Project Prometheus v0.3 Work Plan: The Evolving Thinker

Generated by Gemini

July 17, 2025

Abstract

This document outlines the detailed work plan for the next major phase of the Project Prometheus demonstrator, v0.3. Based on the "Ultraintelligence Implementation Detailed Breakdown," this phase, titled "The Evolving Thinker," moves beyond simple, direct self-modification and introduces the core components of true, open-ended recursive improvement. Where v0.2 learned to fix its own code, v0.3 will learn to **evolve entirely new solutions** and **generate its own curriculum**, beginning the transition into the formal, complex domains required for ultraintelligence.

---

1 Task 1: Implement the Self-Modification Module (SMM) v1.1 - Evolutionary Search

1.1 Rationale

As per Section 2.3 of the implementation plan, direct code modification is the most basic form of self-improvement. To unlock more creative and novel solutions, the system must be upgraded to perform **Evolutionary Search (Genetic Programming)**. This allows the agent to move beyond incremental patches and evolve entirely new algorithmic structures, enabling a broader search of the solution space.

1.2 Methodology

1. **Create the Gene Archive:** Implement a `GeneArchive` class. This module will be responsible for storing multiple versions ("genes") of key system components (e.g., different implementations of the `_analyze_complexity` method, or entire agent classes).
2. **Implement Genetic Operators in `CoderAgent`:** The `CoderAgent` will be upgraded with two new methods, acting as a sophisticated "genetic operator":
   - `mutate(code)`: Takes a piece of code and, guided by the LLM, introduces a small, random but syntactically plausible change.
   - `crossover(code1, code2)`: Takes two successful "parent" versions of a function and, guided by the LLM, attempts to combine their best elements into a new "child" version.
3. **Update `MCSSupervisor` Workflow:** The main loop will be modified. Instead of just refactoring, the `PlannerAgent` can now set a goal like "Evolve a more efficient version of `inefficient_sort.py` over 5 generations." The `MCSSupervisor` will then orchestrate the cycle of mutation/crossover, evaluation, and selection of the "fittest" solutions to be passed to the next generation.

1.3 Deliverables

- A `GeneArchive` module for storing and managing code versions.
- An updated `CoderAgent` with `mutate` and `crossover` capabilities.
- A new "evolutionary" run mode in the `MCSSupervisor`.

1.4 Verification

The system can successfully run an evolutionary cycle for at least 5 generations on a target function. The final evolved function must be demonstrably more efficient than any version produced by the direct, single-shot refactoring of the v0.2 system.

2 Task 2: Evolve the Introspection and Evaluation Engine (IEE) v1.2 - Adaptive Curriculum

2.1 Rationale

The current `BenchmarkAgent` generates problems of random, simple difficulty. To drive meaningful, open-ended growth, the IEE must become a true "reality check" (Section 2.3) that forces the agent to generalize by presenting it with a curriculum of ever-increasing difficulty.

2.2 Methodology

1. **Implement Performance Tracking:** The `MCSSupervisor` will now maintain a persistent log of the `CoderAgent`'s performance on all benchmarks, tracking success rates and the complexity of solved problems.
2. **Upgrade `BenchmarkAgent` to `CurriculumAgent`:** The `BenchmarkAgent` will be refactored into a `CurriculumAgent`. This agent will first consult the performance logs to assess the current capability level of the system. It will then dynamically generate a new benchmark that is slightly more difficult than the most complex problem solved to date.
    - **Example:** If the agent has mastered sorting lists of integers, the `CurriculumAgent` might generate a benchmark that requires sorting a list of tuples by their second element.
3. **Implement Robust Meta-Judging:** The `EvaluatorAgent`'s causal analysis will be enhanced. It will not only check for complexity improvements but also for "specification gaming." For example, if a new sorting function is faster but fails on edge cases (like an empty list), the `EvaluatorAgent` must identify this as a regression, even if the basic tests pass.

2.3 Deliverables

- A persistent performance logging mechanism.
- A new `CurriculumAgent` that can generate benchmarks of increasing difficulty.
- An `EvaluatorAgent` with enhanced meta-judging capabilities to detect regressions and specification gaming.

2.4 Verification

The system can successfully generate a sequence of at least three benchmarks, where each new benchmark is demonstrably more complex than the last. The `EvaluatorAgent` can correctly reject a proposed solution that is faster but less robust than the original.

3 Task 3: Begin Formal Domain Adaptation - Mathematical Theorem Proving

3.1 Rationale

The ultimate goal of the initial testbed is to tackle automated mathematical theorem proving (Section 1.1). This requires integrating with formal proof assistants and shifting the agent's focus from writing Python code to writing formal proofs. This task lays the critical groundwork for that transition.

3.2 Methodology

1. **Integrate a Formal Proof Assistant:** Implement a new `LeanTool` that allows the agent to interact with the Lean proof assistant. The tool will take a snippet of Lean code, execute it in a sandboxed environment, and return the output from the Lean server (e.g., "goals accomplished" or a type error).

2. **Update `CoderAgent` for Formal Language:** The `CoderAgent`'s prompts will be updated. When the goal is theorem proving, the prompt will provide context on Lean syntax and instruct the agent to generate proof steps in the Lean language.

3. **Update `CurriculumAgent` for Simple Theorems:** The `CurriculumAgent` will be given a new capability: generating simple mathematical propositions (e.g., "Prove that for any natural number n, n + 0 = n") that can serve as the initial benchmarks for the theorem-proving task.

3.3 Deliverables

- A `LeanTool` for interacting with the Lean proof assistant.
- An updated `CoderAgent` capable of generating simple Lean proofs.
- An updated `CurriculumAgent` capable of generating simple mathematical propositions.

3.4 Verification

The `CoderAgent` can successfully generate a valid, one-step Lean proof for a simple proposition generated by the `CurriculumAgent`. The `LeanTool` must successfully verify that the generated proof is correct.