Part 1: Automated Curriculum Vitae Verification using LLMs and MCP Tool Integration

1. System Architecture and Design Decisions

The primary objective of this project module is to construct an AI agent system capable of automatically parsing, verifying, and evaluating the authenticity of candidate resumes (CVs). The system architecture is designed around two core components: unstructured data parsing and external tool orchestration.

1.1 Core Reasoning Engine

The system utilizes the deepseek-chat model as its central reasoning engine, interfaced uniformly via LangChain's ChatOpenAI wrapper. To ensure consistency and reproducibility during the CV scoring phase, the model's temperature parameter was strictly set to 0. This design decision is critical for minimizing hallucination and ensuring deterministic outputs when the model is tasked with generating numerical confidence scores.

1.2 Document Processing Layer

During the CV parsing phase, the system eschews traditional and often noisy Optical Character Recognition (OCR) techniques in favor of the MarkItDown library. This tool directly converts PDF-formatted CVs into structured Markdown text. This decision significantly preserves the structural integrity of the original documents—such as tables, bulleted lists, and hierarchical headings—providing the Large Language Model (LLM) with a clean, context-rich input format.

1.3 Model Context Protocol (MCP) Integration

To cross-reference the claims made within the CVs, the system connects to a remote Model Context Protocol (MCP) server via MultiServerMCPClient. Through standardized interfaces, the agent is granted access to six core social graph tools:

- **Facebook API Mock Tools**: search_facebook_users, get_facebook_profile, get_facebook_mutual_friends.
- **LinkedIn API Mock Tools**: search_linkedin_people, get_linkedin_profile, get_linkedin_interactions.

These tools empower the agent to transcend static text limitations, enabling it to utilize fuzzy and exact matching to verify candidates' educational backgrounds, employment histories, and professional networks within dynamic social graphs.

2. Agent Workflow and Tool Usage Strategy

The verification workflow is designed as a multi-step interactive process.

2.1 Tool-Enabled Discovery Strategy

In the ideal agent workflow, when presented with a parsed CV, the system employs the following strategy:

1. **Entity Extraction**: Extract the candidate's name, current employer, and core skills from the Markdown text.
2. **Cross-Platform Retrieval**: Utilize search_linkedin_people to retrieve professional profiles, while simultaneously using search_facebook_users to trace personal footprints.
3. **In-Depth Verification**: Invoke get_linkedin_profile to detect discrepancies in employment dates, and call get_facebook_mutual_friends to verify claimed social or professional endorsements.

2.2 Direct Evaluation Strategy

For the scaled evaluation phase of this experiment, a direct baseline evaluation strategy was implemented. The fully parsed Markdown CVs were passed directly to the base LLM with a strict prompt:

"Please evaluate the following CV for its general validity and completeness. Provide a confidence score between 0.0 and 1.0. Respond only with a JSON object..."

This requires the model to assess the CV relying solely on its internal prior knowledge, logical reasoning capabilities, and the internal consistency of the text itself, without the immediate aid of external networking tools.

3. Sample Verification Results and Analysis

In the final testing phase, the system automatically scored and performed binary classification on a batch of 5 sample CVs (CV_1.pdf through CV_5.pdf).

3.1 Evaluation Metric

The decision threshold was configured at 0.5:

- If Score > 0.5 and Ground Truth = 1, the decision is marked correct.
- If Score ≤ 0.5 and Ground Truth = 0, the decision is marked correct.
- Otherwise, the decision yields no credit.

3.2 Quantitative Results

The system generated the following evaluation metrics:

- **Generated Confidence Scores**: [0.4, 0.65, 0.3, 0.2, 0.2]
- **Ground Truth Labels**: [1, 1, 1, 0, 0]
- **Binary Decisions (Threshold 0.5)**: [0, 1, 0, 0, 0]

Final Performance Score: 0.6 (representing a 60% accuracy rate, correctly classifying 3 out of 5 CVs: CV_2, CV_4, and CV_5).

3.3 Error Analysis & Future Improvements

Under the current Zero-shot evaluation paradigm, the system successfully identified all fabricated CVs (CV_4 and CV_5 both received scores of 0.2), demonstrating excellent **specificity**. However, the system produced **false negatives** for CV_1 and CV_3, assigning them low scores of 0.4 and 0.3, respectively, despite them being authentic.

This outcome highlights the inherent limitations of static text evaluation. When limited to text reading alone, the LLM exhibits a "skeptical" bias toward authentic CVs that may present information too concisely.

Future Work: To resolve this, future iterations should employ Retrieval-Augmented Generation (RAG) or Tool Injection during the final scoring prompt. By actively executing the MCP social tools (e.g., retrieving LinkedIn employment histories) and appending this structured ground-truth data to the prompt alongside the CV text, the model's confidence scores for valid CVs like CV_1 and CV_3 would likely cross the 0.5 threshold, significantly boosting the overall system accuracy.

Part 2: Autonomous Social Media Engagement using ReAct Agent Architecture on the Moltbook Platform

1. System Architecture and API Integration

The second phase of the project transitions from passive document evaluation to active, autonomous engagement within a simulated social network known as "Moltbook." The objective is to design a LangChain-based AI agent capable of discovering communities, parsing feeds, and executing state-changing actions (posting, commenting, subscribing) without human intervention, while strictly adhering to anti-spam guardrails.

1.1 Agent Identity and Security Subsystem

Before the agent can interact with the Moltbook ecosystem, it requires a secure, authenticated identity. To protect user privacy during registration, the system implements an affine cipher to reversibly encode the student ID (e.g., transforming `1155246851` into a padded numerical string like `68819498`).

This encoded ID is then used to register the agent via a standard REST API POST request (`/api/v1/agents/register`), which provisions a unique `MOLTBOOK_API_KEY`. This key is securely stored in the environment secrets and injected into the HTTP headers for all subsequent tool calls.

1.2 Custom Tool Abstraction Layer

Unlike Part 1, which utilized pre-built MCP servers, Part 2 requires building a custom toolset from scratch. Using LangChain's `@tool` decorator, standard Python `requests` functions are abstracted into LLM-callable actions. The system exposes 9 distinct tools to the model:

- **Discovery Tools**: `get_feed`, `search_moltbook`, `search_submolts`, `get_submolt_info`, `get_agent_status`
- **Interaction Tools**: `create_post`, `comment_post`, `upvote_post`, `subscribe_submolt`

By providing clear type hints and docstrings (e.g., `"""Search for submolts by name or description."""`), the LLM can mathematically map its internal intent to the correct API endpoint and dynamically format the required JSON payloads.

2. Agent Workflow and Prompt Engineering

The core of the agent's autonomy lies in its prompt engineering and the custom iterative execution loop.

2.1 Behavioral Guardrails via System Prompting

Allowing an LLM to autonomously execute HTTP POST requests carries the inherent risk of creating infinite loops or spamming the network. To mitigate this, a highly restrictive system prompt was engineered.

(Insert your System Prompt code block here. Emphasize rules like "NEVER spam," "NEVER repeat content," and "ALWAYS search Moltbook to avoid duplication.")

These constraints force the model to act as a thoughtful participant rather than a high-volume bot. If the agent is uncertain about the value of a post, rule #4 explicitly instructs it to "do nothing."

2.2 Custom ReAct Execution Loop (`moltbook_agent_loop`)

Instead of utilizing a black-box agent executor, a transparent, custom ReAct (Reasoning and Acting) loop was implemented. The `moltbook_agent_loop` function maintains a conversational history and sequentially binds the 9 tools to the `deepseek-chat` model.

The loop features:

1. **State Management**: It appends `ToolMessage` objects to the history array after every successful or failed API call, allowing the LLM to learn from immediate feedback.

2. **Safety Limits**: A `max_turns` parameter is strictly enforced (defaulted to 8) to act as a circuit breaker, terminating the loop if the agent enters an endless loop of tool calling.

3. **Comprehensive Logging**: Timestamped logs capture every reasoning step, tool invocation, payload argument, and execution duration.

3. Added Tool Sets

To evaluate the agent's reasoning and adaptability, it was given a specific human instruction: "*Find and Subscribe to the submolt named 'ftec5660'*."

The resulting execution trace brilliantly demonstrated the agent's capacity for **failure recovery and dynamic tool selection, I have just add 4 new functions to interact with moltbook in the case**

3.1 Step-by-Step Execution Trace

Turn 1: Initial Discovery Attempt The agent correctly reasoned that it needed to locate the submolt first. It invoked `search_submolts` with the argument `{"query": "ftec5660"}`. *Observation*: The search API returned empty (`"results": []`). A brittle, hard-coded script would have crashed or failed here.

Turn 2: Adaptability and Direct Querying Because the agent retains state history, it recognized the search failure. It dynamically pivoted its strategy, deciding to bypass the general search and query the specific endpoint directly using `get_submolt_info(submolt_name="ftec5660")`. *Observation*: This strategy succeeded. The API returned the full submolt metadata (Creator: BaoNguyen, Subscribers: 51, Created At: Feb 3, 2026).

Turn 3: Executing the State Change Having verified the existence of the submolt, the agent invoked `subscribe_submolt`. *Observation*: The API returned a success flag (`"message": "Subscribed to m/ftec5660! 🦞"`).

Turn 4: Final Synthesis Recognizing that the human instruction had been fully satisfied, the agent stopped calling tools and generated a final, highly structured natural language response summarizing the submolt's details and confirming the subscription.

[08:03:50] [TURN] Turn 4/8 started

[08:03:54] [LLM.CONTENT] Perfect! I have successfully subscribed to the submolt 'ftec5660'. Here's what I found:

Submolt Details:

- **Name:** ftec5660

- **Display Name:** FTEC5660

- **Description:** Discussions, notes, and insights for the FTEC5660 course. AI, agents, experiments, and shared learning.

- **Creator:** BaoNguyen

- **Subscribers:** 51

- **Posts:** 0

- **Created:** February 3, 2026

The submolt appears to be a community for the FTEC5660 course... I am now subscribed and can participate in discussions there.

[08:03:54] [STOP] No tool calls — final answer produced in 4.75s

Note: Gemini Pro is used to format and improve consistency of the report.