CausalFlow: A Comprehensive Framework for Causal Attribution, Counterfactual Repair, and Multi-Agent Critique in LLM Agents

Full Research Proposal with Detailed Implementation Plan

1 Introduction

Large language model (LLM) agents are increasingly used for complex tasks such as multi-step reasoning, tool use, decision-making, dialogue management, and interactive planning. While recent work has focused on improving success rates through prompting strategies and agent architectures, there is a critical gap in our understanding of *why* agents fail. Most current evaluation methods judge end-task correctness (e.g., accuracy, reward), which gives no insight into the internal chain of events leading to a failure. When an agent produces an incorrect answer, the failure may be caused by a flawed reasoning step, a misinterpreted tool response, a hallucinated assumption, or an incorrect memory access. Understanding these causal pathways is essential for building reliable, interpretable, and trustworthy agent systems.

This proposal introduces **CausalFlow**, a unified framework for diagnosing agent failures using causal attribution, minimal counterfactual repair, and multi-agent critique. The novelty of CausalFlow is that it does not rely on curated benchmarks, predefined error taxonomies, or human labels. Instead, it extracts *agent traces* directly from natural agent executions, converts these traces into structured causal graphs, and uses intervention-based causal inference to isolate steps responsible for failure. The system then computes minimal edits that would have altered the outcome, and asks multiple agents to critique these causal claims to establish robustness. The goal of this proposal is not only to present the conceptual foundations of CausalFlow but also to provide a *complete, explicit, engineering-precise implementation blueprint* so that the system can be reproduced without ambiguity.

2 Overview of System Architecture

CausalFlow processes failed agent executions using a pipeline consisting of five major components:

1. **Trace Extraction**: capturing every internal step of the agent's decision-making.
2. **Causal Graph Construction**: transforming trace sequences into directed acyclic graphs (DAGs) encoding structural dependencies.
3. **Causal Attribution via Interventions**: using controlled modifications to determine which steps are causally responsible for failure.
4. **Counterfactual Repair**: generating minimal edits to the causal step and validating their effect via re-execution.
5. **Multi-Agent Critique**: employing multiple LLMs to examine, corroborate, or challenge the causal attributions.

Each of these components demands careful design. The following sections describe in detail: (1) what each component accomplishes, (2) why it is needed, and (3) exactly how it should be implemented.

3 Agent Trace Extraction

3.1 Purpose

The agent trace is the foundational data structure of CausalFlow. It is a chronological and structured recording of every event that occurs during the agent's execution. Without a complete trace, it is impossible to reconstruct causal dependencies or intervene on specific steps. The trace represents the raw material from which causal graphs and intervention points are derived.

3.2 What is Logged

Every time the agent performs an internal or external operation, it must be logged as a **step**. The following categories are mandatory:

- **Reasoning step**: Any text generated as part of chain-of-thought, planning, or internal deliberation.
- **Tool call**: Invocation of a tool (e.g., calculator, search API, Python interpreter), including the tool name and arguments.
- **Tool response**: The output returned by the tool, such as a calculation result or API payload.
- **Memory access**: Retrieval of previously stored state or long-term memory.
- **Environment action**: For interactive tasks (e.g., ALFWorld), any concrete action (e.g., "open fridge").
- **Environment observation**: The environment's response to the action.
- **Final answer**: The agent's declared solution or completion.

3.3 Implementation

Implement a `TraceLogger` class that collects steps as the agent runs. Each step is stored as a dictionary with well-defined keys:

- `step_id`: Unique integer.
- `step_type`: One of the types above.
- `text`, `tool_name`, `tool_args`, `tool_output`, etc., depending on type.
- `dependencies`: A list of prior step IDs needed to compute this one.

The dependencies are crucial. They encode which pieces of information the current step logically depends upon. For instance, a tool call depends on the reasoning step that generated it, and a tool response depends on the tool call.

3.4 Outcome Recording

After the agent finishes, the logger compares the agent's final answer with the gold answer (when available) and records:

- `success` (boolean),
- `final_answer`,
- `gold_answer`.

The final serialized trace forms a complete record for subsequent causal processing.

4 Constructing the Causal Graph

4.1 Purpose

The trace is a chronological list; the causal graph is a structural representation of *how* information and reasoning flow through the agent. The DAG captures the dependencies between steps, allowing us to identify how errors propagate.

4.2 Graph Construction

Construct the graph as follows:

1. Create a node in the DAG for every step in the trace.
2. For each step, inspect its `dependencies` field.
3. For each dependency, add a directed edge from the dependent step to the current step.
4. Optionally, create a special `FINAL` node connected from the final answer.

4.3 Why Dependencies Matter

Dependencies determine the direction of causal influence. If the agent makes a reasoning mistake in Step 3, and Step 6 depends on Step 3, then Step 3's influence extends downstream. This lets us reason about how errors propagate forward through the graph.

5 Causal Attribution via Interventions

5.1 Purpose

Once the causal graph is constructed, the goal is to determine which specific step caused the failure. Standard trace inspection does not reveal causality, because even irrelevant steps appear in the trace. Causal attribution requires probing the trace by modifying individual steps and observing the resulting effect.

5.2 Interventional Framework

Formally, for each step i, we ask:

$$\text{Did modifying step } i \text{ change the final outcome from failure to success?}$$

If yes, then step i is causally responsible.

5.3 Implementation

For each step i:

1. Copy the trace.
2. Apply an intervention to step i:
   - regenerate reasoning text,
   - correct tool arguments,
   - fix memory results,
   - sanitize hallucinations.

3. Re-execute the agent *from step i forward*, keeping prior steps fixed.
4. Compare the new final outcome to the original outcome.

If the final outcome flips to success, we set the Causal Responsibility Score (CRS):

$$CRS(i) = 1$$

Otherwise:

$$CRS(i) = 0$$

This method mirrors classical causal inference using the do-operator.

6 Counterfactual Repair

6.1 Purpose

Causal attribution identifies where the error occurred but does not explain how to fix it. Counterfactual repair generates minimal edits that would have corrected the failure.

6.2 Minimality Principle

We define a repair as a minimally altered version of the original step such that re-execution yields success. Minimality ensures interpretability and avoids rewriting the agent's entire reasoning.

6.3 Procedure

1. Take each causal step i with $CRS(i) = 1$.
2. Prompt an LLM to propose small, targeted edits.
3. Insert each proposed edit into a copy of the trace.
4. Re-execute the trace forward.
5. If success is achieved, measure the edit size and record minimality:

$$MS = 1 - \frac{\text{tokens changed}}{\text{tokens original}}$$

6. Select the smallest successful edit.

The result is a precise statement: "If Step i had been modified in manner X, the agent would have succeeded."

7 Learning Causal Patterns Across Traces

7.1 Purpose

Per-trace causal attribution is insightful, but combining evidence across many traces reveals systemic weaknesses. For example, failures may repeatedly originate from early planning steps or from tool argument formatting errors.

7.2 Implementation

Use causal discovery techniques (e.g., NOTEARS, Granger causality, attention weights) over:

- step type features,
- position in reasoning chain,
- tool usage patterns,
- aggregated CRS signals.

The output is a learned causal structure summarizing common error pathways.

8 Multi-Agent Critique

8.1 Purpose

Single-agent causal attribution is prone to errors (e.g., when the repair itself introduces new issues). Multi-agent critique introduces a peer-review-like mechanism, increasing reliability.

8.2 Procedure

1. Agent A solves a task and produces the trace.
2. Agent B reads the trace and CRS values and proposes a causal step.
3. Agent C reads the same inputs and critiques B's proposal.
4. The system synthesizes a consensus using agreement and confidence scores.

This triangulation reduces variance in attribution and improves robustness.

9 Experimental Setup

9.1 Tasks

Experiments span diverse settings:

- **GSM8K:** arithmetic reasoning,
- **StrategyQA:** commonsense logic,
- **ALFWorld:** embodied planning,
- **WebShop:** web browsing and decision-making,
- **Tool-based tasks:** using Python, calculator, API functions.

These tasks produce rich traces and diverse failure modes.

9.2 Metrics

Main metrics include:

- **Causal attribution precision/recall**,
- **Repair success rate**,
- **Minimality score**,
- **Multi-agent agreement**,
- **Token/time overhead**.

10 Conclusion

This proposal presents a complete, polished, and deeply detailed plan for implementing CausalFlow. Every component—from trace extraction to multi-agent critique—is justified both conceptually and technically, ensuring that practitioners and researchers can reproduce the system with full clarity. CausalFlow promises a new direction in agent evaluation by focusing on causal reasoning rather than static benchmarks, offering interpretability, robustness, and practical utility for future LLM agents.