves in standard algebraic notation from performance.bin
    for the current board position. If no book moves exist, returns an empty list.
    """
    moves = []
    with chess.polyglot.open_reader("performance.bin") as reader:
        for entry in reader.find_all(board):
            san_move = board.san(entry.move)
            moves.append(san_move)
    return moves

def is_ai_turn() -> bool:
    return bool(board.turn) == (ai_pos == 0)

def make_ai_move(move: str) -> None:
    """
    Given a string representing a valid move in chess notation, pushes move onto chess board.
    If non-valid move, raises a ValueError with message "Illegal move.
    If called when it is not the AI's turn, raises a ValueError with message "Not AI's turn."
    THIS FUNCTION DIRECTLY ENABLES THE AI TO MAKE A MOVE ON THE CHESS BOARD.
    """
    if is_ai_turn():
        try:
            board.push_san(move)
        except ValueError as e:
            raise ValueError(e)
    else:
        raise ValueError("Not AI's turn.")

def make_user_move(move: str) -> None:
    """
    Given a string representing a valid move in chess notation, pushes move onto chess board.
    If non-valid move, raises a ValueError with message "Illegal move.
    If called when it is not the player's turn, raises a ValueError with message "Not player's turn."
    If valid-move, updates the board and displays the current state of the board.
    """
    if not is_ai_turn():
        try:
            board.push_san(move)
        except ValueError as e:
            raise ValueError(e)
    else:
        raise ValueError("Not player's turn.")

def print_fragment(fragment, round_index=0):
    print(fragment.content, end="", flush=True)
```

Next, we'll set up for the chess match with an AI agent. By using the `lms.Chat()` call, we'll provide instructions to our chess AI agent for when the agent is playing for white or black.

```
chat_white = lms.Chat("""You are a chess AI, playing for white. Your task is to make the best move in the current position, using the pro
                      You should use your overall chess knowledge, including openings, tactics, and strategies, as your primary method to
                      Use the provided tools as an assistant to improve your understanding of the board state and to make your moves. Alw
                      if they are available. Be prudicious with your checks and captures. Understand whether the capturable piece is hang
                      comparison to the piece you are using to capture. Consider the different ways the opponent can defend a check, to p


chat_black = lms.Chat("""You are a chess AI, playing for black. Your task is to make the best move in the current position, using the pro
                      You should use your overall chess knowledge, including openings, tactics, and strategies, as your primary method to
                      Use the provided tools as an assistant to improve your understanding of the board state and to make your moves. Alw
                      if they are available. Be prudicious with your checks and captures. Understand whether the capturable piece is hang
                      comparison to the piece you are using to capture. Consider the different ways the opponent can defend a check, to p
```

Finally, we'll set up two functions to track the match: `update_board()` and `get_end_state()`.

By using the `model.act()` call we used for tool calling previously, we'll feed the agent instructions (`chat`) we defined, the tools available for its use and establish a `max_prediction_rounds`. This function shows the maximum number of independent tool calls the agent can make to execute a specific move.

After running the next cell, an empty input field should appear for you to write out your moves. If you're unsure of the moves available, type \`help\` and the notations of available moves will be displayed where the first initial is the initialed name of the piece (\"B\" is bishop, \"Q\" is queen, and so on. But \"N\" is knight since \"K\" represents king and no first initial is for a pawn). The next letter and number listed are the row and column to move that piece to. For the notation of special cases like castling or ambiguous piece moves, see the algebraic notation (chess) Wikipedia page.

Good luck!

```
move = 0
import chess.svg

board.reset()
ai_pos = round(random.random())

def update_board(move = move, ai_pos = ai_pos):
    """
    Updates the chess board display in the notebook.
    """
    clear_output(wait=True)  # Clear previous output
    print(f"Board after move {move+1}")
    if (ai_pos == 1):
        display(SVG(chess.svg.board(board, size=400)))
    else:
        display(SVG(chess.svg.board(board, size=400, orientation = chess.BLACK)))

def get_end_state():
    """
    Returns the end state of the chess game.
    """
    if board.is_checkmate():
        return "Checkmate!"
    elif board.is_stalemate():
        return "Stalemate!"
    elif board.is_insufficient_material():
        return "Draw by insufficient material!"
    elif board.is_seventyfive_moves():
        return "Draw by 75-move rule!"
    elif board.is_fivefold_repetition():
        return "Draw by fivefold repetition!"
    else:
        return None

clear_output(wait=True) # Clear any previous output from the cell
if (ai_pos == 1):
    display(SVG(chess.svg.board(board, size=400)))
else:
    display(SVG(chess.svg.board(board, size=400, orientation = chess.BLACK)))

# 2. Loop through moves, apply each move, clear previous output, and display new board
userEndGame = False
while True:

    if ai_pos == 0:
        # AI's turn
        model.act(
            chat_white,
            [get_move_history, legal_moves, possible_captures, possible_checks, get_book_moves, make_ai_move],
            on_message=print,
            max_prediction_rounds = 8,
        )


        if is_ai_turn(): # failsafe in case AI does not make a move
           make_ai_move(legal_moves()[0])  # Default to the first legal move if AI does not respond

        update_board(move)
        move += 1
        game_over_message = get_end_state()
        if game_over_message:
            print(game_over_message)
            break

        # User's turn
        while True:
            user_move = input("User (Playing Black): Input your move. Input 'help' to see the list of possible moves. Input 'quit' to end
            if user_move.lower() == 'quit':
                print("Game ended by user.")
                userEndGame = True
                break
            if user_move.lower() == 'help':
                print("Possible moves:", legal_moves())
                continue
            try:
                make_user_move(user_move)
                break
            except ValueError as e:
                print(e)

        if userEndGame:
            break

        update_board(move)
        move += 1
        game_over_message = get_end_state()
        if game_over_message:
            print(game_over_message)
            break
    else:
        # User's turn
        while True:
            user_move = input("User (Playing White): Input your move. Input 'help' to see the list of possible moves. Input 'quit' to end
            if user_move.lower() == 'quit':
                print("Game ended by user.")
                userEndGame = True
                break
            if user_move.lower() == 'help':
                print("Possible moves:", legal_moves())
                continue
            try:
                make_user_move(user_move)
                break
            except ValueError as e:
                print(e)

        if userEndGame:
            break

        update_board(move)
        move += 1
        game_over_message = get_end_state()
        if game_over_message:
            print(game_over_message)
            break

        model.act(
            chat_black,
            [get_move_history, legal_moves, possible_captures, possible_checks, get_book_moves, make_ai_move],
            max_prediction_rounds = 8,
            on_message=print,
        )

        if is_ai_turn(): # failsafe in case AI does not make a move
           make_ai_move(legal_moves()[0])  # Default to the first legal move if AI does not respond

        update_board(move)
        move += 1
        game_over_message = get_end_state()
        if game_over_message:
            print(game_over_message)
            break
```

Summary

In this notebook, we demonstrated how integrating tools can enhance the utility and agentic capability of LLMs. We illustrated that by providing an LLM with access to predefined external functions, it can transcend its core language processing capabilities to perform tasks like accurate calculations or interface with external systems. It cannot do this reliably on its own. The key takeaway is that tool-use empowers LLMs to delegate specific subproblems to specialized routines, allowing them to ground their responses in factual data or precise operations. This approach not only improves accuracy but also enables LLMs to engage in more complex, interactive workflows, effectively transforming them into more versatile and powerful assistants.