TECHNICAL DOCUMENTATION - DataWarrior-Agent

1. Table of Contents

2. Current Features

DataWarrior-Agent is a platform that allows controlling the cheminformatics software DataWarrior through a conversational interface powered by large language models (LLMs). The available features are as follows:

2.1. File Operations

- Opening molecular files (SDF, MOL, CSV with SMILES, TXT)
- Exporting data in CSV format
- Automatic snapshot saving for review

2.2. Molecular Descriptor Calculation

- Calculation of all 59 molecular descriptors available in DataWarrior
- Selective calculation of specific descriptors from the following categories:
  - Molecular weight (totalWeight, fragmentWeight, fragmentAbsWeight)
  - Drug-likeness (logP, logS, acceptors, donors, sasa, rpsa, tpsa, druglikeness)
  - Toxicity (mutagenic, tumorigenic, reproEffective, irritant, nasty, pains)
  - Structure (shape, flexibility, complexity, fragments)
  - Atoms (heavyAtoms, nonCHAtoms, metalAtoms, negAtoms, stereoCenters, aromAtoms, sp3CFraction, sp3Atoms, symmetricAtoms)
  - Bonds (nonHBonds, rotBonds, closures)

- Rings (largestRing, rings, carbo, heteroRings, satRings, nonAromRings, aromRings, and others)
- Functional groups (amides, amines, alkylAmines, arylAmines, aromN, basicN, acidicO)
- Stereo (stereoConfiguration)
- 3D properties (globularity, globularity2, surface3d, volume3d)

2.3. Filtering

- Application of Lipinski's filter (Rule of Five) to identify drug-like molecules

2.4. Data Manipulation

- Deletion of columns from the dataset
- Generation of new 2D coordinates for molecular structures so that molecules are oriented the same way if they share a common part (common scaffold)

2.5. Data Querying

- Reading the current dataset for analysis
- Natural language querying of data contents (columns, values, statistics)
- Querying performance strongly depends on the LLM used: a small model like Llama3.1 8B will perform much worse than GPT-4 or 5...
- GPT-4o and GPT-4o-mini models give good results but can make mistakes on minimum/maximum identification tasks and complex statistical calculations, especially GPT-4o-mini.

2.6. Session Tracking

- TCP connection status verification
- Session information querying
- List of available tools

3. LLM Orchestration Architecture

This section explains in detail how the artificial intelligence system of DataWarrior-Agent works: how the LLM receives user messages, how it decides which actions to perform, and how it interacts with DataWarrior macros.

3.1. Orchestrator Definition

The orchestrator is a Python class named DataWarriorOrchestrator defined in the file llm/orchestrator.py. It is not the LLM itself, but a program that coordinates interactions with the LLM.

Specifically, the orchestrator:

- Receives user messages from the Streamlit interface
- Constructs the prompts (instructions) to send to the LLM
- Sends these prompts to the LLM via an API (OpenAI, Anthropic, or Ollama)
- Parses (analyzes) the LLM responses to extract actions to perform

- Triggers the execution of corresponding tools/macros
- Manages conversation history and compression if needed

The LLM itself is an external service (GPT-4 at OpenAI, Claude at Anthropic, or a local model via Ollama). The orchestrator communicates with this service through a client defined in llm/llm_client.py.

3.2. Available Models and Pricing

The interface offers the following models by default:

Model	Provider	Input Price (1M tokens)	Output Price (1M tokens)	Category
GPT-4o	OpenAI	$2.50	$10.00	Large model
GPT-4o-mini	OpenAI	$0.15	$0.60	Large model
GPT-4-turbo	OpenAI	$10.00	$30.00	Large model
Claude-3.5-Sonnet	Anthropic	$3.00	$15.00	Large model
Llama3.1 8B	Ollama (local)	Free	Free	Small model
Mistral 7B	Ollama (local)	Free	Free	Small model

The GPT-4o-mini model offers the best quality-to-price ratio for most use cases. It is cheaper than GPT-4o while offering quite satisfactory performance. However, for good performance in filtering and result analysis, GPT-4o is much better.

3.3. Adding a New Model

This section explains how to extend DataWarrior-Agent to use other language models. The procedure differs depending on whether it is a cloud model via API (OpenAI, Anthropic) or a local model via Ollama.

3.3.1. Cloud Models (OpenAI, Anthropic)

If you want to add new cloud models (for example GPT-5 or other OpenAI/Anthropic models), follow these steps:

File llm/llm_client.py - Method is_large_model()

Add the model name to the list of large models if applicable:

```
@staticmethod
def is_large_model(model: str) -&gt; bool:
    large_models = [
        "gpt-4", "gpt-4o", "gpt-4o-mini", "gpt-4-turbo",
        "gpt-5",  # Add new large models here
        "claude-3-opus", "claude-3-sonnet", "claude-3-5-sonnet",
        "llama3.1:70b", "llama3.1:405b"
        # ...
```

File streamlit_app.py - Model dropdown list

Add the model to the corresponding selection list:

```
# For OpenAI
openai_model = st.selectbox(
    "Model",
    ["gpt-4o-mini", "gpt-4o", "gpt-4-turbo", "gpt-5"],  # Add here
    key="openai_model"
```

```
# For Anthropic
anthropic_model = st.selectbox(
    "Model",
    ["claude-3-5-sonnet-20241022", "claude-3-opus-20240229"],  # Add here
    key="anthropic_model"
```

Note: No additional installation is required for cloud models. Only a valid API key is needed.

3.3.2. Local Models (Ollama)

To add a new local model via Ollama, the procedure is different because the model must first be downloaded locally.

Prerequisite: Installing Ollama

If Ollama is not yet installed on your system, refer to the "Step 3: Install Ollama (Optional)" section of the README.md file which details:

- Installing Ollama
- Downloading models
- Verifying the installation

Procedure for adding a local model

Download the model via Ollama

Before being able to use a local model, you must download it:

```
# Example: download Mistral 7B
ollama pull mistral:7b

# Example: download LLaMA 3.2
ollama pull llama3.2:3b

# Verify the model is properly installed
ollama list
```

Test the model locally

Verify that the model works correctly:

```
ollama run mistral:7b "Hello, how are you?"
```

Configure the model in the interface

Once the model is downloaded, it can be used directly in the Streamlit interface:

- In the sidebar, select "Ollama (Local)"
- In the "Model name" field, enter the exact model name as it appears in ollama list (for example: mistral:7b, llama3.2:3b)
- Click "Connect"

Important note: For Ollama models, the interface uses a free text field (st.text_input) rather than a dropdown list. This allows using any locally installed model without code modification. Simply enter the exact model name.

Classify the model as large or small (Optional)

If you want a new local model to use the Two-Stage Workflow strategy (reserved for large models), add it in llm/llm_client.py:

```
@staticmethod
def is_large_model(model: str) -&gt; bool:
    large_models = [
        "gpt-4", "gpt-4o", "gpt-4o-mini", "gpt-4-turbo",
```

```
        "claude-3-opus", "claude-3-sonnet", "claude-3-5-sonnet",
        "llama3.1:70b", "llama3.1:405b",
        "mistral:22b",  # Example: add a large-size Mistral model
        # ...
```

By default, Ollama models are treated as small models and use the Single-Stage Workflow strategy with semantic search.

3.3.3. Large Model vs. Small Model Distinction

The "large model" vs. "small model" classification determines the processing strategy used:

Two-Stage Workflow Strategy

- Models: GPT-4, GPT-4o, GPT-4o-mini, Claude-3.5 Sonnet, LLaMA 70B+
- Characteristics: Two separate LLM calls (planning then parameterization). Better accuracy.

Single-Stage Workflow Strategy

- Models: LLaMA 8B, Mistral 7B, other models <10B
- Characteristics: A single LLM call with prior semantic search. Variable performance.
- This distinction is detailed in the "Two Processing Strategies" section of this document.

3.3.4. Summary

Cloud Models (API)

- Installation: None (API key only)
- Code configuration: Modify is_large_model() + streamlit_app.py
- Interface addition: Add to dropdown list

Local Models (Ollama)

- Installation: ollama pull model_name
- Code configuration: Optional (is_large_model() if large model)
- Interface addition: Enter the name in the text field

References

- Complete Ollama installation: see README.md → Step 3: Install Ollama (Optional)
- List of available models on Ollama: https://ollama.com/library
- Processing strategies: see "Two Processing Strategies" section of this document

3.4. General Principle: The LLM Never Sees the Macros

A fundamental point to understand: the LLM does not directly know the DataWarrior macro files (.dwam). It does not know what a macro is, nor how they are structured.

The LLM only knows tool descriptions that are presented to it in its context. These descriptions are stored in two JSON configuration files:

config/macro_descriptions.json: Short descriptions for the planning phase

```
{ "macros": {
    "calculate_all_descriptors": "Calculate all 59 molecular descriptors",
    "calculate_descriptors": "Calculate specific molecular descriptors (logP, MW, TPSA, etc.)",
    "apply_lipinski_filter": "Apply Lipinski Rule of Five filter",
    "open_file": "Open a molecular file (SDF, MOL, CSV with SMILES)",
    "read_dataset": "Read the current dataset to analyze its contents"
} }
```

config/macro_schemas.json: Detailed schemas with parameters for the parameterization phase

```
{ "schemas": {
    "calculate_descriptors": {
      "description": "Calculate specific molecular descriptors",
      "parameters": {
        "type": "object",
        "properties": {
          "descriptors": {
            "type": "array",
            "items": {"type": "string"},
            "description": "List of descriptor names to calculate"
          }
        },
        "required": ["descriptors"]
      },
      "valid_descriptors": ["logP", "logS", "totalWeight", "tpsa", ...],
      "aliases": {"mw": "totalWeight", "hbd": "donors", "hba": "acceptors"}
    }
} }
```

3.5. The Two Processing Strategies

The application uses two different strategies depending on the power of the model used. This distinction is necessary because LLM performance degrades significantly when the context becomes too long, a phenomenon called "lost in the middle": models struggle to use information located in the middle of the context.

3.5.1. Why Two Strategies?

Large models (GPT-4, GPT-4o, GPT-4o-mini, Claude-3) have a large context capacity (128k tokens or more) and excellent reasoning capabilities. They can process the complete list of tools with their detailed schemas without noticeable degradation.

Small models (Llama 8B, Mistral 7B via Ollama) have a limited effective context capacity. Although LLaMA 3.1 8B officially supports 128k tokens, studies show that its performance is optimal around 8,000 tokens and degrades significantly beyond 16,000 tokens. Overloading them with information causes hallucinations and parameter errors.

3.6. Strategy 1: Two-Stage Workflow (Large Models)

This strategy uses two separate LLM instances (two distinct invocations with different contexts) to separate planning from execution.

3.6.1. Why Two Separate LLM Instances?

Using two distinct instances (rather than a single continuous conversation) offers several advantages:

- Context isolation: The second instance is not cluttered with descriptions of all irrelevant tools during parameterization
- Separation of responsibilities: Planning (which tools?) is completely separated from execution (with which parameters?)
- Reduced cognitive load: Each instance focuses on a specific task, reducing the risk of confusion
- Independent optimization: Each instance can use different configurations (higher temperature for creative planning, lower for precise parameterization)

3.6.2. Stage 1: Macro Selection and Planning

Objective: Understand the user's request and select the appropriate tools.

What the first LLM instance receives:

- The user's natural language query
- The concise macro catalog (macro_descriptions.json file) containing short descriptions (~30 words per macro) for all available tools
- The conversation history
- The dataset context if available (column names, number of rows)

What this instance does NOT receive:

- The detailed JSON schemas with parameter specifications
- The complete lists of valid descriptors

Concrete example:

The user writes: "Calculate logP and molecular weight"

System prompt sent to instance 1:

```
You are a helpful assistant for DataWarrior, a cheminformatics platform.

AVAILABLE TOOLS:
- calculate_all_descriptors: Calculate all 59 molecular descriptors including
  weight, logP, toxicity...
- calculate_descriptors: Calculate specific molecular descriptors (logP, MW,
  TPSA, etc.)
- apply_lipinski_filter: Apply Lipinski Rule of Five filter to identify
  drug-like molecules
- open_file: Open a molecular file (SDF, MOL, CSV with SMILES)
- read_dataset: Read the current dataset to analyze its contents
- delete_columns: Delete specified columns from the dataset
- generate_2d_coordinates: Generate new 2D coordinates for molecular structures
- save_as_csv: Export the current dataset to CSV format
[... complete list of tools with short descriptions ...]

RESPONSE FORMAT (JSON only):
{ "selected_macros": ["list", "of", "selected", "tools"],
  "plan": "Technical execution plan",
  "response": "Natural language response to show the user" }
```

User message added next:

```
Calculate logP and molecular weight
```

Response from instance 1:

```
{ "selected_macros": ["calculate_descriptors"],
  "plan": "Calculate logP and molecular weight (totalWeight) descriptors
           for all molecules",
  "response": "I will calculate logP and molecular weight for your molecules." }
```

Relevant method: **_stage1_planning()** in orchestrator.py

History stored in: **self.planning_history**

3.6.3. Stage 2: Parameterization and Execution

Objective: Generate the precise parameters for each selected tool.

What the second LLM instance receives:

- The description of the workflow validated at Stage 1
- Only the detailed JSON schemas of the selected tools (typically 1-10 tools, ~500-5000 tokens)

- The conversation history
- The dataset metadata (column names, data types)

What this instance does NOT receive:

- The complete catalog of 200 macro descriptions
- The schemas of unselected tools

Concrete example (continued):

System prompt sent to instance 2:

```
You are a parameterization assistant for DataWarrior. Generate the correct
parameters for each selected tool.

SELECTED TOOLS AND THEIR SCHEMAS:
### calculate_descriptors
Description: Calculate specific molecular descriptors
Parameters: {
  "type": "object",
  "properties": {
    "descriptors": {
      "type": "array", "items": {"type": "string"},
      "description": "List of descriptor names to calculate"
    }
  },
  "required": ["descriptors"]
}
Valid descriptors: ["logP", "logS", "totalWeight", "tpsa", "donors",
                    "acceptors", ...]
Aliases: {"mw": "totalWeight", "molecular weight": "totalWeight",
          "hbd": "donors", ...}

RESPONSE FORMAT (JSON only):
{ "tool_calls": [
    {"name": "tool_name", "arguments": {...}, "reason": "Brief explanation"}
  ]
}
```

Message added:

```
User wants: Calculate logP and molecular weight
Selected tools: calculate_descriptors
```

Response from instance 2:

```
{ "tool_calls": [
    { "name": "calculate_descriptors",
      "arguments": {"descriptors": ["logP", "totalWeight"]},
      "reason": "User requested logP and molecular weight calculation" }
  ]
}
```

Relevant method: _**stage2_parameterization()** in orchestrator.py

History stored in: **self.execution_history** (separate from planning)

3.6.4. Tool Execution

After Stage 2, the orchestrator returns the tool_calls to the Streamlit interface which:

- Calls the calculate_descriptors() function in mcp_server.py
- This function modifies the calcul_selective_descriptors.dwam macro via MacroModifier.update_property_list()
- The macro path is sent via TCP to DataWarrior

- DataWarrior executes the macro and calculates the descriptors
- The result is returned to the interface

3.7. Strategy 2: Single-Stage Workflow (Small Models)

For local models via Ollama, the system uses a different approach that combines semantic search and execution in a single step.

3.7.1. Problem with Small Models

Loading the complete macro catalog would be impractical:

- ~200 descriptions × ~30 tokens each ≈ 6,000 tokens
- This would consume approximately 75% of the effective context before even adding the user query, history, or detailed schemas
- The model would hallucinate or ignore critical information

3.7.2. Preliminary Step: Semantic Search

Before querying the LLM, the system performs a vector search to pre-select relevant tools:

- Query embedding: The user query is converted to a 384-dimensional vector via the all-MiniLM-L6-v2 model (~80 MB, runs on CPU)
- Macro embedding database: All macro descriptions are pre-indexed in a vector database
- Similarity calculation: Cosine similarity is calculated between the query and each macro
- Top-K selection: Only the 5 most similar tools are retained (empirical threshold balancing coverage and context)
- Automatic addition: read_dataset is always added if not already present (for data questions)

Example: For the query "calculate logP", the search returns:

- calculate_descriptors (score: 0.82) ← relevant
- calculate_all_descriptors (score: 0.71) ← relevant
- read_dataset (score: 0.45)
- apply_lipinski_filter (score: 0.38)
- save_as_csv (score: 0.22)

Relevant file: **semantic_search.py**

3.7.3. Single Step: Combined Selection and Parameterization

Unlike large models, a single LLM instance receives all the filtered context:

What the LLM instance receives:

- The 5-6 pre-selected macro descriptions WITH their complete JSON schemas
- The user query and conversation history
- The dataset metadata (if loaded)
- Strict rules on JSON format

Prompt example:

```
You are a DataWarrior assistant for molecular data analysis.
STATUS: DATASET LOADED (150 rows, 8 columns)

AVAILABLE TOOLS:
```

```
### calculate_descriptors
Description: Calculate specific molecular descriptors
Parameters: ['descriptors']  Required: ['descriptors']

### calculate_all_descriptors
Description: Calculate all 59 molecular descriptors
Parameters: NONE - use empty arguments {}

### read_dataset
Description: Read current dataset to analyze contents
Parameters: NONE - use empty arguments {}

=== STRICT RULES ===
1. Output ONLY valid JSON - no text before or after
2. Use ONLY the parameters listed above
3. DO NOT invent parameters that don't exist
4. Tools with "Parameters: NONE" must use empty arguments: {}

=== RESPONSE FORMAT ===
{ "tool_calls": [{"name": "tool_name", "arguments": {}, "reason": "why"}],
  "response": "Your answer to the user" }
```

User message:

```
Calculate logP
```

LLM response:

```
{ "tool_calls": [
    { "name": "calculate_descriptors",
      "arguments": {"descriptors": ["logP"]},
      "reason": "Calculate logP as requested" }
  ],
  "response": "I am calculating logP for your molecules." }
```

Relevant method: **_process_single_stage()** in orchestrator.py

History stored in: **self.conversation_history**

3.7.4. Mandatory Argument Filtering

Small models frequently invent non-existent parameters. The streamlit_app.py file contains a filtering mechanism:

```
TOOL_VALID_PARAMS = {
    "open_file": ["filename"],
    "save_as_csv": ["filename"],
    "calculate_all_descriptors": [],
    "calculate_descriptors": ["descriptors"],
    "apply_lipinski_filter": [],
    "delete_columns": ["columns"],
    "read_dataset": [],  # No parameters!
}

def filter_arguments(tool_name: str, arguments: dict) -&gt; dict:
    valid_params = TOOL_VALID_PARAMS.get(tool_name, [])
    if not valid_params:
        return {}  # Ignore all invented arguments
    return {k: v for k, v in arguments.items() if k in valid_params}
```

For example, if the LLM generates {"filename": "data.csv"} for read_dataset (which has no parameters), the filtering removes this invented argument.

3.7.5. Limitations of Small Models

• Explicit requests required: Vague requests ("analyze these molecules") may retrieve generic macros rather than the expected specific operations

• Limited complex workflows: The system is less effective at proposing multi-step workflows combining macros from semantically distant categories

• Incorrect calculations: Statistics on data (min, max, average) are often wrong

• Variable latency: On CPU only, response times can reach 30-60 seconds; with GPU (even consumer-grade), 2-5 seconds

3.8. Stage 3: Final Response Generation (Common to Both Strategies)

This step occurs only after the successful execution of the read_dataset tool. It is identical for both large and small models.

Objective: Answer questions about the data with actual dataset values, not assumptions.

When is it triggered?

When the user asks a question that requires analyzing the data:

• "How many molecules have a logP > 3?"

• "What is the average molecular weight?"

• "List molecules with TPSA < 100"

Process:

• The orchestrator detects that read_dataset has been executed successfully

• It invokes a third LLM instance (or third call) with the actual data

• This instance receives the original question AND the dataset content (truncated if too large)

• It generates a response based on the actual data

Example:

The user asks: "How many molecules have a logP > 3?"

After execution of read_dataset, the prompt for Stage 3:

```
User question: How many molecules have a logP &gt; 3?

Tool results:
[TOOL: read_dataset] Status: success
Rows: 150, Columns: 8
Column names: Name, Structure, MW, logP, TPSA, HBD, HBA, logS
Data:
Name,MW,logP,TPSA,HBD,HBA,logS
Molecule1,324.5,2.3,78.2,2,5,-3.1
Molecule2,289.3,4.1,92.1,3,6,-2.8
Molecule3,412.7,5.2,65.4,1,4,-4.5
...

RULES:
1. Use the ACTUAL data from the tool results to answer
2. Be precise - use exact values from the data
3. For numerical questions, calculate from the ACTUAL data provided

Answer the question:
```

LLM response:

```
Based on the data, 47 out of 150 molecules have a logP greater than 3.
```

Trigger code (in streamlit_app.py):

```
needs_stage3 = any(
    tr.get("tool_name") == "read_dataset" and tr.get("status") == "success"
    for tr in tool_results
)
if needs_stage3:
    final_response = asyncio.run(
        st.session_state.orchestrator.generate_final_response(
            prompt, tool_results)
    )
```

Relevant method: **generate_final_response()** in orchestrator.py

3.9. Token Management and Compression

Token limits differ depending on the model type:

When the history exceeds the limit, the compress_history() function:

- Preserves the 30% most recent messages
- Summarizes the oldest 70% via a dedicated LLM call to 10% of their original size
- Inserts the summary at the beginning of the history

3.10. Summary Table of the Two Strategies

4. General Code Architecture

The application is organized according to a modular architecture comprising four main components:

```
DataWarrior-Agent/
■■■ core/             # Low-level communication with DataWarrior
■■■ llm/              # Language model orchestration
■■■ macros/           # DataWarrior macro files (.dwam)
■■■ scripts/          # Startup and shutdown scripts
■■■ config/           # JSON configuration files
■■■ mcp_server.py     # MCP tool server
■■■ streamlit_app.py  # Web user interface
■■■ run.py            # Main launcher
```

The communication flow follows this pattern:

- The user interacts via the Streamlit interface
- The message is transmitted to the orchestrator
- The orchestrator queries the LLM (via API) to determine actions
- MCP tools modify macros and send them via TCP
- DataWarrior executes the macros and saves the results
- The results are read and returned to the user

5. Core Module

The core/ module contains the fundamental components for communication with DataWarrior.

5.1. File: core/__init__.py

This file defines the public exports of the core module. It exposes the essential classes and functions for external use:

```
from .tcp_client import (
    DataWarriorTCPClient, TCPConfig, tcp_client,
    send_macro_to_datawarrior, check_connection
)
from .macro_modifier import MacroModifier
```

Its role is to simplify imports in other application modules.

5.2. File: core/tcp_client.py

This file implements TCP communication with the DataWarrior plugin. It constitutes the direct link between the Python application and the DataWarrior software.

5.2.1. TCPConfig Class

This dataclass defines the connection configuration:

- host: server IP address (127.0.0.1 by default)
- port: TCP port (5151 by default, corresponding to the RevalisMacroAgent plugin)
- timeout: wait timeout in seconds (10 by default)

5.2.2. DataWarriorTCPClient Class

This class manages all TCP communication:

Method is_connected(): Checks if the DataWarrior TCP plugin is active by attempting a socket connection. Returns a boolean indicating the connection state.

Method send_macro(macro_path, wait_for_completion, max_wait): Sends the path of a macro to the TCP plugin. This method:

- Checks the existence of the macro file
- Records the timestamp of the synchronization file before execution
- Establishes a socket connection and sends the absolute macro path
- Optionally waits for execution completion by monitoring the synchronization file modification

Method _wait_for_file_change(old_mtime, max_wait): Monitors the update_file.dwar file to detect when DataWarrior has finished executing the macro. This file synchronization technique allows knowing when a long operation is complete.

5.2.3. Utility Functions

- tcp_client: global TCP client instance
- send_macro_to_datawarrior(): shortcut function to send a macro
- check_connection(): shortcut function to check the connection

> ***Importance:*** *This module is critical because it constitutes the sole communication point with DataWarrior. Without it, no operation can be executed in the software.*

5.3. File: core/macro_modifier.py

This file allows dynamically modifying DataWarrior macro files (.dwam) before their execution. It is an essential component that allows parameterizing operations according to user requests.

5.3.1. MacroModifier Class

This class manages reading, writing, and modifying macro files.

Base methods:

- get_macro_path(macro_name): returns the complete path of a macro
- macro_exists(macro_name): checks if a macro exists
- read_macro(macro_name): reads the textual content of a macro
- write_macro(macro_name, content): writes the content of a macro

Specific modification methods:

update_filename(macro_name, new_filename): Modifies all fileName= parameters in a macro. Uses a regular expression to find and replace file paths. This method is used primarily for the file opening macro.

update_filename_in_task(macro_name, task_name, new_filename): Modifies the fileName= parameter only in a specific task. This precision is necessary for macros containing multiple tasks with different files, such as save_csv_file which saves in both CSV and DWAR.

update_property_list(macro_name, properties): Updates the list of descriptors to calculate in a macro. Descriptors are separated by tabs, as per the format expected by DataWarrior.

update_column_list(macro_name, columns): Updates the list of columns for deletion operations.

update_structure_column(macro_name, structure_column): Modifies the name of the column containing molecular structures. This method is important because the structure column name varies depending on the input file format (for example "Structure" for SDF files, "Structure of SMILES" for CSV files with SMILES).

update_all_structure_columns(structure_column): Applies the structure column modification to all macros that use it (descriptor calculation, Lipinski filter, 2D coordinate generation).

Information methods:

- list_macros(): lists all available macros
- get_macro_info(macro_name): returns macro information (detected parameters, current values)

> ***Importance:*** *This module enables the system's flexibility. Without it, macros would be static and could not adapt to varying user requests.*

6. LLM Module

The llm/ module contains the system's artificial intelligence, managing interaction with language models and tool orchestration.

6.1. File: llm/__init__.py

This file defines the LLM module exports and documents the two-strategy architecture:

- Large models (GPT-4, Claude): Two-step workflow (planning then parameterization)
- Small models (Ollama): Semantic search followed by single-step execution

6.2. File: llm/llm_client.py

This file implements a unified client for different language model providers.

6.2.1. ModelProvider Enumeration

Defines the three supported providers: OpenAI, Anthropic, and Ollama.

6.2.2. LLMConfig Class

Configuration dataclass containing:

- provider: the provider (OpenAI, Anthropic, Ollama)
- model: the model name
- api_key: API key (optional for Ollama)
- base_url: base URL (for Ollama: http://localhost:11434)
- temperature: model creativity (0.3 by default)
- max_tokens: maximum number of tokens in response

6.2.3. LLMClient Class

This class abstracts the differences between LLM providers.

Method _init_client(): Initializes the client according to the provider. For OpenAI and Anthropic, it instantiates the official SDKs. For Ollama, it configures the base URL for REST calls.

Method chat(messages, system_prompt): Main asynchronous method that sends a message and returns the response. It dispatches to the appropriate method based on the provider.

Private chat methods:

- _chat_openai(): uses OpenAI's Chat Completions API
- _chat_anthropic(): uses Anthropic's Messages API
- _chat_ollama(): manually builds the prompt and calls Ollama's REST API

Static method is_large_model(model): Determines if a model is considered "large" or "small". This distinction influences the orchestration strategy used.

6.2.4. Factory Functions

- create_openai_client(): creates an OpenAI client
- create_anthropic_client(): creates an Anthropic client
- create_ollama_client(): creates a local Ollama client

> ***Importance:*** *This module enables model interchangeability. The user can choose their provider without modifying the rest of the code.*

6.3. File: llm/semantic_search.py

This file implements semantic search for small models. It allows pre-selecting the most relevant tools before presenting them to the LLM.

6.3.1. SemanticSearch Class

Method _load_model(): Loads the all-MiniLM-L6-v2 embedding model from sentence-transformers. This model is forced to CPU to avoid CUDA compatibility issues.

Method embed(text): Generates the embedding vector (384 dimensions) for a given text.

Method embed_batch(texts): Optimized version for multiple texts.

Method cosine_similarity(a, b): Calculates the cosine similarity between two vectors. A value close to 1 indicates strong semantic similarity.

Method build_tool_index(tools): Builds an embedding index for all available tools. Each tool is represented by the concatenation of its name and description.

Method search(query, tools, top_k): Finds the most relevant tools for a user query. It:

- Ensures the index is built
- Computes the query embedding
- Compares with each tool by cosine similarity
- Returns the top_k tools sorted by decreasing score

Cache methods:

- save_cache(): saves pre-computed embeddings
- load_cache(): loads embeddings from disk

> ***Importance:*** *This module is essential for small models as it reduces context by presenting only relevant tools. Without it, small models would be overwhelmed by too much information.*

6.4. File: llm/token_manager.py

This file manages token limits and conversation history compression.

6.4.1. TokenLimits Class

Dataclass defining limits according to model type:

- csv_warning_threshold: warning threshold for large CSV files
- csv_max_tokens: absolute limit for CSV files
- history_max_tokens: limit triggering history compression
- history_compression_ratio: percentage of history preserved after compression
- summary_ratio: summary size relative to deleted content

6.4.2. Limit Constants

LARGE_MODEL_LIMITS: for GPT-4 and Claude (30000/50000 CSV tokens, 100000 history)

SMALL_MODEL_LIMITS: for Ollama (4000/6000 CSV tokens, 15000 history)

6.4.3. Estimation Functions

estimate_tokens(text): Estimates the number of tokens in a text. Uses the approximation of one token per four characters, sufficient for limit management.

estimate_messages_tokens(messages): Estimates tokens for a list of messages, including metadata overhead.

6.4.4. read_dataset_csv Function

Reads a CSV file and returns its content with limit management:

- Checks file existence
- Reads and parses the content

- Extracts metadata (number of rows, columns)
- Estimates tokens
- Truncates if necessary while preserving the header and first rows
- Returns a dictionary with status, message, data, and metadata

6.4.5. compress_history Function

Compresses conversation history when it exceeds limits:

- Calculates the cutoff point (preserves 30% of recent messages)
- Creates a summary of deleted messages via the LLM
- Replaces old messages with the summary
- Returns the new history and an information message

The _create_history_summary summary function asks the LLM to synthesize previous exchanges by focusing on requested and executed actions, without including raw data.

> ***Importance:*** *This module prevents context overflows that would cause errors. It ensures continuity of long conversations.*

6.5. File: llm/orchestrator.py

This file is the heart of the system's intelligence. It orchestrates LLM calls and determines the actions to execute. Its detailed operation has been explained in the "LLM Orchestration Architecture" section.

6.5.1. Data Classes

ToolCall: Represents a tool call with name, arguments, and reason.

OrchestratorResult: Orchestration result containing success, tool calls, response, possible error, and warnings.

6.5.2. Configuration

Configuration files are loaded at startup:

- macro_descriptions.json: short descriptions for the planning phase
- macro_schemas.json: detailed schemas for parameterization

6.5.3. DataWarriorOrchestrator Class

Constructor: Initializes the orchestrator with an LLM client. Automatically determines the strategy (two-step or single-step) based on the model size. Initializes semantic search for small models.

Method set_compression_llm(api_key): Configures a dedicated LLM for compression (recommended: GPT-4o-mini for its cost/performance ratio).

Dataset context management:

- set_dataset_context(): injects data into the context for answering questions
- clear_dataset_context(): clears the context

Main method process(user_message):

- Checks and compresses history if needed
- Dispatches to the appropriate workflow (two-step or single-step)
- Adds any warnings

6.5.4. Utilities

_clean_json_response(): Cleans responses to extract valid JSON (removes markdown tags, finds matching braces).

reset_history(): Resets all histories.

get_token_usage(): Returns current token usage.

6.5.5. Factory Functions

- create_orchestrator_openai(): creates an OpenAI orchestrator with compression
- create_orchestrator_anthropic(): creates an Anthropic orchestrator
- create_orchestrator_ollama(): creates an Ollama orchestrator

> ***Importance:*** *This module is the brain of the system. It translates user intentions into concrete actions on DataWarrior.*

7. MCP Server

7.1. File: mcp_server.py

This file exposes DataWarrior tools via the MCP (Model Context Protocol) protocol. It bridges the gap between the LLM orchestrator and concrete operations.

7.1.1. Configuration

Definition of project paths:

- MACROS_DIR: macros folder
- CONFIG_DIR: configuration folder
- DATA_INPUT_DIR: input data folder
- DATA_OUTPUT_DIR: output folder
- SNAPSHOT_CSV_PATH: snapshot path for read_dataset

The descriptor catalog is loaded from descriptors_catalog.json.

7.1.2. SessionState Class

Manages the DataWarrior session state:

- source_file: loaded source file
- smiles_column: detected SMILES column
- structure_column: structure column name
- file_loaded: loading indicator
- history: action history

Method set_source(): Configures the source file and automatically updates all macros with the correct structure column name.

7.1.3. _auto_save_snapshot Function

Automatic dataset save after each data-modifying operation. This function:

- Updates the save_csv_file macro to save to _snapshot.csv

- Executes the macro
- Allows read_dataset to access the current data

7.1.4. MCP Tools

read_dataset(): Reads the current CSV snapshot. Automatically excludes structure columns (encoded, bulky, and useless for analysis). Returns cleaned data with metadata.

open_file(filename): Opens a molecular file. This function:

- Resolves the path (relative or absolute)
- Copies to a standardized name (user_upload.{ext})
- Detects the SMILES column for CSV files
- Updates the open.dwam macro
- Launches DataWarrior via subprocess with the macro
- Performs an initial auto-save

save_as_csv(filename): Exports the dataset to CSV. Updates the macro and executes it.

calculate_all_descriptors(): Calculates all 59 molecular descriptors. Executes the corresponding macro and performs an auto-save.

calculate_descriptors(descriptors): Calculates specific descriptors. Validates descriptor names, resolves aliases, updates the macro, and executes it.

get_available_descriptors(category): Lists available descriptors, optionally filtered by category.

apply_lipinski_filter(): Applies the Rule of Five filter.

delete_columns(columns): Deletes columns from the dataset.

generate_2d_coordinates(): Generates new 2D coordinates.

get_connection_status(): Checks the TCP connection.

get_session_info(): Returns session information.

list_available_tools(): Lists all tools organized by category.

7.1.5. _detect_smiles_column Function

Automatically detects the SMILES column in a CSV file by searching for standard names (smiles, smile, smi, canonical_smiles, isomeric_smiles).

7.1.6. CLI

The file can be executed directly with options:

- --test: tests tools locally
- --list: lists available tools
- Without options: launches the MCP server in stdio mode

> ***Importance:*** *This module defines the interface between the Python world and DataWarrior. Each tool corresponds to a possible user action.*

8. Streamlit Interface

8.1. File: streamlit_app.py

This file implements the web user interface with Streamlit.

8.1.1. Configuration

- Page configured in wide mode with microscope icon
- URLs defined for the project and noVNC
- List of valid parameters for each tool (for filtering)

8.1.2. filter_arguments Function

Filters arguments invented by small models. Local LLMs like Llama sometimes invent non-existent parameters. This function:

- Retrieves the list of valid parameters for the tool
- Removes any unrecognized argument
- Logs removed arguments for debugging

8.1.3. CSS Styling

Application of a custom dark theme:

- Sidebar in dark colors (#1a1a2e)
- Buttons with hover effect
- Colored progress bars
- Version badge

8.1.4. API Key Management

- load_api_keys(): loads keys from .config/api_keys.json
- save_api_keys(): saves keys persistently

8.1.5. execute_tool Function

Executes an MCP tool with context management:

- Filters invalid arguments
- Executes the tool
- For read_dataset: injects data into the orchestrator context
- For data-modifying tools: performs an auto-refresh of the context

8.1.6. Streamlit Session State

- chat_history: message history
- orchestrator: LLM orchestrator instance
- model_type: connected model type
- api_keys: saved API keys
- last_opened_file: last opened file

8.1.7. Sidebar

The sidebar contains:

Model selection:

- Provider choice (Ollama, OpenAI, Anthropic)
- Provider-specific configuration (API key, model name)
- Connect button

DataWarrior status:

- TCP connection indicator
- Instructions if not connected

Token usage:

- Progress bar for history
- Progress bar for dataset
- Warnings if near limits

Actions:

- Button to clear chat
- Button to clear dataset context

8.1.8. Main Area

VNC View: Iframe displaying DataWarrior via noVNC (height 950px).

Upload section: File uploader accepting SDF, MOL, CSV, TXT. For each new file:

- Clears the old context
- Deletes the old snapshot
- Opens the file in DataWarrior

Chat section:

- Container with scrollable history
- Input field for messages

Message processing:

- Addition of the user message
- Execution of the orchestrator
- Execution of determined tools
- Final response generation if needed
- Display of results

> ***Importance:*** *This file is the user entry point. It brings together all components into a coherent interface.*

9. DataWarrior Macro System

DataWarrior macros are XML files with the .dwam extension that define automated task sequences.

9.1. General Structure

9.2. Macro: open.dwam

This macro performs two operations:

RevalisMacroAgent5151: starts the TCP agent plugin on port 5151. This custom plugin allows DataWarrior to receive commands over the network.

openFile: opens the specified file. The path is dynamically modified by the Python code.

Starting the TCP agent is crucial as it establishes the communication channel between Python and DataWarrior. As soon as an SDF, CSV, or other file is placed in the file box present in the Streamlit interface, this macro executes automatically.

9.3. Macro: calcul_all_descriptors.dwam

This macro:

calculateCompoundProperties: calculates all 59 molecular descriptors. Descriptors are separated by tabs in propertyList. The largestFragment=true option ensures only the largest fragment is considered (useful for salts).

saveFileAs: saves the file in DWAR format. This save serves as a synchronization signal for the TCP client.

9.4. Macro: calcul_selective_descriptors.dwam

Identical structure to calcul_all_descriptors, but the propertyList parameter contains a placeholder (totalWeight) that is dynamically replaced by the list of descriptors requested by the user.

9.5. Macro: lipinski_filter.dwam

This macro applies the Lipinski rule:

- Calculates the necessary properties (weight, logP, hydrogen bond donors and acceptors)
- Applies range filters:
  - Molecular weight between 0 and 500 Da
  - logP between -2000 and 5
  - H donors between 0 and 5
  - H acceptors between 0 and 10

Molecules outside these ranges are filtered (hidden, not deleted).

9.6. Macro: delete_column.dwam

Deletes the specified columns. The columnList is dynamically modified. Columns are separated by tabs.

9.7. Macro: generate_new_2d_coord.dwam

Generates new 2D coordinates for molecular structures. The scaffoldMode=centralRing option orients molecules around their central ring.

9.8. Macro: save_csv_file.dwam

This macro performs two saves:

- CSV export to the specified path (dynamically modified)

- DWAR save for synchronization

The double save allows exporting data while maintaining the synchronization mechanism.

9.9. Importance of the update_file.dwar File

This file plays a central role in synchronization. Each macro saves it at the end of execution. The TCP client monitors its modification date to detect the end of operations. It is a simple but robust mechanism for asynchronous communication.

10. Startup Scripts

10.1. File: scripts/start_vnc.sh

This script starts the VNC stack to display DataWarrior in the browser:

- Cleanup: stops existing processes (Xvfb, x11vnc, websockify)
- Xvfb startup: launches a virtual X server on display :2 with a resolution of 2560x900 pixels and 24-bit color
- Openbox configuration: creates a configuration for the window manager that:
  - Automatically maximizes all windows
  - Removes decorations (title bars)
  - Gives focus automatically
- x11vnc startup: launches the VNC server connected to display :2, with password and share mode
- noVNC startup: launches the websocket proxy that allows VNC access from a web browser

10.2. File: scripts/start_all.sh

This script starts the entire application:

- Launches the VNC script in the background
- Waits 5 seconds for initialization
- Launches Streamlit in headless mode on port 8501
- Displays access URLs

10.3. File: scripts/stop_all.sh

Properly stops all processes:

- Streamlit
- Xvfb
- x11vnc
- websockify
- Openbox
- DataWarrior

10.4. File: run.py

Python launcher that:

- Executes the stop script
- Executes the VNC script
- Executes the startup script
- Automatically opens the browser to the application URL

11. Data Flow

11.1. Scenario: Descriptor Calculation

- The user types "calculate logP and molecular weight"
- The message is sent to the orchestrator
- The orchestrator (Stage 1) analyzes the request and selects the calculate_descriptors tool
- The orchestrator (Stage 2) determines the parameters: {"descriptors": ["logP", "totalWeight"]}
- The interface executes the MCP tool calculate_descriptors
- The MCP server:
  - Validates the descriptor names
  - Modifies the calcul_selective_descriptors.dwam macro to set propertyList=logP\ttotalWeight
  - Sends the macro path via TCP
- DataWarrior:
  - Receives the path via the TCP plugin
  - Executes the macro
  - Calculates the descriptors
  - Saves the DWAR file
- The TCP client detects the file modification and returns success
- The MCP server executes the auto-save to create the CSV snapshot
- The interface displays the confirmation to the user

11.2. Scenario: Data Question

- The user types "how many molecules have a logP greater than 3?"
- The orchestrator detects a data question
- If the dataset context is absent:
  - Selects the read_dataset tool
  - Executes the tool to load the data
- The orchestrator (Stage 3):
  - Receives the dataset data
  - Analyzes the CSV to count molecules meeting the criterion
  - Generates a precise response
- The interface displays the response with the exact count

12. Guide: Adding a New Feature

This section explains step by step how to extend DataWarrior-Agent with a new feature. We will use the example of adding a function to filter molecules by molecular weight range.

12.1. Step 1: Create the Macro in DataWarrior

The first step is to manually record the macro in DataWarrior.

Open DataWarrior with a test file

Launch DataWarrior and open a molecular file (SDF or CSV) to have data to work with.

Start recording

In the menu, go to:

Macro → Start Recording

From this point on, all actions you perform in DataWarrior will be recorded.

Perform the desired actions

Manually execute the operations you want to automate. For example, for a molecular weight filter:

- Go to Chemistry → Calculate Properties
- Select "Total Molweight"
- Click OK
- Apply a filter on the Molweight column
- Save the synchronization file

Important: at the end of your actions, save the file in DWAR format to the synchronization path:

File → Save As → ./data/input/update_file.dwar

This save is necessary for the TCP client to detect the end of execution.

Stop recording

Macro → Stop Recording

Export the macro

Macro → Export Macro

Save the file with a descriptive name in the macros/ folder, for example:

macros/filter_by_weight.dwam

12.2. Step 2: Modify the Exported Macro

Open the exported .dwam file with a text editor. You will get something like:

Modifications to make:

Rename the macro:

```
&lt;macro name="filter_by_weight"&gt;
```

Correct the save path to use a relative path:

```
&lt;task name="saveFileAs"&gt;
    fileName=./data/input/update_file.dwar
&lt;/task&gt;
```

Identify the parameters that will need to be dynamically modified. In this example, the values 200 and 500 of the filter will need to be parameterizable.

The final macro:

12.3. Step 3: Add a Method in MacroModifier

If your macro requires dynamic parameters, you must add a method in core/macro_modifier.py.

Open the file and add a new method:

```
def update_weight_range(
    self, macro_name: str, min_weight: float, max_weight: float
) -&gt; dict:
    """
    Updates the weight range in the filter_by_weight macro.

    Args:
        macro_name: Name of the macro (without .dwam)
        min_weight: Minimum molecular weight
        max_weight: Maximum molecular weight

    Returns:
        dict: Dictionary containing:
            - status (str): 'success' or 'error'
            - message (str): Descriptive message about the operation
    """
    # Read the macro content
    content = self.read_macro(macro_name)
    if content is None:
        return {
            "status": "error",
            "message": f"Macro not found: {macro_name}"
        }

    # Pattern to find settings=MIN MAX in changeRangeFilter
    pattern = r'(&lt;task name="changeRangeFilter"&gt;\s*settings=)[^\n]+'

    # Check if the weight filter settings exist
    if not re.search(pattern, content):
        return {
            "status": "error",
            "message": "No weight filter settings found in macro"
        }

    # Replace with new values
    updated_content = re.sub(
        pattern,
        rf'\g&lt;1&gt;{min_weight} {max_weight}',
        content
    )

    # Write the updated macro
    if self.write_macro(macro_name, updated_content):
        return {
            "status": "success",
            "message": f"Updated weight range to: {min_weight} - {max_weight}"
        }
    else:
        return {
            "status": "error",
            "message": "Failed to write macro"
        }
```

12.4. Step 4: Add the Tool in the MCP Server

Open mcp_server.py and add a new function decorated with @mcp.tool():

```
@mcp.tool()
def filter_by_weight(min_weight: float = 0, max_weight: float = 500) -&gt; dict:
    """
    Filter molecules by molecular weight range.

    This tool removes molecules outside the specified weight range,
    keeping only those with molecular weight between min_weight and
    max_weight.

    Args:
        min_weight: Minimum molecular weight in Daltons (default: 0)
        max_weight: Maximum molecular weight in Daltons (default: 500)

    Returns:
        dict: Status dictionary with success/error message
    """
    # Check DataWarrior connection
    if not check_connection():
        return {
            "status": "error",
            "message": "DataWarrior TCP plugin not connected."
        }

    # Validate input parameters
    if min_weight &lt; 0:
        return {
            "status": "error",
            "message": "Minimum weight cannot be negative"
        }
    if max_weight &lt;= min_weight:
        return {
            "status": "error",
            "message": "Maximum weight must be greater than minimum"
        }

    # Update macro with weight range parameters
    result = macro_modifier.update_weight_range(
        "filter_by_weight", min_weight, max_weight
    )
    if result["status"] != "success":
        return result

    # Execute the DataWarrior macro
    macro_path = MACROS_DIR / "filter_by_weight.dwam"
    result = send_macro_to_datawarrior(macro_path)

    # Handle successful execution
    if result["status"] == "success":
        session.add_to_history("filter_by_weight", {
            "min_weight": min_weight,
            "max_weight": max_weight
        })
        _auto_save_snapshot()
        return {
            "status": "success",
            "message": f"Filtered molecules to weight range
                        {min_weight}-{max_weight} Da"
        }
    return result
```

Don't forget to add the tool in the list_available_tools() function:

```
"filters": {
```

```
    "apply_lipinski_filter": "Apply Lipinski Rule of Five",
    "filter_by_weight": "Filter molecules by molecular weight range"
    # New line
},
```

12.5. Step 5: Update Configuration Files

Modify config/macro_descriptions.json

Add the description of your new tool:

```
{ "macros": {
    "filter_by_weight": "Filter molecules by molecular weight range,
        keeping only those within specified min/max values",
    ...
} }
```

Modify config/macro_schemas.json

Add the detailed schema of your tool:

```
{ "schemas": {
    "filter_by_weight": {
      "description": "Filter molecules by molecular weight range",
      "parameters": {
        "type": "object",
        "properties": {
          "min_weight": {
            "type": "number",
            "description": "Minimum molecular weight in Daltons",
            "default": 0
          },
          "max_weight": {
            "type": "number",
            "description": "Maximum molecular weight in Daltons",
            "default": 500
          }
        },
        "required": []
      },
      "examples": [
        {"min_weight": 200, "max_weight": 400},
        {"max_weight": 300}
      ]
    },
    ...
} }
```

12.6. Step 6: Update the Streamlit Interface

Open streamlit_app.py and make two modifications:

Add valid parameters

In the TOOL_VALID_PARAMS dictionary, add your tool:

```
TOOL_VALID_PARAMS = {
    "filter_by_weight": ["min_weight", "max_weight"],  # New line
    ...
}
```

Add the function import

In the imports at the top of the file:

```
from mcp_server import (
```

```
    ...,
    filter_by_weight  # New line
)
```

Add to the tools dictionary

In the execute_tool() function:

```
tools = {
    "filter_by_weight": filter_by_weight,  # New line
    ...
}
```

Add to data-modifying tools

```
DATA_MODIFYING_TOOLS = {
    "filter_by_weight",  # New line
    ...
}
```

12.7. Step 7: Test the New Feature

- Restart the application with ./scripts/stop_all.sh then ./scripts/start_all.sh
- Connect to an LLM in the interface
- Load a molecular file
- Test your new tool in natural language:
  - "Filter molecules between 200 and 400 daltons"
  - "Keep only molecules under 300 Da"
- Verify in DataWarrior that the filter is properly applied

12.8. Practical Tips

- **Test the macro manually first:** Before integrating into the code, make sure the macro works correctly by executing it manually in DataWarrior.
- **Use relative paths:** Always use ./data/input/ and ./data/output/ rather than absolute paths.
- **Think about synchronization:** Each macro must end with a save to ./data/input/update_file.dwar.
- **Document the schema well:** The more precise the schema, the better the LLM will understand how to use the tool.
- **Handle errors:** Validate parameters and return clear error messages.