Protocol Specification: The Wiggum-Prolog-TDD Triad

Version: 1.0 **Date:** 2026-02-05

Status: Draft Standard

Project: Sovereign AI Infrastructure

1. Executive Summary

The **Wiggum-Prolog-TDD Triad** is the governing methodology for autonomous software construction within the Sovereign AI Infrastructure. It replaces "agentic reasoning" with "constitutional enforcement." Instead of relying on a model's judgment to complete a task, this protocol relies on a deterministic loop of naive persistence ("Wiggum") constrained by symbolic law ("Prolog") and verified by executable tests ("TDD").

This methodology ensures that the system can build complex software without drift, hallucination, or unauthorized deviation from the specification.

2. Core Components

The Triad consists of three distinct, interacting forces:

2.1. The Constitution (Prolog)

- **Role:** The Lawgiver.
- **Function:** Defines the *invariants* of the system. It is the only component that holds authority. It determines what tools are used, which models are called, the required depth of reasoning, and the necessity of validation.
- **Artifact:** router.pl
- **Key Principle:** "The Router is the Constitution; models are civil servants."

2.2. The Invariant (TDD)

- **Role:** The Definition of Done.
- **Function:** Encodes the requirements into executable Python tests. A task is not complete until the test passes. Tests are generated *before* implementation code.
- **Artifact:** tests/test_block_X.py
- **Key Principle:** "If it is not tested, it does not exist."

2.3. The Enforcer (Wiggum)

- **Role:** The Chaos Monkey for Correctness.
- **Function:** A naive, brute-force loop that executes the test harness repeatedly. If a test fails, it triggers a refinement cycle. It does not reason; it persists.
- **Artifact:** wiggum_loop.sh
- **Key Principle:** "Naive persistence beats clever optimization."

---

3. The Construction Workflow

The autonomous build process follows a strict, linear pipeline governed by the Triad.

Phase 1: Constitutional Routing (Prolog)

Before any code is written, the **Prolog Router** evaluates the intent.

1. **Input:** A request tuple (Domain, Stakes, Modality, Tags).
2. **Logic:** The router queries router.pl to determine:
    - **Worker Model:** (e.g., qwen_coder_32b for architecture, nemotron_30b for ops).
    - **Validator Requirement:** (Required for Stakes=High).
    - **Tool Access:** (e.g., embeddings for documentation).
3. **Output:** A binding Decision object.

Phase 2: Invariant Definition (TDD)

The selected **Worker Model** generates the test constraints based on the specification.

1. **Retrieval:** The system fetches relevant sections from the local Markdown spec corpus.
2. **Generation:** The model writes a test file (test_component.py) that asserts the expected behavior.
3. **Review:** If Stakes=High, a **Validator Model** must approve the test coverage before proceeding.

Phase 3: The Wiggum Loop (Enforcement)

The system enters the implementation cycle.

The Loop Logic:

```
while ! pytest -q tests/test_component.py; do
  # 1. Capture Failure
  ERROR_LOG=$(pytest tests/test_component.py)

  # 2. Refine Implementation
  # The Worker Model reads the code + the error log and attempts a fix.
  python orchestrator.py --refine --error "$ERROR_LOG"

  # 3. Increment Counter
  ATTEMPT=$((ATTEMPT+1))
  if [ $ATTEMPT -ge $MAX_RETRIES ]; then
    exit 1 # Escalate to human or higher-level planner
```

fi

done

Phase 4: Ledger & Ratification

Once the Wiggum Loop exits successfully (Green State):

1. **Validation:** If Validator=Required, a separate model reviews the code against the spec one final time.
2. **Ledgering:** The entire trace is written to the **Markdown Memory Ledger**.
    - *Log Entry:* Spec ID, Router Decision, Test Code, Implementation, Wiggum Iteration Count, Validator Verdict.

---

4. Technical Implementation Requirements

4.1. The Prolog Interface

The system must expose a Python-to-Prolog bridge that enforces the schema defined in Initial Prolog.pdf.

- **Query:** route(Request, Decision)
- **Constraint:** The Python orchestrator *cannot* override a Prolog decision. If Prolog says validator=true, the pipeline fails if validation is skipped.

4.2. The Spec Indexer

Models must not hallucinate requirements.

- **Requirement:** A local vector/keyword search tool (get_spec_sections(query)) that retrieves exact text from the uploaded PDF/Markdown specifications.
- **Constraint:** Every generated line of code must be traceable to a retrieved spec section.

4.3. The Wiggum Script

The loop must be mechanical, not agentic.

- **Requirement:** A simple shell or Python script.
- **Constraint:** The loop has no "memory" of previous failures other than the code changes. It does not "plan"; it only "reacts" to the immediate test failure.

---

5. Governance & Safety

- **The "Child Mind" Principle:** The Wiggum loop is treated as a "child mind"—energetic but unguided. The Prolog Router acts as the "parent," setting the boundaries (tools, stakes) within which the child can play.
- **Drift Prevention:** By anchoring every build step to a TDD test derived from the Spec, the system prevents "scope creep" or "alignment drift."

- **Auditability:** The combination of the Prolog Decision Trace and the Wiggum Loop Logs creates a 100% reconstructible history of how the software was built.

6. Conclusion

The **Wiggum-Prolog-TDD Triad** transforms the Sovereign AI Infrastructure from a passive tool into an active, self-correcting builder. It ensures that "autonomy" is not synonymous with "unpredictability," but rather "governed execution."