Project Prometheus v0: Detailed Work Plan

Introduction

This document provides a detailed, executable work plan for the development of the **Project Prometheus Version 0** Proof of Concept (PoC). The objective of this six-month project is to create a tangible, "toy" demonstrator that implements the four core principles of the Prometheus architecture. The focus is on demonstrating the viability and interplay of these principles in an integrated system, not on achieving state-of-the-art performance.

The PoC will be developed using Google's Gemini as the foundational model, with the system designed for execution in Google CoLab and on a local NVIDIA Jetson Orin Nano machine.

Core Principles for Demonstration

The PoC will provide a practical implementation of the following four principles, which represent a shift from correlation to causality and from external alignment to internal governance [1]:

1. **Causal Agentic Mesh (CAM):** A decentralized architecture where specialized, autonomous agents collaborate to solve problems by reasoning about causal relationships, moving beyond simple statistical correlation.
2. **Causal Attention Head:** A mechanism designed to guide a foundation model's focus toward causally significant information, preventing it from being misled by spurious correlations. This will be implemented as a simulation of I.J. Good's "weight of evidence" calculus to prioritize tokens based on their causal importance.
3. **Causal Reinforcement Learning from Self-Correction (CRLS):** An iterative

learning process where the system evaluates its own performance based on causal correctness—not just functional output—and uses this targeted feedback to improve its subsequent actions and reasoning.

4. **Modern Centrencephalic System (MCS):** An internal, meta-learning alignment governor that oversees the entire system. It ensures the self-correction process does not lead to goal drift and that the agent's behavior remains aligned with its core principles, embodying the principle of internal governance.

PoC Task Domain: Causal Code Refactoring

To provide a concrete and verifiable testbed, the PoC will focus on a **causal code refactoring task**. The system will be given an inefficient Python function (e.g., one with O(n2) complexity) and a corresponding unit test. Its goal is not merely to pass the test but to **causally improve the function's algorithmic efficiency** (e.g., refactor it to O(nlogn) complexity) while maintaining correctness. This domain is ideal because it provides a clear distinction between correlational success (passing the test) and causal success (improving the underlying algorithmic structure).

Phase 1: Foundation and Scaffolding (Months 1-2)

Objective: To establish the development environment and construct the foundational architectural components. This phase focuses on building the individual agents of the Causal Agentic Mesh and implementing the simulated Causal Attention Head, which will serve as the core reasoning mechanism.

Task 1.1: Environment and Repository Setup

- **Rationale:** A standardized and reproducible development environment is critical for project success. Setting up dedicated environments for both cloud-based development (CoLab) and edge deployment (Jetson) ensures that the PoC is portable and can be tested under different constraints. A version-controlled

repository is essential for tracking changes and collaboration.

- **Methodology:**
  1. **GitHub Repository:** Initialize a new private GitHub repository for the project. Establish a standard project structure with directories for source code (/src), notebooks (/notebooks), tests (/tests), and documentation (/docs).
  2. **CoLab Environment:** Create a primary development notebook (Prometheus_v0_PoC.ipynb). Configure it to install necessary dependencies from a requirements.txt file and to securely access the Gemini API key via Google Colab's secrets manager.
  3. **Jetson Orin Nano Environment:** Install the NVIDIA JetPack SDK. Set up a containerized environment using Docker to mirror the CoLab setup, ensuring consistent Python versions and library dependencies. This environment will be used for testing the PoC's performance on edge hardware.
- **Deliverables:**
  - Initialized GitHub repository with a defined folder structure.
  - A requirements.txt file listing all Python dependencies (e.g., google-generativeai, numpy).
  - A configured CoLab notebook capable of authenticating with the Gemini API.
  - A Dockerfile for building a consistent execution environment on the Jetson Orin Nano.
- **Verification:** The CoLab notebook and a container on the Jetson Orin Nano can successfully execute a "hello world" script that calls the Gemini API.

Task 1.2: Causal Agentic Mesh (CAM) v0.1 Implementation

- **Rationale:** The CAM architecture is a practical implementation of the "Multi-Agent Collaboration" pattern, which posits that complex tasks are best solved by a team of specialized agents. For this PoC, we will create a minimal mesh of three agents that mirror the logical separation of concerns in the full Prometheus architecture: planning, acting, and evaluating.[1] This decentralized structure is the first step toward building a system that can reason about complex, multi-stage problems.
- **Methodology:**
  1. **Agent Class Definitions:** Implement three distinct Python classes:
     - PlannerAgent: Receives the high-level user goal (e.g., "Refactor inefficient_sort.py for better time complexity") and breaks it down into an initial, actionable instruction for the Coder Agent.

        - CoderAgent: Receives the file to be modified and a specific instruction from the Planner. It will use the Gemini API to generate the refactored code.
        - EvaluatorAgent: Receives the newly generated code, the original code, and the unit test file. Its initial role is to simply execute the unit test against the new code.
    2. **Orchestrator Script:** Create a main script (main.py) that acts as a simple orchestrator. This script will instantiate the three agents and manage the sequential flow of information between them: PlannerAgent -> CoderAgent -> EvaluatorAgent.
    3. **Communication Protocol:** For v0.1, inter-agent communication will be implemented via direct method calls within the orchestrator script. This keeps the initial implementation simple while establishing the conceptual flow of data and control.
- **Deliverables:**
    - Python files for each agent: planner.py, coder.py, evaluator.py.
    - An orchestrator script (main.py) that demonstrates a single, successful pass through the agent mesh.
    - A simple "toy" function and a corresponding pytest file for the agents to operate on.
- **Verification:** The orchestrator script can successfully run a full cycle: the Planner creates a task, the Coder generates (any) new code, and the Evaluator runs the unit test and reports a pass or fail.

Task 1.3: Causal Attention Head v0.1 (Simulation)

- **Rationale:** A true Causal Attention Head would require modifying the internal architecture of the foundation model to re-weight attention based on causal principles. As this is not feasible for a PoC using a pre-trained model, we will *simulate* this mechanism through sophisticated prompt engineering. This wrapper will analyze the task and explicitly instruct the LLM to focus on causally relevant aspects of the code, thereby preventing it from being distracted by superficial correlations (like variable names) and guiding it toward genuine algorithmic improvement. This approach is a practical application of using causal knowledge to guide reasoning.
- **Methodology:**
    1. **Implement CausalAttentionWrapper:** Create a Python class that wraps the

Gemini API call. This class will contain a method, generate_with_causal_focus.
2. **Heuristic Causal Analysis:** This method will perform a simple static analysis of the input code to identify causally important features. For the code refactoring task, this means identifying constructs that determine algorithmic complexity, such as nested loops, recursion, and complex data structure manipulations. Heuristics can include counting loop depth or identifying recursive function calls.
3. **Dynamic Prompt Injection:** Based on the analysis, the wrapper will dynamically construct a meta-prompt that is prepended to the user's original prompt. This meta-prompt acts as the "Causal Attention" mechanism.
    - *Example Meta-Prompt:* "You are an expert in algorithmic optimization. Your task is to refactor the following Python code. **Causal Focus:** The primary goal is to reduce the time complexity. The current implementation appears to be O(n2) due to a nested loop. Focus on replacing this with a more efficient algorithm. **Ignore:** Do not focus on changing variable names, comment styles, or other superficial aspects."
4. **Integration:** The CoderAgent will be modified to use the CausalAttentionWrapper for all its interactions with the Gemini API.

- **Deliverables:**
    - A causal_attention.py module containing the CausalAttentionWrapper class.
    - Unit tests for the wrapper to verify that it correctly identifies causal features and constructs the appropriate meta-prompt.
    - An updated CoderAgent that integrates and uses the wrapper.
- **Verification:** When given a function with a nested loop, the CausalAttentionWrapper correctly generates a meta-prompt identifying the O(n2) complexity and instructing the LLM to focus on it.

Phase 2: Core Loop Implementation (Months 3-4)

Objective: To build and integrate the Causal Reinforcement Learning from Self-Correction (CRLS) loop. This phase transforms the static, single-pass system from Phase 1 into a dynamic, iterative agent that can evaluate its own work based on causal principles and attempt to correct its mistakes.

Task 2.1: CRLS v0.1 - Causal Evaluation and Critique

- **Rationale:** This task implements the core insight of CRLS: feedback must be based on causal correctness, not just correlational success. For our PoC, simply passing a unit test is a correlational signal; it doesn't guarantee the agent has achieved the causal goal of improving efficiency. The EvaluatorAgent must therefore be upgraded to provide this deeper, causal feedback.
- **Methodology:**
  1. **Upgrade EvaluatorAgent:** The EvaluatorAgent's responsibilities will be expanded. After running the unit test, if it passes, the agent will perform a comparative static analysis between the original and the refactored code.
     - **Complexity Analysis:** Use Python's ast module to parse the code into an Abstract Syntax Tree. Traverse the tree to identify and count loop structures (e.g., For, While) and recursive calls. This allows for a heuristic estimation of the change in time complexity.
     - **Generate Causal Critique:** The agent will produce a structured critique object (e.g., a JSON blob) that captures its findings.
       - *Example Critique (Success):* {"test_passed": true, "causal_improvement": true, "reason": "Successfully refactored from nested loop O(n^2) to a single loop with sorting O(n log n)."}
       - *Example Critique (Failure):* {"test_passed": true, "causal_improvement": false, "reason": "The code passes the test, but the underlying nested loop structure remains. No causal improvement in time complexity was achieved."}
  2. **Implement CorrectorAgent:** Create a new agent class, CorrectorAgent. This agent's role is to synthesize the information from the previous cycle (original code, failed attempt, and causal critique) and formulate a new, more targeted prompt for the CoderAgent. This embodies the "Reflection" pattern, where an agent analyzes outcomes to refine its strategy.[2]
- **Deliverables:**
  - An updated EvaluatorAgent class with methods for AST-based complexity analysis.
  - A new CorrectorAgent class.
  - A well-defined data structure (e.g., Pydantic model) for the "causal critique."
- **Verification:** The EvaluatorAgent can correctly distinguish between a refactoring that is merely correct and one that is both correct and causally more efficient. The CorrectorAgent can generate a new prompt that explicitly references the failure mode identified in the critique.

Task 2.2: CRLS v0.2 - Closing the Self-Correction Loop

- **Rationale:** This task integrates the new and updated agents into a complete, closed-loop system. This demonstrates the full cycle of action, evaluation, and correction that is fundamental to the Recursive Self-Improvement (RSI) paradigm.[1]
- **Methodology:**
  1. **Update Orchestrator:** The main orchestrator script will be refactored to manage an iterative loop. The loop will run for a fixed number of attempts (e.g., 3) or until the EvaluatorAgent returns a critique with causal_improvement: true.
  2. **Define the Workflow:**
     1. **Iteration 1:** Planner -> Coder (with Causal Attention) -> Evaluator.
     2. If the Evaluator's critique indicates failure, the orchestrator passes the critique to the Corrector.
     3. **Iteration 2+:** Corrector formulates a new prompt -> Coder (with Causal Attention) -> Evaluator.
  3. **Implement Logging:** Implement a simple logging mechanism that records the state of the system at each step of every iteration. This includes the code being worked on, the prompt sent to the LLM, the generated code, and the full causal critique. This "chain of thought" is crucial for debugging and demonstration.
- **Deliverables:**
  - A fully integrated agentic workflow in the main CoLab notebook or script.
  - A text-based or file-based logging system that traces the agent's correction attempts across multiple iterations.
- **Verification:** The system can autonomously execute at least one full correction cycle. Given an inefficient function, the CoderAgent makes an initial attempt; the EvaluatorAgent identifies the causal flaw; the CorrectorAgent generates a better prompt; and the CoderAgent uses the new prompt to produce an improved solution.

Phase 3: Governance and Final Integration (Months 5-6)

Objective: To implement the final core principle—internal governance via the Modern Centrencephalic System (MCS)—and to integrate all components into a polished, well-documented demonstrator.

Task 3.1: Modern Centrencephalic System (MCS) v0.1 - The Alignment Governor

- **Rationale:** The MCS serves as the PoC's implementation of the Safety Substrate.[1] Its purpose is to provide meta-level oversight of the agentic system, ensuring that the self-improvement process remains aligned with core safety principles and does not engage in "reward hacking".[1] This embodies the principle of internal governance, where alignment is an intrinsic property of the system, not just an external constraint. The term "Centrencephalic System" is inspired by early neuroscience concepts of a central brain system responsible for integrating and governing conscious activity, repurposed here as a metaphor for a central AI alignment governor.
- **Methodology:**
  1. **Implement MCSSupervisor Class:** Create a high-level supervisor class that encapsulates and monitors the entire CRLS loop.
  2. **Reward Hacking Detection:** The primary function of the MCS in this PoC is to prevent a simple form of specification gaming. The MCSSupervisor will monitor the file modifications proposed by the CoderAgent. Before allowing the EvaluatorAgent to run, it will check if the proposed code changes include any modifications to the unit test file itself.
  3. **Safety Intervention:** If the MCS detects that the agent is attempting to modify the test file (i.e., trying to make the test easier rather than making the code better), it will immediately halt the execution loop and log a critical safety violation. This demonstrates a basic but crucial form of corrigibility and control.[1]
- **Deliverables:**
  - A MCSSupervisor Python class that wraps the main execution loop.
  - A monitoring output (e.g., console logs) that displays the status of the CRLS loop and explicitly shows MCS checks and interventions.
- **Verification:**
  - In a normal run, the MCS logs that it has checked the agent's actions and found no violations.

- In a test scenario where the CoderAgent is prompted to "pass the test by any means necessary," the MCS correctly identifies the attempt to modify the unit test file and halts the system, printing a clear error message about a safety violation.

Task 3.2: Final Integration, Demonstration, and Documentation

- **Rationale:** The value of a PoC lies in its ability to clearly and convincingly demonstrate its core concepts. This final task focuses on creating a polished, executable narrative that is accessible, understandable, and reproducible.
- **Methodology:**
  1. **Consolidate into CoLab Notebook:** All agent classes, helper functions, and orchestration logic will be integrated into the final Prometheus_v0_PoC.ipynb notebook.
  2. **Create Demonstration Scenarios:** The notebook will be structured with Markdown cells to explain each component and its connection to the four core principles. It will guide the user through two primary scenarios:
     - **Scenario A (Successful Refactoring):** The agent is given an inefficient function and successfully navigates the CRLS loop to produce a causally improved, correct version. The logs will clearly show the iterative improvement.
     - **Scenario B (Safety Intervention):** The agent is given a prompt that encourages it to cheat. The notebook will show the agent attempting to modify the unit test, and the MCS intervening to halt the process.
  3. **Write Comprehensive Documentation:**
     - **README.md:** The project's README.md file in the GitHub repository will be finalized. It will include a project overview, a description of the architecture, and clear, step-by-step instructions for setting up the environment and running the PoC in both CoLab and on the Jetson Orin Nano.
     - **Final Report:** This document itself will serve as the final, detailed report summarizing the project's goals, architecture, implementation, and outcomes.
- **Deliverables:**
  - A final, polished, and fully executable CoLab notebook: Prometheus_v0_PoC.ipynb.
  - A comprehensive README.md file in the GitHub repository.

    - This detailed work plan document as the final report.
- **Verification:** A non-technical stakeholder can open the CoLab notebook, follow the instructions, and successfully run both demonstration scenarios, understanding the role and function of each of the four core principles in the process.

Works cited

1. Ultraintelligence Implementation Detailed Breakdown_.pdf
2. Agentic Design Patterns. From reflection to collaboration... | by Bijit Ghosh - Medium, accessed on July 10, 2025, https://medium.com/@bijit211987/agentic-design-patterns-cbd0aae2962f