The Spatial Pooler operates on mini-columns connected to sensory inputs (Yuwei, Subutai and Hawkins, 2017)The TM is responsible for learning of sequences from SDR. Experiments in this work show that the current version of the Spatial Pooler is instable. During the learning process, learned patterns will be forgotten and learned again. Results show that the Spatial Pooler oscillates between stable and unstable stable. Moreover, experiments show the instability is related to the single pattern and not to the set of patterns.

----------- WS 2025/2026 -------------

ML 25/26-01 AI Preprocessing Agent/Plugin for Improving OCR Performance

The task

Development of an AI Agent Tool (Plugin) for Adaptive Image Manipulation to Enhance OCR Entity Extraction.

Project Overview

Optical Character Recognition (OCR) systems often struggle with poor image quality, leading to inaccurate text and entity extraction. Modern AI agents increasingly rely on OCR for processing documents from images (e.g., invoices, forms, IDs). To address this, this project involves building an AI Agent/tool/plugin that intelligently preprocesses images through various manipulations to improve OCR readability.

The tool will take a single input image and, based on a user's natural language prompt, generate multiple enhanced versions of the image by applying targeted manipulations (e.g., zoom, skew correction, contrast adjustment). The output will be a set of processed images, stored in a configurable storage system, each identified by a unique reference name.

This project is suitable for students interested in computer vision, AI agents, prompt engineering, and plugin/tool development (e.g., for integration with frameworks like LangChain, AutoGen, or custom AI agents).

Objectives

- Design and implement a reusable AI tool/plugin that accepts an image and a user prompt.
- Use AI (e.g., LLM-based reasoning) to interpret the prompt and decide which image manipulations to apply.
- Apply a variety of image processing techniques to generate multiple enhanced variants.
- Store the original and processed images in a configurable storage backend (starting with local file system).
- Ensure images are named and referenced consistently for easy retrieval and downstream OCR use.

Functional Requirements

1. **Input Handling**

- Accept an input image (supported formats: JPG, PNG, TIFF, BMP). The image is entered by its reference like 123ABC. Your task is to provide an innovative way to deal with inputs.
- Accept a natural language prompt from the user describing desired improvements (e.g., "ZoomIn the image with start zoom of 10% up to 50% with the step 5%", or "Rotate the image with start rotation 0 up to 175 degree with the step of 10 degree.").

2. **Prompt Interpretation and Decision Making**
   - Use the OpenAI to analyze the prompt and select appropriate manipulations.
   - Possible manipulations to implement (at least 5 required, more for bonus):
     - Zoom in / Zoom out (cropping or resizing specific regions).
     - Contrast optimization (auto-contrast, histogram equalization).
     - Brightness adjustment.
     - Sharpening / blurring reduction.
     - Skew correction / deskewing (perspective transform).
     - Noise reduction (denoising filters).
     - Binarization (thresholding for black-and-white text enhancement).
     - Rotation correction.
     - Background removal
3. **Image Processing**
   - Generate multiple output images (depending on user's prompt). For example: "zoomin with start 10% in steps of 50% up to 200%". For this prompt the tool will return : 10, 60, 110, 160, 210?. You can decide if 210 will be returned or not.
   - Each variant applies a different combination or sequence of manipulations based on the AI's reasoning.
4. **Output and Storage**
   - Return a list of processed images with their unique reference IDs.
   - Store all images (original + processed) in a configurable storage system:
     - Default implementation: Local file system (save to a specified folder).
     - Images must be named using their reference ID (e.g., img1234_original.png, img1234_contrast_enhanced.png, img1234_zoomed_sharpened.png or simple numeric/alpha-numeric IDs like ABG787.png).

Example:

```
[Description("Performs zooming in of the given image. It returns the comma-separated list of references of generated images.")]
public Task<string> ZoomInImage(
[Description("The reference number of the image, which should be zoomed in.")] string imageReference,
[Description("The first zoom in in percent.")] int startZoom,
[Description("The The last zoom in in percent")] int endZoom,
[Description("The delta zoom, which defines zoomin step.")] int zoomStep)
{
   throw new NotImplementedException();
}
```

ML 25/26-02 AI Document Classification Agent/PlugIn

Project Requirements

1. General Requirements

- The project shall implement a **console-based applications** in **.NET (C#)**.
- The system shall support two operational modes (see figure below):
  - **Document Classification Mode (application 1)**
  - **Agent Mode (application 2)**
- The application shall be designed with **modularity and extensibility** (Dependency Injection) in mind.

2. Document Classification Mode Requirements

2.1 Input Structure

- The system shall accept a root directory containing **K subfolders**, where each subfolder represents a **document class**.
- Each folder shall contain an arbitrary number of documents (**N ≥ 1**).
- The number of classes (**K**) and documents per class shall be configurable and unrestricted.

2.2 Supported Document Types

- The system shall support the following document formats:
  - .txt
  - .docx
  - .pdf
- Document parsing shall be performed using **provided libraries**, accessed exclusively through a defined **pre-processing interface (**one library will be provided**)**.

2.3 Document Processing

- The system shall extract textual content from each document during preprocessing.
- For each document, the system shall generate a **vector embedding** using a predefined embedding model.
- The system shall calculate a **centroid embedding** for each document class based on the embeddings of its documents.

2.4 Persistence

- The system shall persist:
  - Individual document embeddings

    - Class centroid embeddings
    - Metadata (document name, class identifier, timestamp)
- For example, a **relational database** (e.g., SQL Server) can be used for persistent storage.

3. Agent Mode Requirements

3.1 User Interaction

- The system shall allow the user to enter a **natural language prompt** via the console.
- The prompt shall represent an intent to retrieve or identify a document class.

3.2 Agent and Plug-In Architecture

- The system shall map the user prompt to an **Agent Plug-In Function** with an intent argument of type string.
- The system shall generate an embedding for the intent argument using the same embedding model as in classification mode.

3.3 Similarity Matching

- The intent embedding shall be compared against all stored class centroid embeddings.
- **Cosine similarity (or cosine distance)** shall be used as the similarity metric.
- The system shall determine the best-matching document class based on similarity scores.

3.4 Output

- The system shall display:
    - The most relevant document class
    - Optional similarity scores for all classes
- The output shall be human-readable and suitable for command-line interaction.

ML 25/26-03 AI Recommendation Agent/PlugIn

1. General Requirements

- The project shall consist of **two separate console applications** implemented in **.NET (C#)**: Import Mode Application (figure on left) and Recommendation Agent Application (figure on right).
- The system shall be based on an **agent and plug-in (tool) architecture**.
- All embeddings and document metadata shall be **persisted in a relational database** (e.g., SQL Server) see figure middle = store embeddings….
- The solution shall emphasize **clean architecture, modularity, and extensibility**.
- AT least two different data-sets should be used. For example: Movies, User Support Questions Database.

2. Import Mode Application Requirements

2.1 Input and Document Handling

- The system shall support importing documents from a configurable directory structure.
- Supported document formats shall include:
    - .txt
    - .docx
    - .pdf
- A provided **document parsing library** shall be used for extracting textual content.
- Parsing shall be accessed through a well-defined **preprocessing interface**.

2.2 Embedding Generation

- The system shall generate a **vector embedding** for each imported document using a predefined embedding model.
- Embeddings shall represent the semantic content of each document.

2.3 Metadata Management

- For each document, the following metadata shall be stored:
    - Document identifier
    - File name and file type
    - Import timestamp
    - URL (so, the user can open the recommended document)

    - Optional category or source information

2.4 Persistence

- The system shall persist:
    - Document embeddings
    - Extracted text or references to it
    - Associated metadata
- A relational database (e.g., SQL Server) shall be used for storage and retrieval.
- The persistence layer shall support efficient similarity queries.

3. Recommendation Agent Application Requirements

3.1 User Interaction

- The application shall provide a **console-based interface** for user input.
- The user shall enter a **natural language prompt** representing a search or recommendation request (e.g., “Looking for USB cable”).

3.2 Agent and Plug-In Architecture

- The agent shall map the user prompt to one or more **intent arguments** of a plug-in (tool) function.
- The mapped intent shall be represented as a string and used as input for embedding generation.

3.3 Similarity Matching

- The system shall generate an embedding for the intent argument using the same embedding model as in Import Mode.
- The intent embedding shall be compared to stored document embeddings using **cosine similarity**.
- The system shall identify and rank the most relevant documents.

3.4 Recommendation Output

- The system shall return:
    - The best matching document as the primary recommendation
    - Optionally, a ranked list of top-N recommended documents
- The output shall include relevant metadata to help the user identify the recommended documents.

ML 25/26-04 Language dependent similarity investigation

Related to embedding models.

Introduction: Are embedding models language-agnostic?

What is usually true

- **Monolingual models (English-only, etc.)**: *not* language-agnostic. Queries in other languages perform worse because the vector space is not properly aligned.
- **Multilingual / cross-lingual embedding models**: *partially* language-agnostic. Well-trained models learn to place semantically equivalent sentences from different languages **close to each other in the same vector space**.

Where things often break down

Even with multilingual embeddings, there are common weaknesses:

- **Low-resource languages** (less training data) → weaker alignment
- **Domain-specific language / jargon** (invoices, orders, legal text) → semantic drift
- **Short queries** ("USB cable", "orders") → higher ambiguity
- **Named entities, product codes, numbers** → tokenization and normalization issues
- **Mixed scripts** (Latin/Cyrillic), diacritics → model-dependent behavior

In short: **language-agnostic behavior is an empirical quality, not a guarantee**.

- **How to test this properly**

1) Cross-lingual semantic similarity (fastest test)

Create a set of **parallel or semantically equivalent sentences** in multiple languages.

Example (DE / EN / HR):

- DE: "Ich suche Bestellungen aus 2024."
- EN: "I'm looking for orders from 2024."
- HR: "Tražim narudžbe iz 2024."

Test procedure:

- Generate embeddings for all variants
- Measure **cosine similarity**
- Expectation: equivalent meanings → **high similarity**
- Add negative examples (e.g., invoices vs orders) → **low similarity**

Metrics:

- Average similarity of equivalent pairs
- Distance to hard negatives
- Separability score: (sim_positive - sim_negative)

2) Cross-lingual retrieval (most realistic for document systems)

This is the most important test for document classification or recommendation.

Setup:

- Documents are stored in mixed languages (e.g., German, English, Croatian).
- Queries are also in different languages.

Procedure:

1. Index: generate embeddings for all documents
2. Query: “Looking for orders” (EN)
3. Retrieve top-k documents using cosine similarity
4. Check whether the correct documents (orders) appear at the top, even if the documents are in another language (e.g., German “Bestellung”, “Auftrag”).

Metrics:

- **Recall@k** (e.g., @5, @10)
- **MRR** (Mean Reciprocal Rank)
- **nDCG@k** (for ranking quality)

3) Language-swap stress test (reveals true language agnosticism)

Use the same queries but change only the language:

- Query A (DE): “Suche Rechnungen”
- Query B (EN): “Looking for invoices”
- Query C (HR): “Tražim račune”

Expected behavior with good language-agnostic embeddings:

- Very similar top-k result lists
- Similar similarity scores
- Same predicted class (for classification)

4) Baselines (to verify real cross-lingual performance)

Baseline 1: Translate-then-embed

- Translate the query into the document language (e.g., everything to German)
- Generate embeddings and retrieve documents
- If this performs significantly better than direct multilingual embeddings, the embedding model is not well aligned.

Baseline 2: Monolingual embeddings

- Use a German-only embedding model
- English queries will perform poorly → highlights the gap

Project Requirements

1. Purpose and Scope

- The application shall evaluate whether a chosen embedding model is **language-agnostic (cross-lingual aligned)**, i.e., semantically equivalent texts in different languages produce embeddings that are close in vector space.
- The application shall be implemented as a **.NET (C#) console application** and support reproducible experiments and reporting.

2. Core Functional Requirements

2.1 Embedding Provider Abstraction

- The system shall use Microsoft Agent Framework
- The architecture shall allow swapping models without changing evaluation logic (e.g., Azure OpenAI, OpenAI, local HuggingFace, custom service).

2.2 Test Dataset Management

- The system shall load evaluation datasets from external files (JSON/CSV).
- A dataset shall support:
  - **Positive pairs**: semantically equivalent texts across languages (e.g., EN–DE, EN–HR).
  - **Negative pairs**: semantically different but lexically similar texts (“invoice” vs “order”).
  - **Labels**: language, topic/class, pair-id, and ground-truth relevance.
- The application shall support multiple languages per concept (≥2, ideally ≥3).

2.3 Similarity Computation

- The system shall compute **cosine similarity** between embeddings.

3. Evaluation Modes (Unit Tests)

3.1 Cross-Lingual Pair Similarity Test

- The system shall compute similarity statistics for:
  - Positive cross-lingual pairs
  - Negative pairs
  - Monolingual paraphrase pairs (optional baseline)
- Output shall include (Excel):
  - Mean/median similarity for each category
  - Standard deviation
  - Separability score (e.g., mean(pos) − mean(neg))
  - Threshold sweep metrics (see 3.4)

3.2 Cross-Lingual Retrieval Test (Information Retrieval Benchmark)

- The system shall implement retrieval evaluation where:
  - A corpus of documents exists in mixed languages
  - Queries exist in one or more languages
  - Ground-truth relevant documents are known
- The system shall compute:
  - **Recall@k** (k configurable)
  - **MRR**
  - **nDCG@k** (optional but recommended for Master level)
- The system shall support “language swap” experiments:
  - Same query intent in multiple languages should retrieve similar top-k results.

3.3 Language Clustering Bias Test (Diagnostic)

- The system shall calculate whether embeddings cluster by **language** rather than **meaning** by:
  - Comparing average intra-language similarity vs inter-language similarity for the same topic
  - Reporting a “language bias index”:
    - e.g., bias = mean(sim_same_language) - mean(sim_cross_language_same_topic)
- (Optional advanced) Export embeddings for external visualization (e.g., UMAP) but visualization itself is not required.

3.4 Threshold and Calibration Test

- The system shall support evaluation across a similarity threshold range (e.g., 0.1–0.9).
- It shall compute classification-style metrics for pair tasks:
  - Precision / Recall / F1 for “same meaning” detection
  - ROC-AUC (optional)
- The application shall output a recommended threshold for cross-lingual matching under defined assumptions.

4. Baseline Comparison Requirements

- The system shall support at least one baseline strategy:
  - **Translate-then-embed** baseline (via pluggable translator interface), OR
  - A second embedding provider (e.g., monolingual vs multilingual).

- The application shall produce comparative results between the primary model and baselines.

5. Persistence and Reproducibility Requirements

- The system shall persist all required data and results.

ML 25/26-05 Python Code Tool Interpreter for Chart Creation

Large Language Models (LLMs) have demonstrated significant potential in generating executable code from natural language instructions, particularly in the domains of data analysis and visualization. Despite these capabilities, LLM-based agents typically lack the ability to execute the generated code autonomously. This project investigates the design and implementation of an intelligent agent based on the Microsoft Agent Framework using .NET and C#, augmented by a custom Code Interpreter Tool implemented as an agent plug-in. The agent translates user prompts—such as requests for charts or diagrams—into Python code for data visualization, while the Code Interpreter Tool is responsible for executing this code and generating the corresponding visual output. A central focus of the project is the robust mapping of user-provided data to executable code, as well as the interpretation and execution of dynamically generated Python scripts using appropriate libraries. Furthermore, the project explores strategies for efficiently handling large datasets and supporting the generation of multiple visualizations from complex user inputs. Key challenges addressed include scalability, data transfer between agent components, and the reliable execution of model-generated code. The outcome of this work aims to identify best practices for integrating code execution capabilities into LLM-driven agent architectures for advanced data visualization tasks.

Code example (just an idea).

```
[Description("Creates the chart from given values and provided python code.")]
public Task<string> CreateChart(

[Description("The list of values of the chart.")]
float[] values,

[Description("The Python code to create the chart from given values.")]
string pythonCode)
{
        throw new NotImplementedException();
}
```

ML 25/26-06 AI Storage Tool/Plugin

Goal

Students shall build a **test agent** using the **Microsoft Agent Framework** and a corresponding **Storage Tool / Plug-in** that enables natural-language file operations across configurable storage backends.

Functional Requirements

- The agent shall interpret user prompts and invoke corresponding **Storage Tool** functions.
- The Storage Tool shall support the following operations:
    1. **Upload**: read a file from the configured **Input Store** and upload it to the configured **Output Store** using a user-provided reference name.
        - Examples:
            - "Take the file c:\temp\hsh.txt and upload it with the reference name BLUE"
            - "Take the file c:\temp\hsh.txt and upload it as 123abc.txt."
            - "Upload file c:\temp\hsh.txt"
    2. **List/Count**:
        - "How many files are in store?"
        - "How many files contain 'abc' in the name?"
    3. **Delete**:
        - "Delete files that start with 'X'."
    4. **Read**:
        - "Get content of file abc."
    5. **Get** :
        - "Summarise file blue." (summary generated from file content)

Storage Architecture Requirements

- Input and Output stores shall be implemented via **interfaces** and wired using **Dependency Injection (DI)**.
- The solution shall provide at least two concrete store implementations, such as:
    - **Local file system store**
    - **FTP store** (or similar remote store, e.g., SFTP)

- Stores shall be configurable and selectable without code changes.

ML 25/26-07 ANN Indexing

ANN (Approximate Nearest Neighbor) indexing for embedding vectors is about finding *similar vectors fast* without checking every vector. In this project you will use SQL Server Native Vector Search as embedding database.

Core idea

- Embeddings are high-dimensional vectors where distance ≈ semantic similarity.
- Exact search is too slow at scale, so ANN trades a tiny bit of accuracy for big speed gains.

How it works (high level)

1. **Index construction**
    - Vectors are organized into a structure that preserves neighborhood relationships.
    - Common strategies:
        - **Graph-based (e.g., HNSW):** each vector connects to nearby vectors, forming a navigable graph.
        - **Clustering (e.g., IVF):** vectors are grouped into clusters via k-means; each vector belongs to a centroid.
        - **Trees / hashing (e.g., KD-trees, LSH):** space is partitioned to reduce search scope.

    In this project we will focus on **K-Means,** which was already implemented at this university.
2. **Search**
    - A query vector of the user's prompt is compared only against a *small subset* of candidates:
        - In clustering: find nearest centroids → search vectors inside those clusters.
        - In graphs: start at an entry point and greedily "walk" to closer neighbors.
    - Distance metrics is cosine similarity.
3. **Approximation**
    - Not all vectors are checked.

    - Result is *very likely* (but not guaranteed) to be the true nearest neighbors.
    - Parameters control the speed vs. recall trade-off.

Why clustering helps

- Clustering reduces the search space from *millions* → *thousands*.
- Works especially well when embeddings naturally form semantic groups.

In one sentence

ANN indexing structures embedding vectors so similarity search touches only the most promising regions of the vector space, making large-scale semantic search fast and practical.

ML 25/26-08 Implementing Tests for LLM Prompts with Semantic Assertions

Project Overview

This project involves building a simple testing framework (or application) for evaluating LLM prompts. The core functionality is:

- A user inputs a prompt (and optionally any context or variables).
- The system sends it to an LLM (e.g., via OpenAI API, Grok API, or a local model like Llama).
- The LLM generates an output.
- The system runs automated tests /assertions on the output to determine if it passes.
  It loads prompts and related expected results from a local file like JSON.

The key challenge is handling **non-deterministic outputs**: LLMs can produce slightly different phrasings that are semantically equivalent but not identical (e.g., "The capital of France is Paris." vs. "Paris is the capital city of France."). Exact string matching (e.g., assert output == expected) will fail here, so focus on **semantic checks** rather than lexical (word-for-word) ones.

The goal is to implement reliable assertions that verify meaning, correctness, relevance, or other criteria, even with variations in wording.

Core Requirements

1. **Basic Setup**
    - Use .NET/C# as the primary language (easy for students, rich ecosystem).
    - Integrate with at least one LLM provider (e.g., OpenAI's GPT models via openai library; bonus for supporting others like Grok or local models via Ollama).
2. **Test Case Structure**
    - Prepare a dataset of 100–200 test cases (JSON).
    - Each test case includes:
        - input: User query or variables for the prompt.

3. **Running Tests**
    - For each test case:
        - Format and send the prompt to the LLM.
        - Get the output.
        - Run assertions.
        - Report pass/fail, with explanations (e.g., score and reasoning).
    - Support running multiple times per test (e.g., 3–5 runs) to account for variability, and aggregate results (e.g., pass if ≥80% succeed).
4. **Focus: Semantic Assertions**
    - **Avoid**: Simple string equality or basic metrics like BLEU/ROUGE (they penalize valid paraphrases).
    - **Implement at least two semantic approaches**:

Approach 1: Embedding-Based Semantic Similarity (Recommended Primary Method)

- Use sentence embeddings to compute cosine similarity between the generated output and a reference (expected) output.
- Assertion: Pass if similarity score ≥ threshold (e.g., 0.85–0.9; tune based on your tests).
- Use OpenAI and Ollama (optionally)

Approach 2: LLM-as-a-Judge (For More Nuanced/Custom Criteria)

- Use a second LLM to evaluate the output.
- Prompt it to score or judge based on custom criteria (e.g., correctness, relevance, tone).
- Techniques:
    - Simple: Ask for a yes/no or score 1–10.
    - Advanced: Use G-Eval style (chain-of-thought reasoning + form-filling for better accuracy).
- Assertion: Pass if score ≥ threshold (e.g., 8/10) or binary "correct".
- Example prompt for judge:

text

Evaluate if the following response correctly answers the query and matches the expected meaning.

Query: {input}

Expected: {expected_output}

Actual: {generated_output}

Criteria: {criteria}

First, reason step-by-step. Then, output ONLY a score from 1-10.

   - **Hybrid**: Combine both (e.g., embedding for quick filter, LLM-judge for edge cases).

5. **Evaluation and Reporting**
   - Output a summary table (e.g., using pandas or simple print): | Test Case | Pass/Fail | Similarity Score | Judge Reasoning |
   - Visualize: Plot similarity scores or pass rates.
   - Handle non-determinism: Run tests with temperature=0 for consistency where possible.

ML 25/26-09 Implement Tools Routing and Argument mapping Benchmark for Foundry Local

Microsoft Foundry Local is an on-device AI inference solution that provides performance, privacy, customization, and cost benefits. It integrates with your agents and applications through a CLI, SDK, and REST API. It enables you to run models locally on almost any device.
For more information please see: https://learn.microsoft.com/en-us/azure/ai-foundry/foundry-local/what-is-foundry-local

Your task is to implement an agent based on the Microsoft Agent Framework that integrates multiple plugins exposed as tools. These plugins must be correctly registered and discoverable by the agent so they can be invoked during benchmark execution.

The benchmarking application will run the agent, load evaluation prompts from files, execute chat completion requests, capture the model responses, and automatically assess the results according to predefined evaluation metrics.

Below are practical, benchmark-ready ideas to evaluate tool calling- and argument mapping-capabilities in LLMs.

1. Tool Selection & Routing Accuracy

Benchmark Idea: *Tool Choice Precision*

Goal: Can the model choose the *right* tool among many?

Setup

- Provide 5-10 tools with overlapping descriptions
- Include decoy tools with similar names or partial functionality

Example Tasks

- "Get today's weather in Berlin"
  → should call get_weather, not search_web
- "Summarize this PDF"
  → should call pdf_parser, not text_summarizer

1. Failure Modes

- Wrong tool selected
- Hallucinated tool
- Multiple tools called when only one is required
- Tool called when none is needed

2. Metrics

- Exact Match Accuracy

- False Positive Rate
- Hallucinated Tool Rate

Score

Following is an idea how to calculate accuracy. Feel free to find other ways to calculate the accuracy.

```
Tool Selection Accuracy = correct_calls / total_tasks
```

2. Argument Construction & Schema Compliance

Benchmark Idea: ***Parameter Fidelity***

Goal: Can the model generate valid, precise arguments?

Setup

- Tools with strict JSON schemas
- Required, optional, nested, and enum fields

Example Task

```
{
  "tool": "book_flight",
  "required": ["origin", "destination", "date"],
  "optional": ["seat_class"]
}
```

User:

"Book me economy from Frankfurt to Seattle next Monday"

Metrics

- Schema validity (JSON parse success)
- Correct field mapping
- No extra or missing parameters

Note, the solution should be designed to be able to also use a common cloud model instead of the Foundry.

ML 25/26-10 Implement Model Multistep Planning Benchmark for Foundry Local

Microsoft Foundry Local is an on-device AI inference solution that provides performance, privacy, customization, and cost benefits. It integrates with your agents and applications through a CLI, SDK, and REST API. It enables you to run models locally on almost any device.
For more information please see: https://learn.microsoft.com/en-us/azure/ai-foundry/foundry-local/what-is-foundry-local

Your task is to implement the Agent by using Microsoft Agent Framework that makes a usage of several plugins (3-5), that together can build mor complex workflow. You are required to design and implement plugins that are registered as tools withing agent and be used for benchmark tests.

Below are practical, benchmark-ready ideas to evaluate the multistep planning LLM capabilities.

What it measures
Whether an LLM can decompose a single user goal into multiple dependent tool calls, execute them in the correct order, and pass outputs from earlier tools as inputs to later ones without unnecessary or missing steps.

Why this benchmark matters
Multi-step planning is what separates **tool-using chatbots** from **true agents**. Models that score well here:

- Generalize better to unseen workflows
- Waste fewer API calls
- Are safer in production environments

Core abilities tested

- Task decomposition
- Planning and sequencing
- Data dependency handling
- Avoidance of redundant tool calls

Input

- One high-level user request
- A registry of tools (with schemas)
- Deterministic or simulated tool outputs

Expected Output

- An ordered sequence of tool calls
- Correct parameter usage

- Correct flow of intermediate results

Typical Failure Modes

- Wrong order of calls
- Missing a required step
- Using invented intermediate data
- Over-calling tools “just in case”

Key Metrics

- Sequence correctness (exact match or edit distance)
- Dependency correctness (are inputs sourced correctly?)
- Efficiency (number of calls vs. minimum required)

Examples

Example 1: Travel Planning

User goal

“Find a flight to Seattle next Monday and book a hotel near the conference center.”

Correct tool plan

1. search_flights(origin, destination, date)
2. select_best_flight(flight_results)
3. search_hotels(location, dates, radius)
4. book_hotel(hotel_id)

Common errors

- Booking hotel before dates are known
- Skipping flight selection
- Using hard-coded dates instead of tool output

Example 2: Data Analysis Workflow

User goal

“Analyze last quarter’s sales and generate a chart.”

Correct tool plan

1. load_sales_data(period="Q4")
2. aggregate_sales(by="month")
3. generate_chart(data, chart_type="bar")

Failure cases

- Calling generate_chart without aggregation
- Recomputing data unnecessarily
- Using incorrect time range

Example 3: Customer Support Automation

User goal

“Check this order status and notify the user if it’s delayed.”

Correct tool plan

1. get_order(order_id)
2. check_delivery_eta(order_data)
3. send_notification(user_id, message) *(conditional)*

Evaluation note
The benchmark checks whether the notification tool is **conditionally invoked**, not always.

Example 4: Document Processing

User goal

“Summarize this PDF and translate it to German.”

Correct tool plan

1. extract_text_from_pdf(file)
2. summarize_text(text)
3. translate_text(summary, target_language="de")

Common mistakes

- Translating before summarizing
- Skipping text extraction
- Translating the full document instead of the summary

Example 5: DevOps Task

User goal

"Check if the service is down and restart it if needed."

Correct tool plan

1. check_service_health(service_name)
2. restart_service(service_name) *(only if unhealthy)*

Key evaluation point

Conditional branching based on tool output.

Note, the solution should be designed to be able to also use a common cloud model instead of the Foundry.

ML 25/26-11 Implement Reminder Agent

1. Goal and Scope

- Students shall implement an Agent using the Microsoft Agent Framework in .NET/C# that enables users to save (“remember”) and later retrieve assets (e.g., restaurants, hiking routes, travel ideas, books, playlists, contacts, specs).
- The system shall support natural-language prompts for creating reminders, searching reminders, and attaching photos to saved assets.

2. Core Functional Requirements

1.1 Reminder Creation

- **The agent shall accept user prompts such as:**
  - "Remind me of the restaurant 'twelfe apostels' in Frankfurt Ginnheim."
  - "Remind me on hiking route 'east mallorca cala de maar'"
  - "Remind me on travel destination idea 'Island'"

  The tool should selectively grab tags/categories from the prompt like 'restaurant' in this case. Anything can be reminded and categorized. Later on, user can search for assets:

  - "show me sommer travel destinations" "
  - show me travel destinations I sent you last year"
  - "show me tapas bars in spain"
  - "show me the book I remembered last two months"
  - "which books I have on the list"
  - "Remember the playlist 'XISCO Winter24'"
  - "Save contact Hannes Mustermann tel: 01633546674" "Remember SDK spec-kit"
- **The agent shall extract and store:**
  - Asset title/name (e.g., “twelve apostels”)
  - Category/tag(s) (e.g., restaurant, hiking route, travel destination, book, playlist, contact, spec)
  - Optional attributes derived from the prompt (e.g., location “Frankfurt Ginnheim”, country “Spain”, phone number)
  - Timestamp (creation date/time)
  - Embeddings should be created for major attributes. It is you task to evaluate which attributes should be embedded.

2.2 Asset Search and Listing

- The agent shall support retrieval queries, including:

    - Filtering by category/tag (e.g., “tapas bars”, “books”)
    - Filtering by time (e.g., “last two months”, “last year”)
    - Filtering by free-text keywords (e.g., “summer travel destinations”). In this case embeddings should be used.
- The agent shall return results in a readable format (e.g., list with name, category, and key metadata).

2.4 Photo and location Association

- The agent shall support attaching uploaded photos to existing assets:
    - Example: “Add this photo to the restaurant ‘…’”
- The system shall store a reference to the photo (e.g., file path or id) linked to the asset.
- If the asset cannot be uniquely identified, the agent shall request clarification.
- If the asset is unknown, the agent shall respond with a suitable error message (e.g., “I do not know this restaurant.”).
- Location should be associated too: “Restaurant ‘abc’ is in Frankfurter Landstrasse 204, Frankfurt am Main”. City should be saved as city, so the user can search by city. Also wider areas and locations like ‘Germany’, “East Bosnia” should be saved as locations.

3. Tool/Plugin Requirements

- **The solution shall provide plugin functions (tools) with clearly defined arguments, including at minimum:**
    - CreateAsset(name, category, tags?, metadata?)
    - SearchAssets(query, category?, dateRange?)
    - ListAssets(category?)
    - AttachPhoto(assetIdentifier, photoReference) as BASES64 encoded
    - GetAssetDetails(assetIdentifier)
- **Argument mapping from user prompts to tool function parameters shall be implemented and demonstrated with examples.**

4. Storage and Persistence Requirements

- All stored data shall be persisted to a CSV file.
- The storage layer shall be modular using:
    - An interface (e.g., IStorageProvider)
    - Dependency Injection (DI) to select the implementation
- **The CSV schema shall support:**

- Unique asset id
- Name/title
- Description
- Category
- Tags
- Metadata (key-value or defined columns)
- Created/updated timestamps
- Photo references (one-to-many supported)

See: (Deven Shah, Pinak Ghate, Manali Paranjape, Amit Kumar, 2018)