# Technical Specification: Deep Agent Architecture for Automated Article Generation

## 1. Executive Summary

The evolution of autonomous agentic systems has reached a critical inflection point. Early paradigms, characterized by the "ReAct" (Reason + Act) loop, demonstrated the potential of Large Language Models (LLMs) to interact with external tools but ultimately failed to scale for complex, long-horizon tasks. These "shallow" agents suffer from catastrophic context drift, reasoning degradation, and a lack of teleological persistence—the ability to maintain a high-level goal over thousands of intermediate steps. This report presents a comprehensive technical specification for the "Deep Agent" architecture, a paradigm shift designed to remediate these failures through the implementation of a Virtual Filesystem (VFS), Explicit Planning mechanisms, and Recursive Sub-agents.

This architectural overhaul targets the legacy "Article Agent," transforming it from a fragile linear script into a robust, stateful cognitive engine capable of producing high-fidelity research reports. The proposed system leverages **LangGraph** for sophisticated state orchestration, **Postgres** for durable checkpointing, and the **Groq API**—utilizing **Llama 3.1 8B** for high-throughput reasoning and **Llama 3.3 70B** for nuanced synthesis—to achieve an optimal balance of cost, latency, and quality. Furthermore, the integration of **DuckDuckGo** and **Trafilatura** provides a cost-effective perception layer, governed by a rigorous **Token Budgeting Middleware** to ensure economic viability.

The report details the theoretical underpinnings, component specifications, and operational strategies required to deploy this system. It argues that by decoupling "Context" from "History" via a VFS, and by enforcing strict hierarchical planning, we can achieve a level of agentic reliability that is currently absent in standard implementations.

## 2. Theoretical Framework: The Shift from Shallow to Deep Cognition

### 2.1 The Pathology of the ReAct Loop

To understand the necessity of the Deep Agent architecture, one must first diagnose the limitations of the incumbent ReAct framework. Introduced as a method to interleave reasoning traces with action execution, ReAct relies on a single, rolling context window to maintain state.1 The agent observes the world, generates a "Thought," executes an "Action," and observes the "Result."

For atomic tasks—"What is the weather in Tokyo?"—this is sufficient. However, for a task such as "Write a comprehensive report on the future of solid-state batteries," the ReAct loop collapses under its own weight. As the agent gathers information, the context window fills with search results, HTML scraps, and intermediate reasoning steps. This accumulation introduces two fatal errors:

1. **Signal Dilution:** As the ratio of noise (raw data) to signal (instructions/goals) increases, the LLM's attention mechanism struggles to attend to the original directive. The agent effectively "forgets" it is writing a report and becomes obsessed with the mechanics of the last search query.2
2. **Context Window Overflow:** Even with 128k context windows, the unstructured accumulation of history is inefficient. It forces the model to re-process thousands of irrelevant tokens at every step, increasing latency and cost while degrading reasoning performance due to the "Lost in the Middle" phenomenon.

### 2.2 The Deep Agent Paradigm

The Deep Agent architecture, exemplified by systems like "Claude Code" or "Deep Research," addresses these pathologies by mimicking human cognitive workflows. Humans do not hold an entire research project in working memory. We utilize **External Memory** (notes, files) and **Executive Function** (planning, task decomposition) to manage complexity.2

This implementation relies on four foundational pillars:

1. **Virtual Filesystem (VFS):** A structured, persistent storage substrate that serves as the agent's long-term working memory. Unlike a vector database, which relies on fuzzy semantic retrieval, a VFS allows for deterministic, agent-directed "Context Pulling".4
2. **Explicit Planning:** The reification of the agent's "intent" into a mutable data structure (e.g., a JSON plan). This plan exists independently of the chat history, acting as an immutable anchor for the agent's teleology.3
3. **Recursive Sub-agents:** The adherence to the Single Responsibility Principle (SRP) by delegating distinct cognitive loads—Research vs. Writing—to specialized sub-graphs. This encapsulates noise, preventing the detailed logs of a web scrape from polluting the high-level drafting context.5
4. **Context Pulling:** The active, intentional selection of information. Instead of a system automatically retrieving "relevant" chunks (RAG), the agent explicitly decides to "read" a file, granting it agency over its own cognitive load.4

## 3. Orchestration Architecture with LangGraph

The structural backbone of the Deep Agent is **LangGraph**. Unlike purely DAG-based (Directed Acyclic Graph) orchestrators, LangGraph supports cyclic computational flows, essential for the iterative nature of research and revision.

### 3.1 The Global State Topology

At the highest level, the application is modeled as a state machine where the state is not just a list of messages, but a typed schema containing the VFS, the Plan, and the Budget. This adheres to the **Interface Segregation Principle**, ensuring that different nodes interact only with the state elements they require.

The proposed OverallState schema is defined as follows:

| **Field** | **Type** | **Description** |
| --- | --- | --- |
| messages | list | The high-level conversation history (User <-> Orchestrator). |
| vfs | FileSystemState | The serialized state of the Virtual Filesystem. |
| plan | PlanState | The current structured plan (JSON). |
| token\_budget | BudgetState | Current consumption vs. limit. |
| user\_preferences | dict | extracted style/tone constraints. |

### 3.2 Hierarchical Subgraphs and State Isolation

A critical requirement is the use of recursive sub-agents. In LangGraph, this is implemented by treating a compiled graph as a node within a parent graph. This design pattern offers superior **State Isolation**.7

When the Orchestrator delegates a task to the Researcher subgraph:

1. **Input Transformation:** The parent state is filtered. The Researcher receives the vfs and the specific task\_description, but *not* the full history of the Planner's deliberations.
2. **Isolated Execution:** The Researcher loops through search-scrape-summarize cycles. It generates dozens of intermediate ToolMessage and AIMessage objects.
3. **Output Transformation:** Upon completion, the Researcher returns *only* the updated vfs (containing new files) and a brief status message. The noisy intermediate steps are discarded, never entering the Orchestrator's context.

This pattern drastically reduces token consumption and prevents the "hallucination via dilution" described earlier. It effectively implements a "Garbage Collection" mechanism for cognitive history.

### 3.3 Persistence and Durability via Postgres

To satisfy the requirement for durability, we integrate **Postgres** as the backing store for LangGraph's checkpointing mechanism. This transforms the agent from a transient script into a durable service.8

The PostgresSaver effectively snapshots the OverallState after every node execution. This allows for:

* **Time-Travel Debugging:** Developers can inspect the state of the VFS at step 50 to diagnose why the agent went off-track.
* **Resumability:** If the Groq API times out or the process is pre-empted, the agent can be restarted. It will load the latest checkpoint from Postgres and resume the specific sub-task (e.g., "Scraping URL #3") without restarting the entire workflow.

## 4. The Virtual Filesystem (VFS): Design and Implementation

The VFS is the defining feature of this Deep Agent. It is not merely a data store; it is the cognitive workspace.

### 4.1 Schema Definition

We reject the complexity of a real OS file system (permissions, symlinks) in favor of a **KISS** (Keep It Simple, Stupid) approach suitable for LLM manipulation. The VFS is modeled as a flat dictionary of file objects, simulating a hierarchy via path strings.4

**VFS State Model (Pydantic):**

Python

class File(BaseModel):  
 name: str  
 content: str  
 created\_at: str  
 updated\_at: str  
 metadata: dict[str, str] # Stores source URL, author, tags  
  
class FileSystemState(BaseModel):  
 files: dict[str, File]  
 working\_directory: str = "/"

### 4.2 The "Context Pulling" Toolset

The agent interacts with the VFS exclusively through a set of defined tools. This abstraction allows us to swap the backend (e.g., to S3 or Redis) without altering the agent's prompts, adhering to the **Dependency Inversion Principle**.

#### 4.2.1 ls (List Files)

This tool allows the agent to survey its knowledge base.

* **Function:** Returns a list of filenames, sizes, and timestamps.
* **Cognitive Function:** "Metacognition." Before answering a question, the agent checks *if* it has the information.
* **Optimization:** For large file systems, ls can accept a path argument to list specific subdirectories, preventing context flooding.

#### 4.2.2 read\_file (Context Pulling)

This is the mechanism of active retrieval.

* **Function:** Returns the content of a specific file.
* **Cognitive Function:** "Focus." The agent explicitly selects the data relevant to the current paragraph it is writing.
* **Heuristic:** To prevent reading massive files, this tool can optionally accept start\_line and end\_line parameters, or the agent can be instructed to read a "summary" file first.

#### 4.2.3 write\_file (Memory Consolidation)

This tool enables the persistence of thought.

* **Function:** Creates or overwrites a file.
* **Cognitive Function:** "Learning." When the *Researcher* extracts data from a website, it does not just pass the text to the *Writer*. It synthesizes the data and saves it to research/topic\_analysis.md. This file becomes a permanent artifact of the system.

### 4.3 VFS vs. Vector Stores

It is crucial to distinguish the VFS from Vector Stores (RAG). Vector stores rely on embedding similarity, which is probabilistic and opaque. If the agent needs "The specific rate limit for the Groq API," a vector store might retrieve generic API documentation. A VFS allows the agent to navigate to research/groq/rate\_limits.txt, guaranteeing the retrieval of the exact, curated context. The VFS is superior for *task-specific* memory where the structure of information (the file hierarchy) carries semantic meaning.

## 5. Explicit Planning and Reasoning (Groq/Llama 3.1)

The "Brain" of the operation is split into two distinct faculties: The **Planner** (Logic/Routing) and the **Writer** (Creativity/Synthesis). This split optimizes for the strengths of the different Llama 3 models available via Groq.

### 5.1 The Planner: Llama 3.1 8B

The Planner is responsible for decomposing the user's high-level request into a Directed Acyclic Graph (DAG) of tasks. We utilize **Llama 3.1 8B** for this role.

* **Why 8B?** Planning is a structural task, not a creative one. The 8B model is sufficient for logic ("If A is done, do B") and is significantly faster and cheaper on Groq.9
* **JSON Mode:** To ensure the plan is machine-readable, we enforce **JSON Mode** in the Groq API (response\_format={"type": "json\_object"}). This prevents the model from outputting conversational filler ("Here is your plan...") and forces a strict schema.10

The Planning Schema:

The output of the Planner is strictly validated against a schema:

JSON

{  
 "goal": "Update implementation plan...",  
 "phases":  
 },  
 {  
 "id": 2,  
 "name": "Drafting Phase",  
 "tasks": [...]  
 }  
 ]  
}

### 5.2 The Writer: Llama 3.3 70B

The Writer is responsible for prose generation. We utilize **Llama 3.3 70B** for this role.

* **Why 70B?** The 70B model demonstrates significantly higher capability in nuance, coherence, and long-form synthesis. It is less prone to repetition and maintains a consistent tone better than the 8B model.12
* **Reasoning Format:** For complex synthesis, we may utilize the reasoning\_format="raw" parameter (if available/supported by the specific endpoint configuration) to allow the model to "think" before writing, improving the logical flow of the arguments.11

### 5.3 Tokenizer Nuances and Budgeting

A critical implementation detail often overlooked is the **Tokenizer**. Llama 3 uses a different tokenizer than GPT-4 (cl100k\_base). While tiktoken is the standard library for counting tokens, it does not natively support the Llama 3 vocabulary (approx. 128k tokens vs. OpenAI's 100k).13

* **Implication:** Using cl100k\_base as a proxy for Llama 3 token counting will result in an error margin (usually undercounting, as Llama 3's larger vocab is more efficient).
* **Mitigation:** For the **Token Budgeting Middleware**, we will use tiktoken with the cl100k\_base encoding but apply a safety multiplier (e.g., 1.1x) to estimated counts. This provides a "safe" upper bound for budgeting without requiring the heavy transformers library to load the exact Llama 3 tokenizer in the lightweight planner environment.15

## 6. Perception Layer: Search and Scraping

The "Deep Agent" must interact with the external world to ground its writing in reality. This is handled by the **Research Subgraph**.

### 6.1 Search Strategy: DuckDuckGo

We utilize the duckduckgo\_search library. While free, this API is aggressively rate-limited. Relying on it without a mitigation strategy will lead to RatelimitException failures that crash the agent.16

**Resilience Architecture:**

1. **Backends:** The library supports multiple backends (api, html, lite). The agent is configured to rotate through these. If api fails (429), it automatically fails over to html scraping of the search results page.16
2. **Exponential Backoff:** We wrap the search tool in a decorator that implements exponential backoff.
   * Attempt 1: Fail.
   * Wait: 2 seconds.
   * Attempt 2: Fail.
   * Wait: 4 seconds.
   * Attempt 3: Fail.
   * Action: Return a specific "Search Unavailable" error code to the Planner, prompting it to skip the task or try a general knowledge fallback, rather than crashing.18

### 6.2 Extraction Strategy: Trafilatura

Raw HTML is toxic to LLM context windows. It is token-heavy and semantic-poor. We use **Trafilatura** for extraction, which outperforms BeautifulSoup in identifying main content text and stripping boilerplate (navbars, ads).19

**The Extraction Pipeline:**

1. **Fetch:** trafilatura.fetch\_url(url) handles the HTTP request, including User-Agent rotation to avoid bot detection.
2. **Extract:** trafilatura.extract(downloaded, include\_comments=False, include\_tables=True). We explicitly disable comments to reduce noise but enable tables as they often contain high-density data relevant to technical reports.
3. **Metadata Injection:** The extraction is not just text. We append the source\_url and date to the file header. This is critical for the Writer to generate accurate citations later in the process.
4. **Fast Mode:** For high-volume research, we enable no\_fallback=True in Trafilatura to speed up processing, accepting a slight trade-off in recall for significant gains in speed.19

## 7. Middleware and Governance

In a production environment, an autonomous agent acts as an unconstrained cost center. Without governance, a loop could run indefinitely, burning through credits.

### 7.1 Token Budgeting Middleware

We implement a "Circuit Breaker" pattern via LangGraph middleware. This logic runs *before* every node execution.

**The Algorithm:**

1. **Budget Allocation:** The user sets a max\_token\_budget (e.g., 1,000,000 tokens) in the initial state.
2. **Tracking:** Every LLM call (Groq) returns usage statistics (prompt\_tokens, completion\_tokens). These are aggregated into a tokens\_consumed state variable.
3. **Enforcement:**
   * **Warning Threshold (80%):** The middleware injects a system\_message into the context: "Warning: 80% of token budget consumed. Wrap up research immediately and move to drafting."
   * **Hard Stop (100%):** The middleware intercepts the flow and forces a transition to the Finalize node, preventing any further research or drafting loops.

### 7.2 Safety and Compliance

Beyond cost, we must ensure the agent does not output harmful content. While Llama 3 has built-in safety guardrails, the middleware adds a layer of **Refusal Detection**. If the model output contains standard refusal strings ("I cannot fulfill this request"), the middleware flags the task as failed and prompts the Planner to rephrase the query, preventing the agent from saving a refusal message as "research data".12

## 8. Detailed Sub-Agent Specifications

### 8.1 The Planner Agent

* **Model:** Llama 3.1 8B (Groq).
* **Responsibility:** Task Decomposition & Route Management.
* **Inputs:** User Query, Current VFS State (File list only), Tool Outputs (Status codes).
* **Outputs:** JSON Plan.
* **Behavior:** The Planner acts as the project manager. It does not read the files; it only checks if they exist. If the plan requires "Research VFS," and the Planner sees research/vfs.md exists and is > 1KB, it marks the task complete.

### 8.2 The Researcher Agent (Subgraph)

* **Model:** Llama 3.1 8B (Groq).
* **Responsibility:** Data Acquisition & Summarization.
* **Inputs:** Task Description ("Find rate limits for Groq").
* **Tools:** duckduckgo\_search, trafilatura, write\_file.
* **Behavior:**
  1. **Search:** Generates 3-5 distinct search queries.
  2. **Filter:** Selects the top 3 most promising URLs based on snippet analysis.
  3. **Scrape:** Extracts content.
  4. **Synthesize:** *Crucial Step.* The Researcher does not save the raw scrape. It passes the scrape through Llama 3.1 8B with the prompt "Extract key facts about Groq rate limits from this text."
  5. **Save:** Writes the extracted facts to research/groq\_limits.txt.

### 8.3 The Writer Agent (Subgraph)

* **Model:** Llama 3.3 70B (Groq).
* **Responsibility:** Synthesis & Prose Generation.
* **Inputs:** Writing Goal, VFS Content.
* **Tools:** read\_file, ls.
* **Behavior:**
  1. **Context Pulling:** Runs ls research/ to see available notes. Runs read\_file on relevant notes.
  2. **Drafting:** Generates a section of the report.
  3. **Self-Correction:** If the drafted section lacks detail (e.g., "The rate limit is [unknown]"), it flags a missing dependency, potentially triggering a callback to the Planner (though in a strict DAG, it might just note the limitation).
  4. **Finalization:** Writes the section to drafts/section\_X.md.

## 9. Quality Assurance: A Robust Testing Strategy

Testing non-deterministic agents is notoriously difficult. We employ a **Mock-First** strategy to validate the logic (State Machine) independently of the intelligence (LLM).21

### 9.1 Mocking the VFS

We do not use the real OS file system for tests. We implement a MockFileSystem that implements the read/write/ls interface but stores data in a Python dictionary.

* **Test Case:** "Writer should fail gracefully if file missing."
* **Setup:** Initialize MockFileSystem with empty dict.
* **Action:** Invoke Writer.
* **Assertion:** Verify Writer catches FileNotFoundError and returns appropriate status, rather than crashing.

### 9.2 Mocking Search (Dependency Injection)

We abstract the search provider behind an interface ISearchProvider.

* **Production:** DuckDuckGoSearchProvider.
* **Test:** StaticSearchProvider.
* **Scenario:** We inject a StaticSearchProvider that always returns a specific set of URLs. This allows us to test the *Trafilatura* integration deterministically—we know exactly what HTML will be fetched (served from a local mock server) and can assert that the agent extracts the correct text.23

### 9.3 Mocking the LLM

We use LangChain's GenericFakeChatModel for unit testing the graph topology.

* **Scenario:** Testing the Planner's routing logic.
* **Setup:** Configure GenericFakeChatModel to return a specific JSON plan string {"tasks": [...]}.
* **Action:** Run the Planner Node.
* **Assertion:** Verify that the graph transitions to the Researcher node. This confirms the routing logic works, without paying for Groq tokens or waiting for Llama 3 inference.21

## 10. Risk Analysis and Mitigation

| **Risk** | **Probability** | **Impact** | **Mitigation Strategy** |
| --- | --- | --- | --- |
| **API Rate Limiting** | High | Critical | Exponential backoff on DuckDuckGo; Rotation of user-agents; Fallback to html backend. |
| **Context Window Overflow** | Medium | High | Strict VFS "Context Pulling" (only read what is needed); Summarization step in Researcher. |
| **Infinite Loops** | Low | High | Token Budgeting Middleware; Max-step limits on LangGraph recursion. |
| **Hallucination** | Medium | Moderate | Writer explicitly cited sources from VFS; "Grounding" prompt instructions. |
| **Data Loss** | Low | Critical | Postgres checkpointing after every step. |

## 11. Conclusion

The proposed "Deep Agent" architecture represents a significant maturity in automated content generation. By moving beyond the transient, chaotic memory of the ReAct loop and embracing a structured, persistent **Virtual Filesystem**, we create an agent that can "think" over long horizons. The integration of **Explicit Planning** ensures teleological consistency, while **Recursive Sub-agents** optimize the economic trade-off between the fast, cheap Llama 3.1 8B and the powerful, expensive Llama 3.3 70B.

This system is not merely a script; it is a cognitive architecture. It possesses memory, intent, and the ability to self-correct. With the robustness provided by **LangGraph** orchestration and **Postgres** durability, it is ready for deployment as a production-grade tool for automated, high-depth research and reporting.

# Deep Agent Implementation Guide: Detailed Operational Procedures

## 12. Implementation Procedures: Step-by-Step

This section translates the architectural specifications into concrete implementation steps. It assumes a standard Python environment and access to the required API keys (Groq, Postgres).

### 12.1 Environment Setup and Dependency Management

The project relies on a specific set of libraries to handle the orchestration, inference, and persistence layers.

**Core Dependencies:**

* langgraph: The orchestration engine.
* langchain-groq: Connector for Llama 3 models.
* duckduckgo-search: The search tool provider.
* trafilatura: The scraping engine.
* psycopg: Postgres driver for persistence.8
* pydantic: For state validation and schema definition.
* tiktoken: For token estimation (budgeting).

Configuration Management:

We use pydantic-settings to manage configuration, ensuring that API keys and budget limits are validated at startup.

Python

from pydantic\_settings import BaseSettings  
  
class Settings(BaseSettings):  
 GROQ\_API\_KEY: str  
 POSTGRES\_URI: str  
 MAX\_TOKEN\_BUDGET: int = 100000  
 USER\_AGENT: str = "DeepAgent/1.0"  
  
 class Config:  
 env\_file = ".env"

### 12.2 The Virtual Filesystem (VFS) Module

This module is the heart of the system. It must be robust, as all agents depend on it.

#### 12.2.1 File System Logic

We implement the FileSystem class to encapsulate the dictionary operations. This provides a clean API for the tools.

* **Path Normalization:** The VFS must handle paths like /research/../drafts correctly. We use Python's pathlib (mocked or adapted) to normalize strings before accessing the dictionary key.
* **Metadata Handling:** Every write operation automatically updates the updated\_at timestamp. This is crucial for the "Janitor" process (if implemented later) to clean up old files.

#### 12.2.2 VFS Tools Implementation

The tools exposed to the LLM are simple wrappers around the FileSystem methods.

**Crucial Implementation Detail:** The read\_file tool includes a safety check. If the requested file is larger than 10,000 characters, it truncates the output and appends a warning: *"...[File truncated. Use 'read\_file\_chunk' to read more]..."*. This prevents the agent from accidentally blowing its own context window by reading a massive scrape.5

### 12.3 The Planner Agent Implementation

The Planner is the entry point of the graph.

#### 12.3.1 Prompt Engineering for JSON

Llama 3.1 8B is capable of JSON, but requires specific prompting to be reliable.

System Prompt:

"You are the Project Manager. Your goal is to plan the execution of a research report. You have access to a Virtual Filesystem. Output your plan as a valid JSON object matching the Plan schema. Do not include any explanation or chatter. Only JSON."

#### 12.3.2 The Replanning Loop

The Planner is not 'fire and forget'. It is a node that is revisited.

* **Input:** The current Plan object from the state.
* **Logic:**
  1. Iterate through tasks.
  2. If a task is in\_progress but the result file exists in VFS, mark it complete.
  3. If a task is failed, generate a sub-plan to fix it or mark it skipped.
  4. Select the next pending task.
* **Output:** An updated Plan object.

### 12.4 The Research Subgraph Implementation

The Research Subgraph is where the "messy" work happens.

#### 12.4.1 State Schema for Subgraph

The Research subgraph has its own state, separate from the global state.

Python

class ResearchState(TypedDict):  
 query: str  
 search\_results: List[dict]  
 scraped\_content: str  
 summary: str  
 vfs\_updates: Dict[str, File] # Files to merge back to global

#### 12.4.2 The "Search-Decide-Scrape" Loop

1. **Node: Search:** Executes duckduckgo\_search. Stores results in search\_results.
2. **Node: Select:** The LLM (8B) reviews the snippets in search\_results. It decides which URL is most relevant.
3. **Node: Scrape:** Executes trafilatura on the chosen URL.
4. **Node: Assess:** The LLM reads the scrape. Is it relevant?
   * *Yes:* Proceed to Summarize.
   * *No/Empty:* Loop back to **Select** and pick the next URL.
5. **Node: Summarize:** Compresses the text.
6. **Node: Save:** adds the file to vfs\_updates.

This inner loop ensures that the Researcher is autonomous. It doesn't bother the Planner with "I clicked a dead link." It just tries the next one.

### 12.5 The Writer Agent Implementation

The Writer is the "consumer" of the VFS.

#### 12.5.1 The Drafting Pipeline

1. **Requirement Analysis:** The Writer reads the task description from the Plan.
2. **Context Assembly:** The Writer executes ls /research. It sees market\_data.txt and competitor\_analysis.txt.
3. **Context Pulling:** It executes read\_file on these documents.
4. **Synthesis:** It generates the draft text.
5. **Critique (Optional):** For high-quality outputs, we can insert a "Critique" node where the 70B model reviews its own draft against the requirement. "Does this draft actually answer the prompt?" If not, it revises.

### 12.6 Testing and Validation Framework

We utilize pytest for the testing harness.

#### 12.6.1 Unit Tests (Logic)

We test the Python logic without any LLMs.

* **Test:** VFS.write -> VFS.read.
* **Test:** Plan.update\_status.
* **Test:** TokenBudget.check\_limit.

#### 12.6.2 Integration Tests (Mocked)

We test the flow of the graph using mocked LLMs.

* **Mocking Groq:** We patch the ChatGroq.invoke method.
  + For the Planner, we return a fixed JSON string.
  + For the Researcher, we return a fixed "I found this URL" string.
* **Assertion:** We assert that the graph moves from Planner -> Researcher -> Writer -> End. We do not care about the *content* of the writing, only that the *process* completed successfully.

#### 12.6.3 Durability Tests

We test the Postgres integration.

* **Scenario:** Run the agent for 5 steps. Kill the process (simulated error). Restart the agent with the same thread\_id.
* **Assertion:** Verify that the VFS state contains the files created in the first 5 steps. Verify that the Plan has the correct tasks marked as complete.

## 13. Operational Economics and Scaling

### 13.1 Cost Analysis

* **Planning (Llama 3.1 8B):** Very cheap. ~500 input tokens / ~200 output tokens per step.
* **Research (Llama 3.1 8B):** Moderate. Processing large HTML scrapes is token-intensive. However, since we filter/summarize *inside* the subgraph, we assume only the summary (small) is passed to the Writer.
* **Writing (Llama 3.3 70B):** Expensive. This is the main cost driver. By using the VFS to limit the context window (reading only relevant files), we significantly reduce the input token cost compared to a "dump everything" approach.

### 13.2 Latency Considerations

The Deep Agent is **slow**. It is not a chatbot. A full report might take 5-10 minutes to generate.

* **Sequential vs. Parallel:** The current plan is sequential. We can optimize by allowing the Planner to spawn multiple Researcher sub-agents in parallel (e.g., one for "Market Size," one for "Competitors"). LangGraph supports this via the Send API (Map-Reduce pattern).
* **Groq Throughput:** Groq's high throughput is the saving grace. Llama 3 on Groq runs at hundreds of tokens per second, making the "inference time" negligible compared to the "search/network time."

## 14. Second-Order Insights: The Cognitive Supply Chain

This architecture reveals a deeper truth about AI systems: we are moving from "Chat" to "Supply Chains."

* **Raw Material:** The internet (via Search).
* **Refining:** The Researcher (turning HTML into Clean Text).
* **Manufacturing:** The Writer (turning Text into Narrative).
* **Logistics:** The VFS (moving data between stations).
* **Management:** The Planner (optimizing the flow).

Just as in industrial supply chains, **inventory management** (VFS organization) is key. A cluttered VFS leads to a confused Writer. We may need to implement "Janitorial Agents" whose sole job is to organize, tag, and archive files in the VFS to keep the workspace clean.

Furthermore, the **Recursive** nature of the system implies that it is **Fractal**. The "Researcher" could, in theory, be another full Deep Agent with its own Planner and Sub-agents. This allows for infinite scaling of complexity, bounded only by the Token Budget.

## 15. Conclusion

The "Deep Agent" architecture detailed herein is a necessary evolution for autonomous knowledge work. By rigorously implementing the Virtual Filesystem, Explicit Planning, and Recursive Sub-agents, we overcome the limitations of the ReAct loop. The system is robust, audit-able (via the Plan and VFS), and economically viable (via Groq and Token Budgeting). This specification provides the complete blueprint for building a production-ready Article Agent capable of deep, sustained research and high-quality synthesis.

#### Works cited

1. Building Deep Agents with LangGraph: A Practical Guide - Medium, accessed December 6, 2025, <https://medium.com/@techofhp/building-deep-agents-with-langgraph-a-practical-guide-686422c1324e>
2. Deep Agents - LangChain Blog, accessed December 6, 2025, <https://blog.langchain.com/deep-agents/>
3. Building a Deep Research Agent with LangGraph And Exa - Sid Bharath, accessed December 6, 2025, <https://www.siddharthbharath.com/build-deep-research-agent-langgraph/>
4. Backends - Docs by LangChain, accessed December 6, 2025, <https://docs.langchain.com/oss/python/deepagents/backends>
5. Deep Agents overview - Docs by LangChain, accessed December 6, 2025, <https://docs.langchain.com/oss/python/deepagents/overview>
6. Conversational Patterns in LangGraph using Subgraphs | by Vinodh S Iyer | Medium, accessed December 6, 2025, <https://medium.com/@vin4tech/conversational-patterns-in-langgraph-using-subgraphs-366d4dd27ebc>
7. Subgraphs - Docs by LangChain, accessed December 6, 2025, <https://docs.langchain.com/oss/python/langgraph/use-subgraphs>
8. Memory - Docs by LangChain, accessed December 6, 2025, <https://langchain-ai.github.io/langgraph/how-tos/persistence_postgres/>
9. Examples - GroqDocs - Groq Console, accessed December 6, 2025, <https://console.groq.com/docs/examples>
10. Structured Outputs - GroqDocs - Groq Console, accessed December 6, 2025, <https://console.groq.com/docs/structured-outputs>
11. Reasoning - GroqDocs - Groq Console, accessed December 6, 2025, <https://console.groq.com/docs/reasoning>
12. Text Generation - GroqDocs - Groq Console, accessed December 6, 2025, <https://console.groq.com/docs/text-chat>
13. tokenizer.py - meta-llama/llama3 - GitHub, accessed December 6, 2025, <https://github.com/meta-llama/llama3/blob/main/llama/tokenizer.py>
14. Why does LLaMA-3 use LF token = 128 'Ä'? : r/LocalLLaMA - Reddit, accessed December 6, 2025, <https://www.reddit.com/r/LocalLLaMA/comments/1cpv7np/why_does_llama3_use_lf_token_128_%C3%A4/>
15. How to count tokens with Tiktoken - OpenAI Cookbook, accessed December 6, 2025, <https://cookbook.openai.com/examples/how_to_count_tokens_with_tiktoken>
16. duckduckgo-search 5.2.2 - PyPI, accessed December 6, 2025, <https://pypi.org/project/duckduckgo-search/5.2.2/>
17. duckduckgo\_search.exceptions.RatelimitException: 202 Ratelimit #6624 - GitHub, accessed December 6, 2025, <https://github.com/open-webui/open-webui/discussions/6624>
18. Duckduckgo search not working - Part 1 2022 - fast.ai Course Forums, accessed December 6, 2025, <https://forums.fast.ai/t/duckduckgo-search-not-working/105738>
19. Quickstart — Trafilatura 2.0.0 documentation - Read the Docs, accessed December 6, 2025, <https://trafilatura.readthedocs.io/en/latest/quickstart.html>
20. With Python — Trafilatura 2.0.0 documentation - Read the Docs, accessed December 6, 2025, <https://trafilatura.readthedocs.io/en/latest/usage-python.html>
21. Test - Docs by LangChain, accessed December 6, 2025, <https://docs.langchain.com/oss/python/langchain/test>
22. Test - Docs by LangChain, accessed December 6, 2025, <https://docs.langchain.com/oss/python/langgraph/test>
23. Mocking External APIs in Agent Tests - Scenario - LangWatch, accessed December 6, 2025, <https://scenario.langwatch.ai/testing-guides/mocks/>