ChatGPT

DeepAgents for MCP-Based Retrieval & Evaluation

Section 1 — Findings

- **DeepAgents Architecture:** LangChain's **DeepAgents** introduce four core components to enable long-horizon reasoning: a **planning tool**, **sub-agents**, a **persistent scratchpad (virtual file system)**, and a **detailed system prompt** [1]. This design, inspired by apps like Claude Code and Manus, lets an agent break down complex tasks, delegate to specialized sub-agents, store intermediate results, and follow extensive guidelines beyond a basic ReAct loop [2].
- **Planner & File-System Tools:** Every deep agent has a built-in **"write_todos"** planner tool to generate a step-by-step TODO list (plan) and a suite of file operations (`write_file`, `read_file`, `edit_file`, `ls`) for note-taking and state persistence [2]. The planning tool helps the agent outline its approach without immediately executing, while the virtual file system serves as a scratchpad to offload and recall information between steps [3] [4]. These file tools are mocked in memory (no real disk writes) to allow safe concurrent agents and avoid state conflicts [4].
- **Sub-Agents:** DeepAgents can spawn **sub-agents** – essentially "shallow" agents with confined scope – to handle specific subtasks in isolation [5]. Each sub-agent has its own prompt and (optionally) a restricted toolset, preventing it from straying beyond its domain. Under the hood, a deep agent is a **hierarchy of agents**: the top-level agent can call a sub-agent as if it were a tool, getting back a result [5]. This design provides context quarantine, ensuring (for example) a coding sub-agent doesn't mix its context with a research sub-agent's data.
- **Prompting & Model:** DeepAgents come with a **built-in system prompt** (inspired by Claude's Code Interpreter prompt) that hard-codes tool usage policies and step-by-step format [6]. Developers supply an additional instruction string to define the agent's role or expertise. By default the Python package uses Anthropic's Claude-2 100k ("claude-sonnet-4") as the LLM for optimal performance [7] [8]. However, you can override the model (and even per-sub-agent model settings) or limit which built-in tools are enabled [8] [9]. Notably, weaker models might struggle with the verbose prompt and tool schemas – as observed when using a 20B open model, which led to output validation errors until switching to Claude [7].
- **API and Usage:** In Python, DeepAgents are created via `create_deep_agent(tools, instructions, subagents=...)` from the `deepagents` package. This returns a LangChain **agent** (built on LangGraph) that you can interact with just like any other chain or agent [10]. You register tools as Python functions (or LangChain `@tool` objects) – for example, an `internet_search(query)` function – and pass them in a list [11] [12]. The agent will include these alongside its default tools. DeepAgents support streaming outputs, memory, and even insertion into LangSmith for tracing since they conform to LangChain's standard agent interface [10]. (A TypeScript/JS version of DeepAgents is also available, with similar concepts and API [13], though the Python implementation is the primary focus for now.)

Minimal Example: Below is a minimal Python snippet that creates a DeepAgent with a custom web search tool and runs it on a user query. The agent is instructed to act as an "expert researcher" and will plan its steps, use the `internet_search` tool as needed, and produce a final report answer:

```python
import os
from deepagents import create_deep_agent

# Define a simple internet search tool (using a hypothetical client or API)
def internet_search(query: str) -> str:
    """Search the web for the query and return snippets."""
    # In practice, call an API like Tavily/Bing here (omitted for brevity)
    return "[Search results for '%s'...]" % query

# System instructions for the agent
research_instructions = """You are an expert researcher.
Your job is to conduct thorough research, then write a polished report.
You have access to an `internet_search` tool for web queries and a file system
for notes.
"""

# Create the deep agent with the tool and instructions
agent = create_deep_agent(
    tools=[internet_search],
    instructions=research_instructions
)

# Invoke the agent on a user question
result = agent.invoke({"messages": [{"role": "user", "content": "What is
LangChain's LangGraph?"}]})
print(result['output'])
```

(This example uses a placeholder search function – in a real setup, you'd integrate an API client and possibly enable the agent's built-in file tools for note-taking. See the official deepagents README for a more complete research agent example [14] [15] *.)*

Section 2 — MCP Interface (Tools vs Resources)

MCP Background: Anthropic's **Model Context Protocol (MCP)** defines a standardized way for AI agents to discover and use external capabilities via a **dual interface** of **Tools** and **Resources** [16] [17]. In MCP terms, **Tools** are actions or functions the model can invoke (like API calls with JSON inputs/outputs), whereas **Resources** are read-only data elements the model or host can fetch as additional context [18] [17]. This aligns with the CQRS (Command Query Responsibility Segregation) pattern: *commands* change state or perform operations, and *queries* retrieve data.

For our DeepAgents integration, we propose an MCP schema exposing the agent's abilities as **Tools (commands)** and the retrieval/file data as **Resources**:

- **MCP Tools (Commands):**
- `agent.research(question, strategy="auto")` – Performs an end-to-end research task. The agent will plan steps, use the appropriate retrieval strategy, maybe consult sub-agents, and return a

comprehensive answer (report). The optional `strategy` parameter can override how retrieval is done (see Section 3) or "auto" lets the agent decide.

- `agent.plan(goal)` – Uses the agent's planner to break down a goal or complex query into a list of TODO steps (without executing them). This returns a task list (which might be written to the agent's scratchpad as well).
- `agent.execute(step)` – Executes a single to-do item or instruction. This could involve calling a sub-agent specialized for that step or just using a tool. It allows an external orchestrator or UI to step through the agent's plan item by item (e.g. review intermediate results or provide feedback between steps).

- *(Optionally, additional commands like* `agent.reflect()` *or* `agent.summarize()` *could be exposed. For instance,* `agent.summarize()` *might compile the contents of the workspace files into a final report. However, the three above cover the core loop: plan → act → iterate.)*

- **MCP Resources (Queries):**

- `retriever://{strategy}/{query}` – A resource endpoint for **retrieval results**. When the host or agent reads this URI, the MCP server will run the specified retrieval `strategy` on the `query` and return the top documents or snippets (without the agent having to explicitly call the search tool). This is effectively a read-only operation that fetches knowledge. It feels like passive data access (search is retrieving data) but is implemented in our system as a resource for consistency – meaning the *host* can decide when to pull in these results [17]. For example, an agent could list available resources or the host could pre-fetch `retriever://bm25/virus origin` to supply the agent with context.
- `workspace://files/{path}` – A read-only resource to fetch the content of the agent's scratchpad files. Since DeepAgents keeps an internal virtual filesystem for notes, this resource lets us retrieve those notes or outputs by filename. For instance, `workspace://files/todo.txt` might return the agent's TODO plan, or `.../research.txt` could return a draft report the agent wrote. Exposing files as MCP resources allows external viewers or evaluators to access the agent's work products without invoking new tools. (Write operations remain tools – e.g. the agent uses its `write_file` tool internally to modify the workspace.)

Tool/Resource Distinction: Tools are invoked by the AI model via function calls, while resources are fetched by the host or via special read requests. In practice, a "search" might be implemented as both: the agent can call a **Search Tool** for active querying, and the host could also provide relevant docs via a **Resource URI**. This dual approach means our agent can operate in two modes: as a fully agentic researcher calling tools on its own, or in a retrieval-augmented generation (RAG) mode where the system supplies context from resources (more on this hybrid in Section 3 and 4).

MCP Configuration: To integrate with MCP, we will run dedicated MCP servers for our tools and resources, and use LangChain's MCP client adapters to expose them to the agent (or other clients). For example, we might run an MCP server for the **retrieval tools** and one for the **agent commands**. Below is a conceptual configuration using LangChain's `MultiServerMCPClient` to connect to two MCP endpoints (one local via stdio, one HTTP):

```
from langchain_mcp_adapters.client import MultiServerMCPClient
# Load .env with API keys (e.g., OPENAI_API_KEY, etc.) and any server URLs
```

```python
# (Ensure secrets like keys are stored in environment variables, not hard-coded)

client = MultiServerMCPClient({
    "retrieval": {
        "transport": "stdio",           # Launch local MCP server as subprocess
        "command": "python",
        "args": ["./mcp_retrieval_server.py"]  # This server provides
retriever:// and maybe tools
    },
    "agent": {
        "transport": "streamable_http",  # Connect to an HTTP MCP server
        "url": "http://localhost:8000/mcp"    # e.g., a deployed deep agent
server
    }
})
tools = await client.get_tools()  # Discover all tools exposed by those servers
```

In this snippet, the *retrieval* MCP server could host our `retriever://` resource (and perhaps a search tool), while the *agent* MCP server might host the `agent.research/plan/execute` commands. The `MultiServerMCPClient` aggregates both, so the LangChain agent (or any MCP-compatible client like Claude) can use tools from multiple servers seamlessly [19] [20]. We choose `stdio` transport for local tools (quick to spin up, communicates via stdin/stdout pipes) and **HTTP/SSE** for long-running or remote services [21]. This separation also aids sandboxing: for instance, the retrieval server could be containerized with access to a database, while the agent server runs with only the LLM and virtual FS.

Security & Sandboxing: MCP's design allows the host to enforce security. We will restrict our MCP servers to only the intended tools/resources. The **virtual file system** in DeepAgents is already a sandbox (files are not actual OS files) [4], and our MCP servers will not expose any shell or unrestricted file access. Tools that perform external actions (like web search or code execution) can be isolated in separate processes or require user confirmation, depending on the deployment. By running retrieval as its own MCP service, we ensure the agent can only retrieve data through controlled queries (no direct database access outside what we expose). Authentication for external APIs (e.g., OpenAI, Tavily) is handled via environment variables and passed to those services – these secrets are not exposed to the LLM, and any MCP server calling them will be configured with key-based access (e.g., loading `OPENAI_API_KEY` from the environment). This separation of concerns (AI agent vs. tool servers) follows MCP's principle of **granular permissions** [22] [17], so each server only does what it's intended to, and the host orchestrates the overall workflow.

Section 3 — Retrieval Strategies Integration

To provide flexible information retrieval for the agent, we plan to implement a **RetrieverFactory** that supports multiple search strategies under one interface. This allows the agent (or the system) to choose the best retrieval approach for a query. The strategies we will include are:

- **Naive** – a default straightforward retriever (e.g. a simple vector similarity search on embeddings without any advanced tuning). This is the baseline semantic search: given the query, return top-k documents by cosine similarity. It's "naive" in that it doesn't use any query reformulation or hybrid scoring. Latency is low (one embedding lookup) but it may miss relevant docs if the query wording is poor.

- **BM25 (Sparse)** – a classical keyword search using BM25 or similar lexical scoring. This strategy excels at precision for keyword-heavy queries (it finds documents containing the query terms) and is very fast (especially with an indexed engine or in-memory BM25) [23]. However, it might fail to retrieve semantically relevant documents that don't share exact terms (lower recall). We'll likely use an existing BM25 implementation (e.g., from `rank_bm25` or an IR library) on a pre-indexed corpus.
- **Vector (Dense)** – a semantic vector similarity search using embeddings. We will embed the query and compare to document embeddings (using a vector store like FAISS or Qdrant). This finds conceptually related docs even if vocabulary differs, improving recall. The trade-off is a bit more latency (embedding the query, plus nearest-neighbor search) and reliance on a good embedding model. We'll cache embeddings for common queries and use a local vector index for speed (to meet our <2s retrieval goal).
- **Multi-Query Expansion** – a strategy where we generate multiple nuanced queries from the original question to cast a wider net. The agent (or a behind-the-scenes process) uses an LLM to paraphrase or generate related queries, then performs a search for each and merges the results. This can dramatically increase recall (finding aspects of the answer the single query missed) [24] [25], at the cost of extra LLM calls and multiple searches (higher latency). We'll limit the number of query variants (e.g., 3) and perhaps do this only for ambiguous or broad questions. Implementation: e.g., use GPT-4 or Claude to produce alternate queries, then feed those to the vector/BM25 retrievers.
- **Re-rank** – a two-step strategy to boost precision. First, retrieve generously (maybe top 10-20 via a baseline method like vector search), then use an LLM to re-rank or filter those results based on relevance to the query. For example, we can prompt an LLM to score each candidate passage for how well it answers the question, and then return the top few. This yields higher answer precision and relevance [26] but with additional LLM overhead. We'll use this for cases where we want the absolute best context at the expense of latency (perhaps as part of "auto" strategy when the initial results are noisy). Caching can mitigate cost: if the same doc set is retrieved for similar queries, reuse the prior LLM judgments.
- **Parent-Document Retrieval** – a strategy to ensure broader context is retrieved. If our documents are chunked (as is common for vector stores), this method will fetch entire source documents when any of their chunks are deemed relevant. In practice, this might mean after getting top chunk results from another retriever, we pull in all chunks from those documents (or at least a summary of the rest of each document). The benefit is higher recall of supporting info (you don't miss context just because it was in an unretrieved chunk of the same doc), at the risk of introducing more irrelevant text (lower precision). This strategy is useful for questions that require understanding a full document's content (e.g., story or lengthy article questions). Implementation-wise, we will store a document-id with each chunk in the index; parent_doc strategy will gather by id.
- **Ensemble (Hybrid)** – a fusion of multiple retrievers' results, using **Reciprocal Rank Fusion (RRF)** or similar to produce a unified ranking [27] [23]. Typically, a hybrid search combines sparse (BM25) and dense (vector) scores, leveraging both keyword overlap and semantic similarity [23]. RRF works by taking the rank of each document in each list and giving a score like `1/(rank + C)` (we'll choose C=60 per convention) and summing these across retrievers [28] [29]. This tends to improve overall recall and robustness: documents that are moderately highly ranked in both lexical and semantic lists will bubble up to the top [30]. Our ensemble strategy will likely combine BM25 and vector (and possibly others like multi-query results) to balance their strengths. The latency is the sum of the individual retrieval calls (so ~2x a single search), but still manageable (well under 2s in local tests for

moderate corpus size). We'll also support weighting if needed (e.g., if vector tends to overshoot, weight BM25 higher) [27] .

All these strategies will be accessible through a single interface so that the agent can invoke them via one tool (e.g., a unified `search` tool that takes a `strategy` parameter). DeepAgents expects tool calls to be atomic and self-contained – the model will issue one function call and get back a result. Therefore, any multi-step logic (like multi-query expansion or ensemble merging) will happen *inside* the tool implementation rather than through multiple agent steps. This is important: the agent's prompt and planning can remain high-level ("search for X using strategy Y"), and our tool function will handle the complexity internally. For example, if using the **multi_query** strategy, the agent might call `search(query="ABC", strategy="multi_query")` once, and that tool will internally prompt an LLM to create sub-queries, run them, and return a collated result – all invisible to the agent. Keeping this orchestration in the tool ensures the agent doesn't get confused by intermediate results and also simplifies traceability (one function call spans the sub-operations).

RetrieverFactory Design: We will implement a `RetrieverFactory` class (or module) that initializes the necessary retrieval backends and provides a method to execute a search with a given strategy. For instance, it might look like:

```
# artifacts/retriever_factory.py (simplified snippet)
from langchain.retrievers import EnsembleRetriever, BM25Retriever
from langchain.vectorstores import FAISS
from langchain.embeddings import OpenAIEmbeddings

class RetrieverFactory:
    def __init__(self, docs):
        # Initialize retrievers (this could load indexes from disk for speed)
        self.bm25 = BM25Retriever.from_texts(docs)        # BM25 index in memory
        embedding = OpenAIEmbeddings()                    # or local embedding
model
        self.faiss = FAISS.from_texts(docs, embedding)    # Vector index
        # Ensemble retriever combining BM25 and vector (weights can be tuned)
        self.hybrid = EnsembleRetriever(retrievers=[self.bm25,
self.faiss.as_retriever()], weights=[0.5, 0.5])
        # (More complex strategies like multi_query will be handled in methods)

    def retrieve(self, query: str, strategy: str = "auto"):
        strategy = strategy or "auto"
        if strategy == "auto":
            # For "auto", we can define a heuristic: e.g., try hybrid first
            return self.hybrid.get_relevant_documents(query)
        elif strategy == "bm25":
            return self.bm25.get_relevant_documents(query)
        elif strategy == "vector":
            return self.faiss.as_retriever().get_relevant_documents(query)
        elif strategy == "ensemble":
            return self.hybrid.get_relevant_documents(query)  # same as hybrid
```

```
        elif strategy == "multi_query":
            return self._multi_query_search(query)
        elif strategy == "rerank":
            return self._rerank_search(query)
        elif strategy == "parent_doc":
            return self._parent_doc_search(query)
        else:
            raise ValueError(f"Unknown strategy: {strategy}")

    def _multi_query_search(self, query):
        # 1. Use an LLM to generate N related queries
        sub_queries = generate_alternative_queries(query)
        # 2. Perform vector search for each sub-query
        results = []
        for q in sub_queries:
            results.extend(self.faiss.as_retriever().get_relevant_documents(q))
        # 3. Merge and deduplicate results (could also use EnsembleRetriever on
results sets)
        merged = fuse_results(results)  # e.g., rank by frequency or max
similarity
        return merged[:5]  # top 5 merged results

    def _rerank_search(self, query):
        # First get initial docs (vector search)
        docs = self.faiss.as_retriever().get_relevant_documents(query)
        # Use LLM to score each doc's relevance to the query
        scored = [(doc, relevance_score(query, doc)) for doc in docs]
        scored.sort(key=lambda x: x[1], reverse=True)
        return [doc for doc, score in scored[:5]]

    def _parent_doc_search(self, query):
        # Get top chunks via vector search
        docs = self.faiss.as_retriever().get_relevant_documents(query)
        # Fetch all chunks from the same source documents
        doc_ids = {d.metadata['source_id'] for d in docs}
        full_docs = [get_full_document(doc_id) for doc_id in doc_ids]
        return full_docs
```

(This is a conceptual outline — in practice we'd handle things like ensuring unique results in multi_query, efficient doc merging, and maybe use LangChain's utilities for multi-vector search. Also, the `OpenAIEmbeddings` *and LLM calls assume API keys configured via environment.)*

The **RetrieverFactory** would be initialized once (loading indexes into memory or establishing DB connections) and then used by the agent's tools. We might register a single LangChain tool called `"retrieve"` or `"search"` that calls `RetrieverFactory.retrieve(query, strategy)` and returns the text of the top documents. For example, the tool function given to DeepAgent could be:

```python
def retrieve_docs(query: str, strategy: str = "auto") -> str:

"""Retrieve relevant documents using the specified strategy and return their
content."""
    docs = retriever_factory.retrieve(query, strategy)
    # Join the results into one string (could also return a structured list)
    return "\n\n".join([d.page_content for d in docs])
```

By returning raw content (or perhaps a summarized version of each doc), the agent can read and reason over it. We will include clear descriptions in the tool's docstring so the agent knows when to use each strategy. For instance, the tool description might say: *"Use this tool to search the knowledge base.* `strategy` *can be 'bm25', 'vector', 'multi_query', etc. Default 'auto' will combine strategies for best results. 'bm25' is good for exact keywords, 'vector' for semantic matches, 'multi_query' for broad questions, etc."* Providing these hints guides the LLM's choice. However, given the complexity, an **automatic strategy** might often be preferred – we can implement `strategy="auto"` to do a quick hybrid search and then decide if further steps are needed (perhaps the agent itself can decide to call retrieve again with a different strategy if results seem insufficient).

Latency & Caching: Each strategy has a performance profile. We aim to keep **single retrieval calls under ~2 seconds**. BM25 and vector searches are usually tens of milliseconds for moderate corpora, so even ensembles are sub-second. The heavier parts are embedding generation and LLM calls (in multi_query or rerank). We mitigate this by caching: embed the corpus in advance; cache embeddings for repeated queries; and memoize LLM outputs for query expansions or reranks if the same query appears again. We'll also tune `k` (number of results) per strategy to balance thoroughness vs. speed – e.g., multi_query might retrieve fewer per sub-query to cap total results, and rerank might only rerank the top 10 instead of 100. With these strategies in place, the agent can retrieve information in whatever way best suits the question, all through the unified interface of our retrieval tool.

Section 4 — Evaluation & Observability

To ensure our system is delivering quality results, we will set up a **evaluation harness** focusing on retrieval-augmented generation metrics. We plan to use **RAGAS (Retrieval Augmented Generation Assessment Scores)** – an open-source framework for evaluating RAG pipelines – integrated with LangChain's evaluation tools [31] [32]. The key metrics we'll measure are:

- **Answer Relevancy** – Is the agent's answer actually answering the asked question, and how directly? This checks if the answer stays on topic and addresses the query. RAGAS measures this by seeing if the answer could be a plausible answer to the question (using an LLM to compare the answer against the question) [26]. A high score means the answer is relevant and focused; a low score indicates it's off-target or incomplete.
- **Context Precision (Context Relevancy)** – How much of the retrieved context was necessary for the answer (signal-to-noise ratio). This metric looks at the content the agent pulled in (source documents) and identifies the fraction of that content which was actually used to produce the answer [33]. For example, if the agent retrieved 5 passages but only one contained facts used in the answer, precision is low. We want the agent to retrieve mostly relevant info. (In RAGAS, this was called *context_relevancy*; we interpret it as precision – we want a small, pertinent context.)

- **Context Recall** – Did the retrieval miss anything critical from the ground truth? This metric requires a known correct answer (ground truth) for the question. It checks whether all the facts in that ground truth answer can be found in the retrieved documents [34]. Essentially, it measures the retriever's coverage. A high context recall means the agent had all the necessary info available; a low score means something was absent (so the answer might be incomplete or unsupported).
- **Faithfulness** – Is the answer supported by the retrieved evidence, and free of hallucinations? This measures factual consistency between answer and sources [35]. RAGAS computes this by extracting statements from the answer and verifying each against the provided context docs [35]. A faithfulness score of 1.0 means every claim in the answer can be found or verified in the retrieved content. This is crucial for trustworthiness – even a relevant answer should not introduce new unsupported facts.

Using these metrics, we can quantitatively evaluate each QA pair handled by our system. We will create a **"golden set"** of around 10–20 question-answer pairs for testing. This set should include diverse queries (some straightforward fact-based, some requiring synthesis or multi-hop reasoning) along with either a ground-truth answer or reference documents. For instance, we might include: *"How did New York City get its name?"* with the ground truth answer about the Duke of York [36], or *"Which borough of NYC has the highest population?"* with answer "Brooklyn" [37]. Each example in the dataset will be a JSON record containing the **query** and **ground_truth** answer(s). (If we don't have a single ground truth answer, we can provide a few key facts expected in a correct answer to evaluate recall and faithfulness.)

Our **evaluation harness** (script `eval_harness.py`) will do the following: for each example in the golden set, run the agent (or just the retrieval + LLM chain) to get an answer and the supporting docs it used, then compute the metrics. We'll leverage `RagasEvaluatorChain` from the `ragas` library for convenience [38]. LangChain provides an integration where you can wrap RAGAS metrics in evaluator chains and call `.evaluate(examples, predictions)` to get a report [39]. For a simple implementation, we might just call each metric's function directly on the result. For example:

```
from ragas.metrics import faithfulness, answer_relevancy, context_precision,
context_recall

results = []
for ex in golden_examples:
    query = ex["query"]
    true_answer = ex.get("ground_truth")
    # 1. Run the agent or QA chain on the query
    output = agent.research(query)  # (this would use our agent tool via MCP or
directly)
    answer_text = output["answer"]
    source_docs = output["source_documents"]  # the retrieved docs the agent saw
    # 2. Prepare result dict as expected by RAGAS (containing question, answer,
docs, and ground truth)
    result = {
        "question": query,
        "answer": answer_text,
        "context": [doc.page_content for doc in source_docs],
        "ground_truths": [true_answer] if true_answer else []
    }
```

```
    # 3. Compute metrics
    scores = {
        "faithfulness": faithfulness(result)["faithfulness_score"],
        "answer_relevancy": answer_relevancy(result)["answer_relevancy_score"],
        "context_precision": context_precision(result)
["context_precision_score"],
        "context_recall": context_recall(result)["context_recall_score"],
    }
    results.append({**result, **scores})
```

We will output the evaluation results in a JSONL or CSV file for analysis. Each line/row will include the query, our answer, and the scores for each metric (0 to 1). For instance, a sample output line (in JSON) might look like:

```
{
  "question": "Which borough of New York City has the highest population?",
  "answer": "According to the 2020 census, Brooklyn has the highest population
of NYC's boroughs.",
  "ground_truths": ["Brooklyn"],
  "faithfulness": 1.0,
  "answer_relevancy": 0.95,
  "context_precision": 0.8,
  "context_recall": 1.0
}
```

This would indicate the agent's answer was correct ("Brooklyn"), supported by sources (faithfulness 1.0), very relevant, and it retrieved the needed info (recall 1.0) with some extraneous content (precision 0.8 meaning maybe it retrieved more than it ultimately used). These metrics help us pinpoint issues: e.g., low context_recall suggests the retriever missed something (so maybe we need multi_query or bigger k), low faithfulness suggests the agent is hallucinating beyond the docs (maybe strengthen the prompt or retrieval quality), low answer_relevancy could mean the agent misunderstood the question (prompt issue or needing better plan), and low precision (context_precision) means we should improve how targeted the retrieval is (maybe via re-ranking or filtering out irrelevant docs).

Telemetry & Traces: Throughout the agent's operation, we will use **LangSmith** (LangChain's observability platform) to capture traces and tool usage. By setting `LANGSMITH_TRACING=true` and using the LangChain integration, each agent run will log a trace with structured data [40] [41]. We expect to see spans for the agent's planning step, each tool call (with tool name and input), and the final answer production. We will enrich these traces with custom tags for deeper analysis. For example, when our `retrieve_docs` tool is called, we can attach metadata like `strategy: bm25` and `num_results: 5` to that span. We'll also measure latency of each step and token usage; LangSmith can capture token counts per LLM call automatically, and we can log timing info via callbacks.

If needed, we can also integrate **OpenTelemetry** to export traces to a viewer like Jaeger or use **Phoenix** (an evaluation dashboard) – but LangSmith's UI should suffice for our needs. The idea is that after running our golden set through the agent, we not only get the RAGAS metrics but also have a click-through trace of each question's reasoning process. In LangSmith or a similar tool, we can inspect a single run: see the model's plan, which tools it called in what order, how long each took, what it retrieved, and how it formulated the answer. This is invaluable for debugging. For instance, if a particular question got a low faithfulness score, the trace might show the agent skipping the retrieval step or using the wrong tool. We can then refine the prompts or tool descriptions accordingly.

We will also take advantage of LangSmith's **evaluation integration**: we can log our RAGAS metrics back to LangSmith by creating a custom evaluator or using the `RagasEvaluatorChain` connected to a LangSmith dataset [42] [43]. This allows us to visualize metric distributions across runs and track improvements over time. Each run's JSON artifact (containing question, answer, sources, scores) can be stored for further analysis or displayed in a dashboard. Ultimately, our goal is to continuously monitor these metrics in development and catch regressions – for example, if a code change or model change causes answer relevancy to drop or tool usage to go haywire, the metrics and traces will make it immediately apparent.

Section 5 — POC Plan & Checklist

Finally, to deliver this as a working proof-of-concept, here's our step-by-step plan:

1. **Environment Setup:** Prepare the development environment with required libraries and services. We will use Python 3.11+ (per LangChain's recommendation [44]) and create a virtual environment (or Docker container). Install the needed packages: `deepagents`, `langchain` (v1.x alpha), `langchain-mcp-adapters`, `mcp` (for server SDK), `ragas`, and an LLM backend (e.g., OpenAI SDK for GPT-4 or Anthropics' for Claude). Also install any vector DB clients (if using Qdrant, its Python client; or `faiss-cpu` for FAISS) and IR libraries (`rank_bm25`, etc.). Set up a `.env` file with all API keys and config values (e.g., `OPENAI_API_KEY`, `ANTHROPIC_API_KEY` if using, `TAVILY_API_KEY` if using web search API, `LANGSMITH_API_KEY` for tracing, etc.) [45] [46]. None of these secrets will be hard-coded – our code will load them at runtime (using `dotenv` or environment variables). If using local models (like Ollama or Llama 2), ensure those are installed and the model weights are available.

2. **Launching Retrieval Service:** Implement and run the **MCP retrieval server** (`mcp_server.py`, see artifact). This server will register the retrieval resource and related tools. Specifically, it will use the `mcp.FastMCP` server class to define:

3. A `@mcp.resource("retriever://{strategy}/{query}")` that calls our `RetrieverFactory.retrieve(query, strategy)` and returns the results (likely as text or as a structured list that the client can parse).
4. Possibly a `@mcp.tool("search")` as well, which could be similar to the resource but invoked as a function call. (Depending on how we integrate with the agent, the resource alone might suffice for retrieval if the agent or host knows to use it. But to allow the agent to autonomously search, defining it as a callable tool is useful – the agent can then do `tools/call` for search).
5. Ensure the server can handle multiple strategies. We'll also add a simple `workspace://` resource for file reading: `@mcp.resource("workspace://files/{path}")` that just reads the corresponding in-memory file from the agent's state (we may need to have a reference to the agent's

state; if the agent is in a different process, an alternative is to have the agent server handle file resources. It might be simpler to keep file access in the agent's own server).

6. Start the MCP server. For local dev, we can run it in stdio mode via `python mcp_server.py` (the `MultiServerMCPClient` will spawn it automatically as shown in Section 2). For a production or integrated test, we could also run it as an HTTP service (set `mcp.run(transport="streamable_http", host="0.0.0.0", port=8001)` for example).

Additionally, if we need a **vector database service** (say Qdrant or Redis for persistent storage of embeddings), we will spin that up (perhaps via Docker). In a simple POC, we can use FAISS in-memory without an external service, which keeps things self-contained. The RetrieverFactory will load the corpus on startup (for POC, maybe a small set of documents, e.g., a sample Wikipedia article or some knowledge base relevant to test questions).

1. **Launching Agent Service:** Implement the **DeepAgent MCP server** (if we choose to expose the agent via MCP). This could be as simple as wrapping our deep agent object's methods in MCP tool endpoints. For instance, in `mcp_server.py` (or another server script), after creating the deep agent, we do:

```
@mcp.tool()
def research(question: str, strategy: str = "auto") -> str:
    """Use the deep agent to research the question and return a final
answer."""

# Optionally, instruct the agent to use a particular retrieval strategy
(e.g., by writing a note or selecting tool)
    if strategy and strategy != "auto":

# We could prepend an instruction for the agent to prefer a certain
strategy
        question_prompt = f"(Use {strategy} retrieval) {question}"
    else:
        question_prompt = question
    output = agent.invoke({"messages": [{"role": "user", "content":
question_prompt}]})
    return output['output']
```

We might also expose `plan(goal)` and `execute(step)` similarly, if we want fine-grained control. For example, `plan` could call `agent.invoke` with a special prompt or utilize the `write_todos` tool directly (via `agent.plan()` if such API exists, otherwise by prompting the agent with something like "Plan only"). However, given DeepAgents doesn't have a direct public `plan()` method, we may simply prompt the agent to output a plan and not continue (or run the planning tool in isolation). In the interest of time, the POC might focus on the `research` command (full loop).

If we run this agent server as an MCP process, we'll connect to it via HTTP as in the config above. Alternatively, since we are controlling the agent in code, we might skip making the agent itself an MCP server and just call it directly in our application logic (especially if the agent and retrieval are in the same process, which is another design: deep agent can directly call RetrieverFactory without MCP overhead). For modularity, though, demonstrating MCP usage is a goal, so having at least one MCP interface (retrieval or agent or both) is desired.

1. **Test End-to-End Locally:** With the retrieval and agent components running, perform a full test of the system on a sample question. For example, run a query through the agent: if using MCP, one way is to instantiate a LangChain agent that uses `MultiServerMCPClient` to get the tools and then call `agent.run("<question>")`. However, since we already have our deep agent built, we can also simulate a user query by directly invoking `agent.research(question)` in code. Measure the timing: ensure that a simple question returns an answer in well under 8 seconds. We expect retrieval to be the minor part (<2s). If the LLM call (answer generation) is heavy, consider using a smaller model or adjusting generation parameters (e.g., limit max tokens for faster response). We should test different retrieval strategies by either hard-coding a strategy or asking a question that triggers multi-step retrieval. Verify that the agent indeed uses the retrieval tool (the trace or console logs should show the `internet_search` or `retrieve_docs` being called). Also verify that the content of the answer seems to incorporate the retrieved info (to qualitatively check that integration is working).

2. **Run Evaluation Harness:** Once the system is functioning, run our `eval_harness.py` on the golden set. This will effectively simulate each question as if a user asked it, gather the agent's answer and sources, and compute metrics. We will likely run this in a non-interactive setting (no human input) and ensure the agent doesn't ask for clarification (the prompts should discourage the agent from asking the user questions – it should ideally use tools to find info or make assumptions). The output will be a metrics report (CSV/JSON) for the golden set. We will review this to see if we meet our quality targets. For example, we'd like to see average **answer relevancy >0.8** (most answers are on point), **faithfulness ~1.0** (no hallucination allowed since answers should be from sources), **context recall high** (the necessary info is usually retrieved). If any metric is lagging, we can iterate: e.g., if context precision is low (meaning a lot of fluff retrieved), we might tighten the retriever or add a re-rank step; if faithfulness is low, ensure the agent's prompt explicitly says to only use info from files and search results (the default prompt does emphasize tool use [47]).

3. **Observability and Tuning:** With LangSmith tracing enabled, inspect a few example traces from the evaluation runs. Ensure that each tool call is properly captured. Check that our custom metadata (like strategy) is appearing. In LangSmith's UI, we'll have a timeline of actions for each question – verify the agent isn't doing extraneous steps (like calling the same search twice unless that's intended) and that sub-agents (if any) are being invoked correctly. We should also see the final answer quality. We can cross-reference the trace with the metric output: e.g., for any outlier (a case with low score), use the trace to diagnose why. All these traces can be stored under a project (we set `LANGSMITH_PROJECT="deep-agents-poc"` in .env) for future reference.

4. **Docker/Deployment (optional):** To ensure reproducibility and easy deployment, we'll create a `docker-compose.yml` that can bring up the necessary services: possibly a `retrieval_service` (running the MCP retrieval server), an `agent_service` (running the deep agent with appropriate model – note: if using open-source LLM, we might need it accessible, e.g., an Ollama server for local models or the OpenAI API for GPT-4), and an evaluation or client container. Given the time constraints and that this is a POC, we can also run everything on one machine without isolation. However, containerizing vector stores (like a Qdrant container for persistent storage) or model

servers (if any) is useful. We will also include **health checks** for these services (e.g., Qdrant's API health endpoint, MCP server responding to `/health` or simply being connectable) to know when the stack is ready.

5. **Acceptance Criteria Validation:** Finally, verify we meet the targets:

6. **Performance:** Time a typical query execution. If using GPT-4 via API, ~5-8 seconds per answer is expected (with 0 temperature). With Claude-2, maybe faster with the 100k context but still a few seconds. Retrieval should be negligible in that total if local. If any query is slower, profile where the time is spent (maybe multi_query generating too many queries, etc.) and optimize (reduce N, or cache embeddings). We can also run a quick load-test of 5-10 queries in parallel to see if any obvious bottlenecks (the MCP approach allows multi-client usage if the model calls can handle it).
7. **Quality:** Ensure that for our golden set, the answers are correct and the RAGAS metrics are satisfactory. If any are not, implement mitigations and re-run eval. For example, if **answer relevancy** is low on a couple of multi-hop questions, perhaps the agent got confused – maybe we add a few-shot example in the system prompt for those cases or break the question down differently. If **faithfulness** dips, perhaps the agent is relying on prior knowledge; we might enforce it to always cite or refer to sources (maybe by tweaking the prompt to say "if something isn't found in the files, state that it's unsure" to avoid hallucination).

8. **Traces & Logging:** Confirm that running with tracing doesn't break anything and that we can see all expected info. Any errors or exceptions should be handled (e.g., if a retrieval tool raises an error for an unknown strategy, the agent should handle it gracefully – possibly via an "I'm sorry I can't do that" message, but ideally the validation prevents it).

9. **Documentation & Artifacts:** Prepare the artifact files (as below) and a README if needed to explain how to run the POC. The code will be commented for clarity, and the environment variables required will be listed in a `.env.example`. Key configurations like model selection, server URLs, etc., will be easily adjustable. We will pin specific package versions in `pyproject.toml` or `requirements.txt` (e.g., `deepagents==0.1.x`, `langchain==1.x.x`, etc.) to ensure the behavior doesn't change unexpectedly. Any experimental features (like using LangChain v1-alpha) will be noted.

Risks & Mitigations:
- *File System Misuse:* Although deep agent tools are sandboxed, there's a risk an agent could attempt to use tools in unintended ways (e.g., write a huge file, or treat the scratchpad as long-term memory and bloat state). Mitigation: set a reasonable limit on file content size and perhaps track the number of file operations. Because the FS is in-memory, it resets per session unless we intentionally carry state over – for now, we'll treat each `agent.research` call as a fresh session (unless we want it to accumulate knowledge, which we don't in POC).
- *Infinite Loops / Long Horizons:* An agent might keep adding todos and never stop (tool misuse or an ill-defined stopping criterion). The DeepAgents framework typically has the agent stop when it's satisfied or when it runs out of steps. We will enforce a max iteration count in our agent loop (DeepAgents likely has this internally, but we can also wrap the call). For example, limit to, say, 50 tool calls – which is plenty for any reasonable query (Manus averaged ~50 in extreme cases [48]). We'll monitor if the agent gets stuck; if so, possibly the plan was too open-ended. Mitigation: improve the prompt to focus it ("Your answer should be concise and you should finish when you have enough info").

- *Tool Hallucination:* The agent might try to call a tool that doesn't exist (if it hallucinated a tool name). The system prompt of DeepAgents explicitly instructs it to only use available tools [47]. Also, by registering our tools with specific names and providing those in the prompt, we minimize this risk. If it does happen, LangChain would throw an error (unknown tool) – which we can catch and either treat as a failure or re-prompt the agent clarifying tool availability. In future, integration with an MCP registry could allow on-the-fly tool creation, but that's beyond scope – we keep the toolset fixed.
- *Model Output Format:* Since we rely on structured output (function calls) for tools, using a model like GPT-4 or Claude that natively supports function calling is important [49]. If we use an open model via LangChain's OpenAI function-calling compatibility, we must be cautious. The blog experience showed a 20B open model had trouble strictly adhering to the expected JSON, causing Pydantic errors [7]. Mitigation: for POC, lean on GPT-4 or Claude for reliability. If we must use open models, we might disable some stricter validation or simplify the schema (e.g., fewer nested fields). We also include few-shot examples in the system prompt demonstrating correct tool usage and JSON format.
- *Version Drift:* Given LangChain and DeepAgents are evolving, we'll pin versions as of this project date (e.g., `deepagents` 0.2.0, `langchain` 1.0.0aX). Our code will be tested with those versions. We'll keep a note of any known bugs (for instance, if LangSmith tracing with deepagents has issues, or if certain LangChain retriever classes are in flux). In case of issues, we have contingency plans: e.g., if the LangChain `EnsembleRetriever` isn't stable, we can implement RRF manually as shown. If the deep agent's default prompt is problematic, we can override it partially via the `instructions` or `builtin_tools` parameters (as the README suggests) to remove a troublesome tool.
- *Data Privacy:* If this system were to search the internet or company data, we must ensure no sensitive info is logged inadvertently. In our POC, we mostly use local or public data. But we will still use environment variables for any keys and not log the content of retrieved docs in traces (we can store metadata or snippet but avoid full text in third-party logs). LangSmith by default might log prompts and outputs; since our data is not sensitive, this is fine, but one could configure it to redact or not log content if needed.
- *Evaluation Bias:* RAGAS metrics themselves rely on LLMs and are not perfect or absolute. We interpret them as signals. We should be careful not to overfit to these metrics (e.g., the agent could be coerced to write answers that mirror the question to score high on answer relevancy, or to copy context verbatim to score high on faithfulness – which might degrade the user experience). To avoid this "evaluation drift," we'll use the metrics in conjunction with manual review. For example, an answer that is exactly a copy of a source might score perfectly but might not be the best format for a user. Our prompt will instruct the agent to write polished answers, and we expect a slight drop in some scores due to paraphrasing – that's acceptable as long as factuality remains high. We will treat RAGAS as a debugging tool, not the ultimate goal to game.

With this plan in place, we will proceed to implementation. The following artifacts outline key components of the POC:

artifacts/mcp_server.py – *MCP server definition for retrieval and agent tools*
artifacts/retriever_factory.py – *RetrieverFactory implementation (strategies)*
artifacts/eval_harness.py – *Script to run evaluation on golden set and output metrics*
artifacts/golden_set.jsonl – *Sample golden Q&A dataset for evaluation*

Artifacts:

artifacts/mcp_server.py:

```
import os
from mcp.server.fastmcp import FastMCP
from deepagents import create_deep_agent
from retriever_factory import RetrieverFactory

# Initialize Retrieval Factory with documents (toy corpus for POC)
DOCS = []  # Load or define your document texts here (list of strings or (text,
metadata) tuples)
retriever_factory = RetrieverFactory(DOCS)

# Create the DeepAgent with our custom retrieval tool
def retrieve_tool(query: str, strategy: str = "auto") -> str:
    """Retrieve relevant content using the specified strategy."""
    docs = retriever_factory.retrieve(query, strategy)
    # Join top docs' content (with separators) to return as one string
    snippets = []
    for doc in docs:
        text = doc.page_content if hasattr(doc, "page_content") else str(doc)
        snippets.append(text[:1000])  # limit size per snippet to avoid huge
outputs
    return "\n\n".join(snippets)

# Agent system prompt/instructions
instructions = (
    "You are an expert research assistant. "
    "Use the tools at your disposal to find information and answer questions
truthfully. "
    "Always cite the information you find in the workspace files. If you cannot
find an answer, say so."
)
tools = [retrieve_tool]  # our custom retrieval tool (DeepAgents will also add
built-ins like write_file, etc.)
agent = create_deep_agent(tools=tools, instructions=instructions)

# Set up MCP server
mcp = FastMCP("DeepAgentServer")

@mcp.tool()
def research(question: str, strategy: str = "auto") -> str:
    """Plan and research the question using the deep agent, returning a final
answer."""

# If a specific retrieval strategy is requested, we inform the agent via the
question prefix.
    query = question if strategy == "auto" else f"[Use {strategy} strategy]
{question}"
    result = agent.invoke({"messages": [{"role": "user", "content": query}]})
```

```
    # Assume result is a dict with 'output' key for final answer (since
deepagents returns a LangGraph output)
    answer = result.get("output") or result.get("answer") or ""
    return answer

@mcp.tool()
def plan(goal: str) -> str:
    """Generate a step-by-step plan (todo list) for the given goal without
executing it."""
    prompt = f"Plan ONLY: {goal}"
    result = agent.invoke({"messages": [{"role": "user", "content": prompt}]})
    # The agent's output likely includes a list of todos if the prompt steered
it correctly.
    plan_text = result.get("output", "")
    return plan_text

@mcp.tool()
def execute(step: str) -> str:
    """Execute a single step (task) using the agent and return the result."""
    # We treat the step as a user instruction to the agent.
    result = agent.invoke({"messages": [{"role": "user", "content": step}]})
    return result.get("output", "")

# Optionally, expose workspace resource to read files (if needed by external
observer)
@mcp.resource("workspace://files/{path}")
def read_file(path: str) -> str:
    """Read the content of a file from the agent's workspace."""
    state = agent.state or {}  # LangGraph State holds files perhaps in
agent.state['files']
    files = state.get("files", {})
    content = files.get(path, "")
    return content

if __name__ == "__main__":
    transport = os.environ.get("MCP_TRANSPORT", "stdio")
    if transport == "stdio":
        mcp.run(transport="stdio")
    else:
        # For HTTP transport, specify host/port
        mcp.run(transport="streamable-http", host="0.0.0.0",
port=int(os.environ.get("MCP_PORT", 8000)))
```

artifacts/retriever_factory.py:

```
from langchain.retrievers import EnsembleRetriever
from langchain_community.retrievers import BM25Retriever
from langchain.vectorstores import FAISS
from langchain.embeddings import OpenAIEmbeddings
# (If using a local model for embeddings, replace OpenAIEmbeddings accordingly)
# Optionally import an LLM for multi-query expansions or reranking:
from langchain_openai import ChatOpenAI

class RetrieverFactory:
    def __init__(self, documents: list[str]):
        # Initialize BM25 (sparse) retriever
        if documents:
            self.bm25 = BM25Retriever.from_texts(documents)
        else:
            self.bm25 = None
        # Initialize FAISS (dense vector) retriever
        if documents:
            embedding_model = OpenAIEmbeddings()  # uses OPENAI_API_KEY
            self.faiss_store = FAISS.from_texts(documents, embedding_model)
            self.vector = self.faiss_store.as_retriever(search_kwargs={"k": 5})
        else:
            self.faiss_store = None
            self.vector = None
        # Ensemble (hybrid) retriever combining BM25 + Vector
        if self.bm25 and self.vector:
            self.ensemble = EnsembleRetriever(retrievers=[self.bm25,
self.vector], weights=[0.5, 0.5])
        else:
            self.ensemble = None
        # Initialize LLM for query expansion and reranking if needed
        try:
            self.llm = ChatOpenAI(model_name="gpt-3.5-turbo", temperature=0)
        except Exception:
            self.llm = None  # handle case if API key not set or using local LLM

    def retrieve(self, query: str, strategy: str = "auto"):
        strategy = strategy.lower()
        if strategy in ("auto", "ensemble"):
            if self.ensemble:
                return self.ensemble.get_relevant_documents(query)
            # Fallback to vector if ensemble not available
            return self.vector.get_relevant_documents(query) if self.vector else
[]
        if strategy == "bm25":
            return self.bm25.get_relevant_documents(query) if self.bm25 else []
        if strategy == "vector":
            return self.vector.get_relevant_documents(query) if self.vector else
```

```
[]
        if strategy == "multi_query":
            return self._multi_query_search(query)
        if strategy == "rerank":
            return self._rerank_search(query)
        if strategy == "parent_doc":
            return self._parent_doc_search(query)
        # If unknown strategy, default to ensemble
        return self.ensemble.get_relevant_documents(query) if self.ensemble else
[]

    def _multi_query_search(self, query: str):
        # Generate multiple query variations using LLM
        sub_queries = [query]
        if self.llm:
            prompt = f"Brainstorm 3 different search queries for: '{query}'."
            try:
                resp = self.llm.predict(prompt)

# Split the LLM response into distinct queries (assuming newline separated)
                sub_queries = [q.strip("- ").strip() for q in resp.split("\n")
if q.strip()]
                sub_queries = [q for q in sub_queries if q] or [query]
            except Exception as e:
                sub_queries = [query]
        # Execute vector search for each sub-query and collect results
        seen_docs = {}
        results = []
        for q in sub_queries:
            docs = self.vector.get_relevant_documents(q) if self.vector else []
            for doc in docs:
                key = getattr(doc, "page_content", str(doc))
                if key not in seen_docs:
                    seen_docs[key] = doc
                    results.append(doc)
        # Simple fusion: just unique union (could sort by similarity, etc.)
        return results[:5]

    def _rerank_search(self, query: str):
        # Get initial docs (using vector as base retrieval)
        docs = self.vector.get_relevant_documents(query) if self.vector else []
        if not docs or not self.llm:
            return docs
        # Ask LLM to rank which document is most relevant
        # (We use a simple approach: have the LLM pick the best snippet)
        combined = "\n\n".join([f"Document {i+1}:\n{doc.page_content[:200]}" for
i, doc in enumerate(docs)])
        prompt = f"You are a verifier. Here is a question:\n{query}
```

```
\n\n{combined}\n\nWhich document(s) contain the answer? Respond with the
document numbers."
        try:
            answer = self.llm.predict(prompt)
            # Parse numbers from answer:
            chosen = [int(s) for s in answer.split() if s.isdigit()]
            if chosen:
                # Return chosen docs at front
                ranked = [docs[i-1] for i in chosen if 1 <= i <= len(docs)]
                # if some not chosen, append them after
                for doc in docs:
                    if doc not in ranked:
                        ranked.append(doc)
                return ranked[:5]
        except Exception:
            pass
        return docs[:5]

    def _parent_doc_search(self, query: str):
        # Perform a normal search first
        docs = self.vector.get_relevant_documents(query) if self.vector else []
        # If documents have metadata with an ID or filename, use that to fetch
full content
        full_docs = []
        seen_ids = set()
        for doc in docs:
            doc_id = None
            if doc.metadata and "source" in doc.metadata:
                doc_id = doc.metadata["source"]
            elif doc.metadata and "doc_id" in doc.metadata:
                doc_id = doc.metadata["doc_id"]
            # If no ID, we treat the doc as full already
            if not doc_id or doc_id in seen_ids:
                full_docs.append(doc)
            else:
                seen_ids.add(doc_id)
                # Here we assume we can retrieve full doc by id (in a real
scenario, store mapping of id->full text)
                full_text = doc.page_content  # placeholder: we only had chunk,
but no full storage in this POC
                doc.page_content = full_text
                full_docs.append(doc)
        return full_docs
```

artifacts/eval_harness.py:

```
import json
from langchain_evaluation import RagasEvaluatorChain  # hypothetical import,
assume integrated
from ragas.metrics import faithfulness, answer_relevancy, context_precision,
context_recall

# Load golden set
examples = []
with open("golden_set.jsonl", "r") as f:
    for line in f:
        if line.strip():
            examples.append(json.loads(line))

# Initialize evaluator chains for each metric
metrics = {
    "faithfulness": RagasEvaluatorChain(metric=faithfulness),
    "answer_relevancy": RagasEvaluatorChain(metric=answer_relevancy),
    "context_precision": RagasEvaluatorChain(metric=context_precision),
    "context_recall": RagasEvaluatorChain(metric=context_recall)
}

# Function to get agent's answer and sources (simulate single-turn QA using our
agent or pipeline)
from mcp_server import agent, retriever_factory  # assuming we can import agent
and retriever from server

def answer_question(query):
    """Run the agent on the query and return answer and source docs."""
    result = agent.invoke({"messages": [{"role": "user", "content": query}]})
    answer = result.get("output", "")
    # If agent wrote a final report to a file, we could read workspace file
here. For simplicity:
    source_docs = []

# If our retrieve_tool already returned content in answer, we might not have
separate source docs.
    # In a refined setup, agent could output references or we capture
intermediate retrievals.
    # Here we skip retrieving source docs explicitly.
    return answer, source_docs

# Evaluate each example
eval_results = []
for ex in examples:
    query = ex["query"]
    true_answers = ex.get("ground_truths", ex.get("ground_truth", []))
    if isinstance(true_answers, str):
```

```
        true_answers = [true_answers]
    # Get the agent's answer
    ans, docs = answer_question(query)
    result_record = {"query": query, "answer": ans, "ground_truths":
true_answers}
    if docs:
        # If we have Document objects, convert to text for evaluation
        context = " ".join([d.page_content for d in docs])
        result_record["context"] = context
    # Compute each metric
    for name, chain in metrics.items():
        try:
            score = chain(result_record)[f"{name}_score"]
        except Exception:
            score = chain(result_record).get(f"{name}_score", None)
        result_record[name] = score
    eval_results.append(result_record)
    print(f"Q: {query}\nA: {ans}\nScores: " +
          ", ".join(f"{k}={result_record[k]:.3f}" for k in metrics.keys()) +
"\n---")

# Save results to JSONL
with open("eval_results.jsonl", "w") as f:
    for rec in eval_results:
        f.write(json.dumps(rec) + "\n")
```

artifacts/golden_set.jsonl:

```
{"query": "How did New York City get its name?", "ground_truths":
["It was named after the Duke of York in 1664 when the English took control."]}
{"query": "Which borough of New York City has the highest population?",
"ground_truths": ["Brooklyn"]}
{"query": "When was the Eiffel Tower built?", "ground_truths": ["1887-1889"]}
{"query": "Name a fruit that is a hybrid of mandarin and pomelo.",
"ground_truths": ["The orange (sweet orange) is a hybrid of mandarin and
pomelo."]}
```


1 13 Deep Agents - Docs by LangChain
https://docs.langchain.com/labs/deep-agents/overview

2 5 7 20 40 41 47 49 Would you like an agent with that? — Introducing the Deep Agents SDK | by Tituslhy | MITB For All | Sep, 2025 | Medium
https://medium.com/mitb-for-all/would-you-like-an-agent-with-that-introducing-the-deep-agents-sdk-c4e0670440ea

3 4 6 8 9 11 14 15 GitHub - langchain-ai/deepagents
https://github.com/langchain-ai/deepagents

10 12 Quickstart - Docs by LangChain
https://docs.langchain.com/labs/deep-agents/quickstart

16 17 18 Anthropic's Model Context Protocol (MCP): A Universal Connector for AI | GPT-trainer Blog
https://gpt-trainer.com/blog/anthropic+model+context+protocol+mcp

19 21 Model Context Protocol (MCP) - Docs by LangChain
https://docs.langchain.com/oss/python/langchain/mcp

22 Model Context Protocol (MCP) for Retrieval-Augmented Generation (RAG) and Agentic AI | by Tamanna | Medium
https://medium.com/@tam.tamanna18/model-context-protocol-mcp-for-retrieval-augmented-generation-rag-and-agentic-ai-6f9b4616d36e

23 24 27 30 How to combine results from multiple retrievers | LangChain
https://python.langchain.com/docs/how_to/ensemble_retriever/

25 RAG Fusion with Reciprocal Rank Fusion | by Abheshith - Medium
https://medium.com/@abheshith7/rag-fusion-with-reciprocal-rank-fusion-73f67934dde7

26 31 33 34 35 36 42 43 Evaluating RAG pipelines with Ragas + LangSmith
https://blog.langchain.com/evaluating-rag-pipelines-with-ragas-langsmith/

28 29 Hybrid search scoring (RRF) - Azure AI Search - Microsoft Learn
https://learn.microsoft.com/en-us/azure/search/hybrid-search-ranking

32 How to evaluate a Langchain RAG system with RAGAs | ML_NEWS3
https://wandb.ai/byyoung3/ML_NEWS3/reports/How-to-evaluate-a-Langchain-RAG-system-with-RAGAs--Vmlldzo5NzU1NDYx

37 38 39 Langchain | Ragas
https://docs.ragas.io/en/v0.1.21/howtos/integrations/langchain.html

44 45 46 48 GitHub - langchain-ai/deep-agents-from-scratch
https://github.com/langchain-ai/deep-agents-from-scratch