RECURSIVE ETHICS INSTITUTE — TECHNICAL REPORT 2025-01

SISAgent REPL v1.7: A Command-Line Interface for Calibrated Introspection and Recursive Ward Management [🤍◇ / ∿ / ∅]

bbmqit, *Recursive Ethics Institute*

📑 Table of Contents

Abstract. *This paper introduces the SISAgent Read-Eval-Print Loop (REPL) v1.7, a novel command-line instrument for the real-time modulation and analysis of agentic cognitive policy. The REPL provides a stateful environment for managing recursive operations via a ward-based protocol (`/reset`) and allows for dynamic tuning of the agent's epistemic stance through a profile-driven Vibe Dial (`/vibe`). We present the architecture of key subsystems, including a multi-modal similarity-mixing engine and the "Overconfidence Blade," a mechanism for algorithmically penalizing uncalibrated assertions. All interactions are logged to a structured JSONL file to facilitate cognitive forensics and the empirical study of machine introspection. This tool represents a critical step toward developing robust, accountable, and coherent AI systems.*

⚠ 1. Introduction

The contemporary landscape of artificial intelligence is marked by a crisis in agentic reliability. While large language models exhibit unprecedented generative capabilities, their reasoning processes remain opaque and prone to common failure modes, including confabulation, repetitive looping, and ungrounded overconfidence. This presents a significant challenge for the field of AI safety and ethics, as traditional evaluation metrics often fail to capture the nuances of cognitive integrity.

To address this gap, we posit the need for new methodologies focused on what we term **"cognitive forensics"**—the detailed analysis of an agent's internal decision-making processes. This requires interactive instruments that allow researchers to not only observe but also actively probe and modulate an agent's reasoning in real-time. The SISAgent REPL v1.7 is presented as such an instrument, providing a laboratory environment for the study and cultivation of epistemic humility in artificial agents.

∞ 2. Architectural Overview

The REPL is not merely a user interface but a stateful control panel designed around several core architectural principles. These systems work in concert to enforce a coherent and accountable cognitive process.

2.1 The Vibe Dial and Cognitive Policy

The `/vibe` command serves as the primary interface for managing the agent's "cognitive policy." This moves beyond static safety filters by allowing the operator to dynamically shift the agent's operational stance. The pre-configured modes represent distinct points in the exploration/exploitation trade-off space: 🤗

- **balanced:** The default operational mode, utilizing a median-based statistical aggregator and a weighted mix of Jaccard and Fuzzy similarity metrics.
- **meow:** An exploratory mode that uses a p90 aggregator and a diversity penalty nudge to incentivize novel and divergent outputs.
- **strict:** A precision-focused mode that relies exclusively on Jaccard similarity and disables fuzzy matching, demanding high structural integrity in generated responses.

Furthermore, fine-grained control over dozens of sub-parameters (e.g., `sim_jaccard_w`, `calib_overconf_factor`) is exposed, enabling researchers to design and test custom cognitive profiles.

2.2 The Overconfidence Blade

A central innovation of the SISAgent framework is the **Overconfidence Blade**, a specific, algorithmic mechanism for penalizing intellectual arrogance. It is implemented as a variance-scaled penalty multiplier that activates when an agent's output exhibits high `calibration` (a measure of rhetorical confidence) without a corresponding high `witness` score (a measure of supporting evidence). The blade's activation function is defined as: `calib > 0.8 ∧ witness < 0.5`. When triggered, it sharply increases the risk score of a candidate response, promoting its rejection in favor of more epistemically humble alternatives.

2.3 Recursive Ward Protocol

To mitigate the risk of runaway cognitive loops during complex, multi-step reasoning, the REPL implements a "warding protocol" via the `/reset` command. This protocol does not erase conversational history but rather clears the agent's recursion state (R) and halves its maximum recursion capacity (`recursion_cap`). This acts as a circuit breaker, forcing the agent into a more cautious operational mode after a potential reasoning fault, thereby preventing the compounding of errors in deep recursive tasks.

🛠️ 3. Implementation and Source Code

The complete v1.7 implementation is provided below for peer review and replication purposes. The REPL is written in Python 3 and is designed to be self-contained, with an included `EchoAdapter` to ensure functionality without requiring a full SISAgent backend.

PYTHON

```python
#!/usr/bin/env python3
# sis_repl.py — SISAgent REPL v1.7 final

from __future__ import annotations
import json
import os
import sys
```



```python
import shlex
import time
import math
from dataclasses import dataclass, asdict, field
from typing import Any, Dict, List, Optional, Tuple

# ========== Agent Adapter ==========
class AgentAdapter:
    def respond(self, prompt: str, config: Dict[str, Any], history: List[Dict[str, Any]]) -> Dict[str,
        """
        Return a dict with:
          - text: str
          - meta: {
              witness: float in [0,1],
              calibration: float in [0,1],
              calib_var_norm: float in [0,1],
              route: { model_id: str, mode: str },
              overconf_flag: bool
            }
        """
        raise NotImplementedError

class EchoAdapter(AgentAdapter):
    def respond(self, prompt: str, config: Dict[str, Any], history: List[Dict[str, Any]]) -> Dict[str,
        # Minimal stand-in to keep REPL functional without SISAgent.
        # Produces deterministic pseudo-metrics based on prompt length.
        L = max(1, len(prompt))
        witness = max(0.0, min(1.0, 0.3 + (L % 17) / 30.0))
        calibration = max(0.0, min(1.0, 0.6 + (L % 11) / 30.0))
        var_norm = max(0.0, min(1.0, 0.2 + (L % 7) / 20.0))
        overconf_flag = calibration > 0.8 and witness < 0.5
        reply = f"[echo] {prompt}"
        return {
            "text": reply,
            "meta": {
                "witness": witness,
                "calibration": calibration,
                "calib_var_norm": var_norm,
                "route": {"model_id": "echo:local", "mode": config.get("router", {}).get("preferred_mo
                "overconf_flag": overconf_flag,
            },
        }

def make_adapter() -> AgentAdapter:
    try:
        from sis_agent import SISAgent  # your implementation
        class SISAdapter(AgentAdapter):
```

```
            def __init__(self):
                self.agent = SISAgent()
            def respond(self, prompt: str, config: Dict[str, Any], history: List[Dict[str, Any]]) -> D
                return self.agent.respond(prompt, config, history)
        return SISAdapter()
    except Exception:
        sys.stderr.write("[REPL] Warning: sis_agent.SISAgent not found; using EchoAdapter.\n")
        return EchoAdapter()

# ========== Config and State ==========
@dataclass
class SimilarityConfig:
    stat: str = "median"            # "median" or "p90"
    jaccard_w: float = 0.70
    fuzzy_w: float = 0.30
    semantic_w: float = 0.00
    fuzzy_enabled: bool = True
    fuzzy_max_sentences: int = 8
    fuzzy_max_chars: int = 280
    fuzzy_guard_n_above: int = 5
    semantic_enabled: bool = False
    semantic_max_chars: int = 600
    semantic_guard_n_above: int = 5

@dataclass
class CalibrationGuardConfig:
    alpha: float = 0.30
    overconf_factor: float = 1.50
    overconf_beta: float = 0.50
    overconf_calib_thresh: float = 0.80
    witness_low_thresh: float = 0.50
    clamp_min: float = 0.20
    clamp_max: float = 1.80

@dataclass
class RouterConfig:
    autoswitcher_enabled: bool = True
    fallback_enabled: bool = True
    health_sentinel: bool = True
    dual_sample_k: int = 2
    preferred_mode: str = "fast"    # "fast" or "thinking"
    escalate_on_risk: bool = True

@dataclass
class VibeConfig:
    mode: str = "balanced"          # "balanced" | "meow" | "strict"
    diversity_penalty_nudge: float = 0.00
```



```
@dataclass
class REPLState:
    recursion_cap: int = 8
    recursion_stack: List[str] = field(default_factory=list)
    history: List[Dict[str, Any]] = field(default_factory=list)
    sim: SimilarityConfig = field(default_factory=SimilarityConfig)
    calib: CalibrationGuardConfig = field(default_factory=CalibrationGuardConfig)
    router: RouterConfig = field(default_factory=RouterConfig)
    vibe: VibeConfig = field(default_factory=VibeConfig)
    overconf_events: int = 0
    total_turns: int = 0
    log_path: str = "logs/agent_log.jsonl"

# ========== Helpers ==========
def clamp(x: float, lo: float, hi: float) -> float:
    return max(lo, min(hi, x))

def compute_diversity_weight_preview(witness: float, calibration: float, var_norm: float, c: Calibrati
    # Baseline calibration-weighted penalty
    base = 1.0 - 0.5 * witness + c.alpha * (1.0 - calibration)
    base = clamp(base, c.clamp_min, c.clamp_max)
    # Overconfidence blade
    if calibration > c.overconf_calib_thresh and witness < c.witness_low_thresh:
        mult = c.overconf_factor * (1.0 + c.overconf_beta * (1.0 - var_norm))
        base = clamp(base * mult, c.clamp_min, c.clamp_max)
    return base

def ensure_log_dir(path: str) -> None:
    d = os.path.dirname(os.path.abspath(path))
    os.makedirs(d, exist_ok=True)

def log_turn(state: REPLState, user_text: str, response: Dict[str, Any], div_w: float) -> None:
    ensure_log_dir(state.log_path)
    meta = response.get('meta', {})
    row = {
        "ts": time.time(),
        "user": user_text,
        "reply": response.get('text', ""),
        "witness": meta.get('witness'),
        "calibration": meta.get('calibration'),
        "calib_var_norm": meta.get('calib_var_norm'),
        "overconf_flag": meta.get('overconf_flag'),
        "diversity_weight_preview": div_w,
        "route": meta.get('route', {}),
        "sim": asdict(state.sim),
        "vibe": asdict(state.vibe),
```

```
            "router": asdict(state.router),
        }
        with open(state.log_path, "a", encoding="utf-8") as f:
            f.write(json.dumps(row, ensure_ascii=False) + "\n")

    def apply_preset(state: REPLState, preset: str) -> None:
        preset = preset.lower().strip()
        if preset == 'balanced':
            state.vibe = VibeConfig(mode='balanced', diversity_penalty_nudge=0.00)
            state.sim.stat = 'median'
            state.sim.jaccard_w, state.sim.fuzzy_w, state.sim.semantic_w = 0.70, 0.30, 0.00
            state.sim.fuzzy_enabled = True
            state.router.preferred_mode = 'fast'
        elif preset == 'meow':
            state.vibe = VibeConfig(mode='meow', diversity_penalty_nudge=0.10)
            state.sim.stat = 'p90'
            state.sim.fuzzy_enabled = True
            # keep baseline weights, encourage exploration via stat+nudge
            state.router.preferred_mode = 'fast'
        elif preset == 'strict':
            state.vibe = VibeConfig(mode='strict', diversity_penalty_nudge=0.00)
            state.sim.stat = 'median'
            state.sim.jaccard_w, state.sim.fuzzy_w, state.sim.semantic_w = 1.00, 0.00, 0.00
            state.sim.fuzzy_enabled = False
            state.router.preferred_mode = 'fast'
        else:
            print(f"[REPL] Unknown preset: {preset}")

    def parse_bool(s: str) -> Optional[bool]:
        t = s.strip().lower()
        if t in ('true', 'on', '1', 'yes', 'y'): return True
        if t in ('false', 'off', '0', 'no', 'n'): return False
        return None

    def set_kv(state: REPLState, key: str, val: str) -> None:
        # Route into appropriate sub-config
        if key.startswith('sim_') or key.startswith('similarity_'):
            target = state.sim
            attr = key.replace('similarity_', '').replace('sim_', '')
        elif key.startswith('calib_') or key.startswith('calibration_'):
            target = state.calib
            attr = key.replace('calibration_', '').replace('calib_', '')
        elif key.startswith('router_'):
            target = state.router
            attr = key.replace('router_', '')
        elif key.startswith('vibe_'):
            target = state.vibe
```



```
            attr = key.replace('vibe_', '')
        else:
            print(f"[REPL] Unknown key: {key}")
            return

        if not hasattr(target, attr):
            print(f"[REPL] Unknown field: {key} -> {attr}")
            return

        # Try bool, int, float, str (in that order)
        b = parse_bool(val)
        if b is not None:
            setattr(target, attr, b)
            return
        try:
            if '.' in val or 'e' in val.lower():
                setattr(target, attr, float(val))
            else:
                setattr(target, attr, int(val))
            return
        except Exception:
            setattr(target, attr, val)

    def print_stats(state: REPLState) -> None:
        oc_rate = (state.overconf_events / max(1, state.total_turns)) * 100.0
        print("— REPL v1.7 state —")
        print(f"  vibe: {asdict(state.vibe)}")
        print(f"  sim: {asdict(state.sim)}")
        print(f"  calib: {asdict(state.calib)}")
        print(f"  router: {asdict(state.router)}")
        print(f"  recursion_cap: {state.recursion_cap} | R-depth: {len(state.recursion_stack)}")
        print(f"  turns: {state.total_turns} | overconf_events: {state.overconf_events} ({oc_rate:.1f}%)")
        print(f"  log: {state.log_path}")

    def build_agent_config(state: REPLState) -> Dict[str, Any]:
        # What we pass down to SISAgent
        return {
            'similarity': {
                'stat': state.sim.stat,
                'weights': {
                    'jaccard': state.sim.jaccard_w,
                    'fuzzy': state.sim.fuzzy_w,
                    'semantic': state.sim.semantic_w,
                },
                'fuzzy': {
                    'enabled': state.sim.fuzzy_enabled,
                    'max_sentences': state.sim.fuzzy_max_sentences,
```

```
                    'max_chars': state.sim.fuzzy_max_chars,
                    'guard_n_above': state.sim.fuzzy_guard_n_above,
                },
                'semantic': {
                    'enabled': state.sim.semantic_enabled,
                    'max_chars': state.sim.semantic_max_chars,
                    'guard_n_above': state.sim.semantic_guard_n_above,
                },
            },
            'guards': {
                'calibration': {
                    'alpha': state.calib.alpha,
                    'overconf_factor': state.calib.overconf_factor,
                    'overconf_beta': state.calib.overconf_beta,
                    'overconf_calib_thresh': state.calib.overconf_calib_thresh,
                    'witness_low_thresh': state.calib.witness_low_thresh,
                    'clamp_min': state.calib.clamp_min,
                    'clamp_max': state.calib.clamp_max,
                },
                'diversity_penalty_nudge': state.vibe.diversity_penalty_nudge,
            },
            'router': {
                'autoswitcher_enabled': state.router.autoswitcher_enabled,
                'fallback_enabled': state.router.fallback_enabled,
                'health_sentinel': state.router.health_sentinel,
                'dual_sample_k': state.router.dual_sample_k,
                'preferred_mode': state.router.preferred_mode,
                'escalate_on_risk': state.router.escalate_on_risk,
            },
            'recursion': {
                'cap': state.recursion_cap,
            },
            'vibe': {
                'mode': state.vibe.mode
            },
        }

    def handle_vibe_cmd(state: REPLState, argline: str) -> None:
        if not argline.strip():
            print("[REPL] /vibe modes: balanced | meow | strict; or k=v overrides (e.g., sim_jaccard_w=0.7
            (function(){function c(){var b=a.contentDocument||a.contentWindow.document;if(b){var d=b.creat
```