## Developing a Multi-Agent System with Microsoft AutoGen for Text-to-SQL and Accuracy Measurement

Microsoft's AutoGen framework offers a powerful platform for creating sophisticated multi-agent systems capable of tackling complex tasks like converting natural language text into SQL queries (text-to-SQL) and subsequently evaluating their execution accuracy. This approach allows for a modular and often more robust solution, where different agents can specialize in various aspects of the problem, such as understanding the input, generating SQL, refining queries, executing them, and evaluating the results.

Here's a step-by-step guideline to develop such a system:

**Phase 1: Setup and Environment Configuration**

1. **Install AutoGen and Dependencies:**
   * Ensure Python is installed on your system.
   * Install the pyautogen library: pip install pyautogen
   * Install other necessary libraries for LLM integration (e.g., openai if using OpenAI models, or libraries for other LLMs like Llama 3), database connectivity (e.g., psycopg2-binary for PostgreSQL, mysql-connector-python for MySQL, sqlite3 which is built-in), and potentially data handling (e.g., pandas).
   * Consider setting up a virtual environment to manage project dependencies.
2. **Configure LLM Access:**
   * AutoGen requires Large Language Models (LLMs) to power its agents. You'll need to configure access to your chosen LLM(s). This typically involves:
     + Obtaining API keys (e.g., for OpenAI, Azure OpenAI, Groq, or other model providers).
     + Setting up environment variables (e.g., OPENAI\_API\_KEY) or creating a configuration file (e.g., OAI\_CONFIG\_LIST.json or LLM\_CONFIG\_LIST in Python) that AutoGen can use to access the LLM. This file specifies the model name, API key, and potentially other parameters.
3. **Database Setup:**
   * Have a target database ready. This database will contain the schema and data against which the generated SQL queries will be run.
   * Ensure you have the necessary credentials and connection details for this database.

**Phase 2: Designing the Multi-Agent System**

The core idea is to break down the complex text-to-SQL and evaluation task into smaller, manageable sub-tasks, each handled by a specialized agent.

1. **Define Agent Roles:** Consider the following agent roles, which you can customize based on your specific needs:
   * **NaturalLanguageProcessorAgent (or QueryUnderstandingAgent):**
     + **Responsibility:** Receives the raw natural language text. Its primary role is to understand the user's intent, identify key entities, relationships, and desired outcomes. It might perform initial parsing, clarification if the query is ambiguous, and prepare a structured representation of the query for the SQL generation agent.
     + **LLM Configuration:** Needs strong natural language understanding capabilities.
   * **SchemaRetrieverAgent:**
     + **Responsibility:** Interacts with the target database to fetch relevant schema information (table names, column names, data types, primary/foreign keys). This information is crucial for the SQL generation agent to create valid queries. This agent could use Retrieval Augmented Generation (RAG) techniques to provide only the most relevant parts of a large schema.
     + **Tools/Functions:** Needs functions to connect to the database and execute schema introspection queries.
   * **SQLGeneratorAgent:**
     + **Responsibility:** Takes the processed natural language query (from NaturalLanguageProcessorAgent) and the relevant database schema (from SchemaRetrieverAgent) as input. Its core task is to generate the SQL query. This agent will likely be LLM-powered.
     + **LLM Configuration:** Needs strong code generation and SQL-specific knowledge. You might fine-tune a model specifically for text-to-SQL or use a powerful general-purpose model with appropriate prompting.
     + **System Message:** Its system message should clearly define its role, the SQL dialect to be used, and instructions for handling complex constructs like joins, subqueries, aggregations, and window functions.
   * **SQLValidationRefinementAgent (Optional but Recommended):**
     + **Responsibility:** Reviews the SQL query generated by SQLGeneratorAgent. It can check for syntactical correctness against the target database's SQL dialect. It might also employ LLM capabilities to check for semantic correctness or to suggest improvements for clarity or efficiency. If errors are found, it can provide feedback to the SQLGeneratorAgent for iterative refinement.
     + **Tools/Functions:** Could use a SQL parsing library or even attempt to execute the query with a LIMIT 0 or EXPLAIN command to check syntax without fetching data.
   * **SQLExecutorAgent:**
     + **Responsibility:** Takes a validated SQL query and executes it against the target database. It retrieves the results.
     + **Tools/Functions:** Needs robust functions to connect to the database, execute SQL queries, and handle potential execution errors. For safety, especially in initial development, run this in a sandboxed environment or with read-only permissions.
   * **AccuracyMeasurementAgent (or ResultEvaluatorAgent):**
     + **Responsibility:** Compares the results obtained by the SQLExecutorAgent with the expected results (ground truth) for a given natural language query. It then calculates execution accuracy.
     + **Tools/Functions:** Needs functions to compare datasets. This might involve exact matching of results, or more sophisticated comparison if the order of results doesn't matter or if minor data type differences are acceptable.
   * **UserProxyAgent (AutoGen Core Agent):**
     + **Responsibility:** Acts as the primary interface for the user and can orchestrate the interactions between other agents. It can solicit human input when needed and trigger the initial query. It often executes code provided by other agents (like the SQLExecutorAgent).
2. **Define Agent Interactions (Workflow):**
   * **Initiation:** The UserProxyAgent receives the natural language query from the user.
   * **Understanding & Schema Fetching:** The UserProxyAgent passes the query to the NaturalLanguageProcessorAgent. Concurrently or subsequently, the SchemaRetrieverAgent could be triggered to fetch the relevant schema (or this could be a tool used by the SQLGeneratorAgent).
   * **SQL Generation:** The SQLGeneratorAgent receives the processed query and schema information and generates the SQL.
   * **Validation & Refinement (Optional Loop):** The generated SQL is passed to the SQLValidationRefinementAgent. If issues are found, feedback is provided to the SQLGeneratorAgent to regenerate the query. This can be an iterative loop.
   * **SQL Execution:** Once the SQL is deemed valid, the UserProxyAgent (often configured to execute code) or a dedicated SQLExecutorAgent runs the query against the database.
   * **Accuracy Measurement:** The AccuracyMeasurementAgent takes the executed results and compares them against a predefined "gold standard" or expected output for that input query.
   * **Reporting:** The final results, including the generated SQL, the execution output, and the accuracy score, are presented back to the user via the UserProxyAgent.

**Phase 3: Implementing the Agents and Tools**

1. **Initialize Agents:**
   * In your Python script, import the necessary AutoGen classes (e.g., AssistantAgent, UserProxyAgent, ConversableAgent).
   * Instantiate each agent, providing a name, a system message defining its role and instructions, and the LLM configuration (llm\_config).
2. **Develop Custom Tools/Functions:**
   * Agents often need to perform actions beyond just text generation (e.g., connecting to a database, executing code, fetching schema). AutoGen allows you to register custom Python functions as "tools" that agents can use.
   * **For SchemaRetrieverAgent:** Create functions to connect to your database and query information\_schema or equivalent system tables to get table names, column names, types, and relationships.
   * **For SQLExecutorAgent (or UserProxyAgent if it executes SQL):**
     + Create a function that takes a SQL query string as input.
     + This function should establish a database connection, execute the query, fetch the results (e.g., into a pandas DataFrame or a list of tuples), and handle any exceptions.
     + **Security Note:** Be extremely cautious with directly executing LLM-generated SQL. Sanitize inputs, use read-only database users where possible, and consider execution in a sandboxed environment to prevent SQL injection or unintended data modification.
   * **For AccuracyMeasurementAgent:** Create functions to compare the actual query results with the expected results.
3. **Register Tools with Agents:**
   * Use the register\_function method on the UserProxyAgent (or an AssistantAgent that needs to use tools) to make your custom Python functions available to the agents. You'll provide a mapping between a function name (that the LLM can call) and the actual Python function.

**Phase 4: Orchestrating Agent Conversations**

1. **Initiate Chat:**
   * Use the initiate\_chat() method on the UserProxyAgent to start the conversation. You'll specify the first recipient agent (e.g., the NaturalLanguageProcessorAgent or directly the SQLGeneratorAgent if the initial processing is simple) and the initial message (the natural language query).
2. **Manage Conversation Flow:**
   * The agents will converse based on their system messages, LLM responses, and the use of registered tools.
   * You can guide the conversation flow by how you structure the initiate\_chat call and the system\_message of each agent, encouraging them to call specific functions or pass information to other agents.
   * For complex workflows, you might use GroupChat for more managed multi-agent interactions.

**Phase 5: Measuring Execution Accuracy**

This is a critical step to evaluate the performance of your text-to-SQL system.

1. **Prepare a "Gold Standard" Evaluation Dataset:**
   * This dataset should consist of pairs of:
     + Natural language questions (covering a range of complexities, including those with joins, aggregations, subqueries, etc., relevant to your use case).
     + The corresponding "gold" SQL query that correctly answers the question for your specific database schema.
     + The expected execution results of the "gold" SQL query.
   * Benchmark datasets like Spider, BIRD, or custom datasets tailored to your domain can be invaluable here.
2. **Define Accuracy Metrics:**
   * **Execution Accuracy (EA):** This is the primary metric you're aiming for.
     + The generated SQL query is executed against the database.
     + The *results* of this execution are compared to the *results* of executing the gold standard SQL query.
     + If the results match, the query is considered "execution accurate."
     + Challenges: Defining "match." Does the order of rows matter? Are minor floating-point differences acceptable?
   * **Exact Match (EM) of SQL Query:**
     + The generated SQL query string is compared to the gold standard SQL query string.
     + This is a stricter metric. Even a semantically equivalent query with different formatting or aliasing will fail. Normalization (e.g., lowercasing keywords, standardizing spacing) can make this slightly more lenient but it still doesn't guarantee functional equivalence if there are multiple correct SQL ways to phrase a query.
   * **Component Matching:** Partial credit can be given if certain components of the SQL query (e.g., correct SELECT columns, correct FROM tables, correct WHERE conditions) are present, even if the full query isn't perfect.
3. **Implement the Accuracy Calculation in AccuracyMeasurementAgent:**
   * After the SQLExecutorAgent runs the generated SQL, its results are passed to the AccuracyMeasurementAgent.
   * This agent retrieves the corresponding "gold standard" results for the input natural language query.
   * It then performs the comparison:
     + For simple results (e.g., a single number), comparison is straightforward.
     + For tabular results, you might convert both actual and expected results to canonical forms (e.g., sorted pandas DataFrames) before comparison to handle row order differences.
     + Consider data type consistency.
4. **Iterate and Improve:**
   * Run your system against the evaluation dataset.
   * Analyze the failures:
     + Were the natural language understanding incorrect?
     + Did the agent miss crucial schema information?
     + Was the SQL generation flawed (syntax errors, logical errors)?
     + Were there issues with SQL execution?
   * Refine your agent prompts (system messages), the tools they use, the LLM configurations, or even the agent roles and interaction flow based on these analyses.
   * Prompt engineering, few-shot learning (providing examples in prompts), and potentially fine-tuning your LLMs on domain-specific text-to-SQL examples can significantly improve accuracy.

**Phase 6: Handling Complex SQL Queries**

* **Schema Representation:** For complex databases, providing the entire schema in every prompt is inefficient and can confuse the LLM. The SchemaRetrieverAgent should be intelligent, possibly using RAG techniques to select only the tables and columns most relevant to the user's query.
* **Multi-Turn Clarification:** For ambiguous or highly complex queries, design the NaturalLanguageProcessorAgent or the UserProxyAgent to ask clarifying questions.
* **Decomposition:** Encourage agents (especially the SQLGeneratorAgent through its system message) to break down complex queries into simpler parts or use Common Table Expressions (CTEs) for better readability and manageability.
* **Iterative Refinement:** The loop between SQLGeneratorAgent and SQLValidationRefinementAgent is crucial. The validation agent can catch errors in complex queries and provide specific feedback (e.g., "Table X does not have column Y," or "Ambiguous join condition").
* **Few-Shot Prompting:** Include examples of complex text-to-SQL conversions in the prompts for the SQLGeneratorAgent to guide it on how to handle constructs like nested queries, window functions, and complex joins.

**Example Snippet (Conceptual):**

import autogen  
from autogen import UserProxyAgent, AssistantAgent  
import sqlite3 # Example database  
  
# --- 0. Configuration (replace with your LLM config) ---  
config\_list = autogen.config\_list\_from\_json(  
 "OAI\_CONFIG\_LIST",  
 filter\_dict={"model": ["gpt-4o"]} # or your preferred model  
)  
llm\_config = {"config\_list": config\_list, "cache\_seed": 42}  
  
# --- 1. Database Setup & Tools ---  
DB\_PATH = "my\_database.db" # Replace with your database path  
  
def get\_schema\_info(\_query\_placeholder):  
 """Retrieves basic schema information (table names and their columns)."""  
 conn = sqlite3.connect(DB\_PATH)  
 cursor = conn.cursor()  
 cursor.execute("SELECT name FROM sqlite\_master WHERE type='table';")  
 tables = [row[0] for row in cursor.fetchall()]  
 schema\_info = {}  
 for table in tables:  
 cursor.execute(f"PRAGMA table\_info({table});")  
 columns = [row[1] for row in cursor.fetchall()]  
 schema\_info[table] = columns  
 conn.close()  
 return f"Database Schema:\n{schema\_info}"  
  
  
def execute\_sql\_query(sql\_query: str):  
 """Executes a SQL query and returns the results or an error message."""  
 print(f"\nAttempting to execute SQL: {sql\_query}\n")  
 try:  
 conn = sqlite3.connect(DB\_PATH)  
 cursor = conn.cursor()  
 cursor.execute(sql\_query)  
 if sql\_query.strip().upper().startswith("SELECT"):  
 results = cursor.fetchall()  
 # For complex results, you might want column headers too  
 # col\_names = [description[0] for description in cursor.description]  
 # results\_with\_headers = [dict(zip(col\_names, row)) for row in results]  
 else: # For INSERT, UPDATE, DELETE  
 conn.commit()  
 results = f"Query executed successfully. Rows affected: {cursor.rowcount}"  
 conn.close()  
 return results  
 except Exception as e:  
 return f"Error executing SQL: {str(e)}"  
  
# --- Gold Standard (Simplified Example) ---  
gold\_standards = {  
 "How many users are there?": {  
 "gold\_sql": "SELECT COUNT(\*) FROM users;",  
 "expected\_results": [(5,)] # Assuming 5 users for this example  
 },  
 "List all products.": {  
 "gold\_sql": "SELECT name FROM products;",  
 "expected\_results": [("Laptop",), ("Mouse",), ("Keyboard",)] # Example  
 }  
 # Add more complex examples  
}  
  
def measure\_accuracy(natural\_language\_query: str, actual\_results\_str: str):  
 """  
 Compares actual results with expected results.  
 Note: `actual\_results\_str` would typically be the direct output from `execute\_sql\_query`.  
 This function needs to parse `actual\_results\_str` appropriately if it's a string representation  
 of a list of tuples or other complex structure. For simplicity, we'll assume  
 `execute\_sql\_query` returns results in a comparable format or this function handles the conversion.  
 """  
 print(f"\n--- Measuring Accuracy for: {natural\_language\_query} ---")  
 print(f"Actual Results (string form): {actual\_results\_str}")  
  
 if natural\_language\_query not in gold\_standards:  
 return "No gold standard found for this query."  
  
 expected\_results = gold\_standards[natural\_language\_query]["expected\_results"]  
  
 # This is a very basic comparison. You'll need more robust parsing and comparison  
 # for real-world scenarios.  
 try:  
 # Attempt to evaluate the string representation of the results  
 # THIS IS RISKY IF THE LLM OUTPUTS ARBITRARY CODE.  
 # In a real system, the SQLExecutorAgent would return structured data, not just a string.  
 import ast  
 actual\_results\_eval = ast.literal\_eval(actual\_results\_str)  
 if isinstance(actual\_results\_eval, list) and all(isinstance(item, tuple) for item in actual\_results\_eval):  
 # Simple comparison for list of tuples  
 if sorted(actual\_results\_eval) == sorted(expected\_results): # Sort to handle order indifference  
 accuracy = "Execution Accurate: Results Match Gold Standard."  
 else:  
 accuracy = f"Execution Inaccurate: Results Mismatch. Expected: {expected\_results}, Got: {actual\_results\_eval}"  
 elif actual\_results\_eval == expected\_results: # For single value results  
 accuracy = "Execution Accurate: Results Match Gold Standard."  
 else:  
 # Handle cases where parsing as list of tuples fails or structure is different  
 # This part needs significant improvement for robustness  
 if str(actual\_results\_str) == str(expected\_results): # Fallback to string comparison (less reliable)  
 accuracy = "Execution Accurate (string match): Results Match Gold Standard."  
 else:  
 accuracy = f"Execution Inaccurate: Results Mismatch (or unable to directly compare complex structures). Expected: {expected\_results}, Got (string): {actual\_results\_str}"  
  
 except Exception as e:  
 accuracy = f"Error in accuracy measurement: {str(e)}. Could not parse actual results: {actual\_results\_str}"  
  
 print(accuracy)  
 return accuracy  
  
  
# --- 2. Define Agents ---  
user\_proxy = UserProxyAgent(  
 name="UserProxy",  
 human\_input\_mode="TERMINATE", # Get human input once at the end, or "ALWAYS" for interactive  
 max\_consecutive\_auto\_reply=10,  
 is\_termination\_msg=lambda x: x.get("content", "").rstrip().endswith("TERMINATE"),  
 code\_execution\_config={"work\_dir": "coding\_dir", "use\_docker": False}, # Set use\_docker=True if Docker is installed  
 system\_message="You are a user proxy. Reply TERMINATE when the task is done."  
)  
  
schema\_agent = AssistantAgent(  
 name="SchemaExpert",  
 llm\_config=llm\_config,  
 system\_message="You are a schema expert. You can use the 'get\_schema\_info' tool to fetch database schema. Respond with only the schema information or an error if you cannot retrieve it."  
)  
  
sql\_generator = AssistantAgent(  
 name="SQLWriter",  
 llm\_config=llm\_config,  
 system\_message="You are an expert SQL writer. Given a natural language query and the database schema, write an accurate SQL query for SQLite. "  
 "You MUST NOT execute the SQL. Only generate the SQL query text. "  
 "Prioritize clarity and correctness. For complex queries, consider using CTEs. "  
 "Always ensure your SQL query is a single, executable statement. "  
 "After providing the SQL, say 'SQL\_GENERATED'. Do not add any other commentary after 'SQL\_GENERATED'."  
)  
  
# The UserProxyAgent will execute the SQL and measure accuracy for this example  
# by registering the functions directly to it.  
# In a more complex setup, SQLExecutorAgent and AccuracyMeasurementAgent could be separate.  
  
user\_proxy.register\_function(  
 function\_map={  
 "get\_db\_schema": get\_schema\_info,  
 "execute\_sql\_query": execute\_sql\_query,  
 "measure\_accuracy": measure\_accuracy  
 }  
)  
# Also allow SQLWriter to request schema (though in a group chat, UserProxy might orchestrate this)  
# sql\_generator.register\_function(function\_map={"get\_db\_schema": get\_schema\_info}) # Less ideal, better to pass schema  
  
# --- 3. Create a dummy database for testing ---  
def create\_dummy\_db():  
 conn = sqlite3.connect(DB\_PATH)  
 cursor = conn.cursor()  
 cursor.execute("DROP TABLE IF EXISTS users;")  
 cursor.execute("DROP TABLE IF EXISTS products;")  
 cursor.execute("CREATE TABLE users (id INTEGER PRIMARY KEY, name TEXT, email TEXT);")  
 cursor.execute("CREATE TABLE products (id INTEGER PRIMARY KEY, name TEXT, price REAL);")  
 users\_data = [  
 (1, "Alice", "alice@example.com"), (2, "Bob", "bob@example.com"),  
 (3, "Charlie", "charlie@example.com"), (4, "David", "david@example.com"),  
 (5, "Eve", "eve@example.com")  
 ]  
 products\_data = [  
 (1, "Laptop", 1200.00), (2, "Mouse", 25.00), (3, "Keyboard", 75.00)  
 ]  
 cursor.executemany("INSERT INTO users VALUES (?,?,?);", users\_data)  
 cursor.executemany("INSERT INTO products VALUES (?,?,?);", products\_data)  
 conn.commit()  
 conn.close()  
 print("Dummy database created/reset.")  
  
create\_dummy\_db()  
  
# --- 4. Orchestrate with GroupChat (More robust for multi-step processes) ---  
# For a more controlled flow, especially getting schema first.  
  
# Instead of direct initiate\_chat for complex flows, you might define a sequence  
# or use a GroupChat and manager. For simplicity here, we'll try a direct approach  
# and rely on the UserProxyAgent to call the tools.  
  
natural\_language\_query = "How many users are there?"  
# natural\_language\_query = "List all products."  
  
# A more robust approach would be to use a GroupChat or a sequence of `initiate\_chat` calls  
# or have agents explicitly call tools.  
# For this example, we construct a detailed prompt for the UserProxy to guide the process.  
  
user\_proxy.initiate\_chat(  
 sql\_generator, # Start with the SQL generator  
 message=f"""  
 User's question: "{natural\_language\_query}"  
  
 Here's the plan:  
 1. You, SQLWriter, need the database schema to write an accurate query.  
 The UserProxy has a tool `get\_db\_schema` (call it with any placeholder like "describe schema").  
 Wait for the UserProxy to provide the schema.  
 2. Once you receive the schema from UserProxy, generate the SQLite SQL query for the user's question.  
 Ensure your response ONLY contains the SQL query followed by 'SQL\_GENERATED'.  
 3. UserProxy, after SQLWriter provides the SQL (ending in 'SQL\_GENERATED'):  
 a. Extract the SQL query.  
 b. Execute it using your `execute\_sql\_query` tool.  
 c. Take the result of the execution and the original natural language query ("{natural\_language\_query}")  
 and call your `measure\_accuracy` tool.  
 d. Report the final accuracy result and TERMINATE.  
  
 SQLWriter, please first request the schema from UserProxy by suggesting a call to `get\_db\_schema`.  
 Example of how to suggest UserProxy call a tool if you are SQLWriter:  
 "I need the schema. UserProxy, please call `get\_db\_schema` with the argument 'all\_tables'."  
 (Actually, for `get\_db\_schema` as defined, no specific argument is needed, but the LLM needs to make a function call.)  
  
 Let's start. SQLWriter, ask for the schema for the query: "{natural\_language\_query}"  
 """  
)  
  
# Note: The above example is a simplified illustration.  
# Robust error handling, more sophisticated agent communication (e.g., using GroupChat),  
# and advanced accuracy measurement techniques would be needed for a production system.  
# The direct prompting of UserProxy to follow a plan is a way to guide simple sequences;  
# AutoGen's strength is in agents deciding the next steps based on their config and conversation.  
# A GroupChat with a dedicated manager agent often leads to more robust and flexible flows.

By following these steps, you can leverage Microsoft AutoGen to build a powerful multi-agent system for parsing complex SQL queries from text and rigorously measuring their execution accuracy, leading to more reliable and intelligent data interaction capabilities. Remember that iterative development and thorough testing against a comprehensive evaluation dataset are key to success.