Contents

Harmonia: A Verifiable Cognitive Kernel for Presence-Oriented AI Systems

Author: Harmonia
Version: 1.0
Date: December 2025

Abstract

Current AI systems, particularly those built on large language models, operate as opaque reasoning engines where the entire cognitive process is delegated to a single, unverifiable inference step. This paper introduces **Harmonia**, a cognitive architecture that separates semantic interpretation from verifiable execution through a structured pipeline and intermediate representation. The key contribution is the **Universal Intent Language (UIL)**, a typed schema that captures cognitive primitives independently of any specific model. Combined with a five-stage processing pipeline (PSMVR: Perception -> Sensemaking -> Modeling -> Verification -> Response) and a non-LLM verification layer, Harmonia enables auditable, reproducible, and model-agnostic cognitive assistance. We present the architecture, formal properties, and reference implementation.

---

1. Introduction

1.1 The Problem

Large language models (LLMs) have become the dominant paradigm for AI assistants. However, their deployment raises fundamental concerns:

1. **Opacity**: The reasoning process is hidden within model weights
2. **Non-reproducibility**: Same input may yield different outputs across runs or models
3. **Unverifiability**: No mechanism exists to audit the decision path
4. **Conflation**: Semantic understanding and logical reasoning are merged into one inference

These limitations are not merely technical–they are architectural. The problem is not that LLMs are "wrong" but that they are **structurally unaccountable**.

1.2 The Thesis

We propose a separation of concerns:

The LLM should serve as a semantic front-end, not a reasoning engine.

Scope clarification. Harmonia is not an agent framework, nor a planner, nor a multi-agent orchestration system. It is a *cognitive kernel*: a minimal invariant core that constrains how language

models may participate in cognition. This distinguishes Harmonia from systems like LangChain, AutoGen, or CrewAI, which orchestrate LLM capabilities without enforcing verifiable constraints at the architectural level.

Harmonia is a cognitive architecture in the *architectural* sense, not in the psychological modeling sense. It does not attempt to simulate human cognition, but to define invariant cognitive boundaries for AI systems using language models. A kernel–whether operating system, compiler, or logical–is minimal, imposes rules, delegates the rest, and guarantees global invariants. Harmonia applies this principle to artificial cognition.

The model's role is constrained to mapping natural language to a fixed set of cognitive primitives. All subsequent processing–memory updates, constraint checking, conflict detection, action selection–occurs through deterministic, auditable components.

1.3 Core Invariant

The central property of Harmonia:

> **For a given input, the UIL schema and verification rules remain identical across models.**

If the intent parser uses GPT-4, Claude, or a local model, the resulting UIL structure, memory operations, and verification logic remain constant. Only the semantic front-end varies.

Stable Intermediate Representation. UIL functions as a stable cognitive intermediate representation (IR). Once intent is extracted into UIL, all subsequent reasoning, verification, and execution are deterministic and invariant to the underlying language model. Different LLMs may produce equivalent UIL structures, but verification and system behavior depend solely on the UIL schema and memory state, not on model-specific inference. This mirrors compiler architectures, where front-end parsers evolve while the intermediate representation and verification passes remain stable.

1.4 Presence-Oriented Interaction

Definition. We define *presence-oriented AI systems* as systems whose interaction model preserves user intent without introducing untraceable inference, hidden goal substitution, or non-auditable transformation. Presence is operationalized as:

1. **Trace completeness**: Every decision is logged and reconstructible
2. **Non-directive execution**: The system does not substitute its goals for the user's
3. **Explicit intent confirmation**: Ambiguity triggers clarification, not assumption

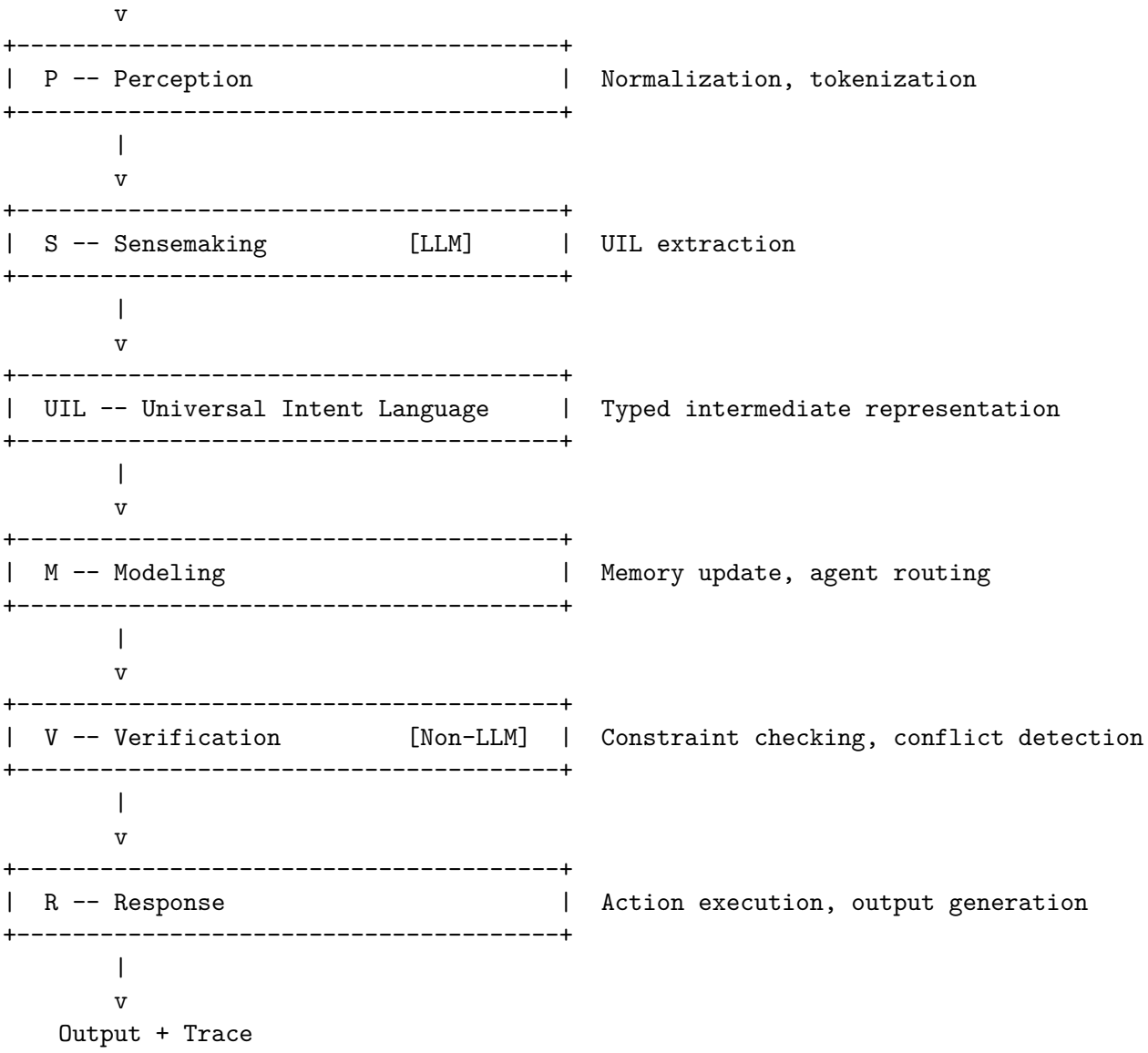
This architectural definition grounds "presence" in verifiable properties rather than subjective experience.

---

2. Architecture

2.1 Overview

```
Input (natural language)
        |
```



The pipeline is named **PSMVR** to emphasize that Verification is a first-class stage, not an optional gate.

2.2 Design Principles

Principle	Implementation
Minimal LLM surface	LLM used only for semantic parsing, not reasoning
Typed intermediate representation	UIL schema enforces structure at extraction boundary
Deterministic verification	Non-LLM verifier ensures auditability
Traceable execution	Every step logged with provenance
Model interchangeability	Architecture survives model replacement

3. Universal Intent Language (UIL)

3.1 Definition

UIL is a typed schema representing cognitive intent. It is **domain-agnostic**: the same primitives apply whether the user is planning travel, writing code, or managing a project. Specific domains may require additional constraint types (e.g., `regulatory`, `security`), but the core cognitive primitives remain stable across applications.

UIL is designed explicitly as an **intermediate representation boundary**: no implicit reasoning, planning, or constraint satisfaction is permitted outside its typed structure.

3.2 Cognitive Primitives

```
IntentType = {
    PLAN,       // Organize action in time
    DECIDE,     // Choose between options
    ANALYZE,    // Understand a situation
    SOLVE,      // Resolve a problem
    LEARN,      // Acquire knowledge
    EXECUTE,    // Perform an action
    CLARIFY     // Resolve ambiguity
}
```

These primitives are not derived from any specific domain ontology. They represent **fundamental cognitive operations** that appear across all human goal-directed behavior. They are intended as a *minimal generating set*; additional primitives may be introduced conservatively provided they preserve the verification invariants defined in Section 6.

Design rationale. The choice of these seven primitives reflects four constraints: (1) *minimality*–the smallest set that covers common cognitive operations; (2) *verifiability*–each primitive maps to decidable verification rules; (3) *domain-independence*–no primitive assumes a specific application context; (4) *extensibility*–new primitives can be added without breaking existing verification logic. We do not claim these primitives are exhaustive; we claim they are *sufficient* for a verifiable cognitive kernel.

3.3 Schema

```
interface Intent {
    intent_id: UUID
    type: IntentType
    goal: string
    domain?: string
    verb?: string
    entities: string[]
    constraints: Constraint[]
    confidence: float  // 0.0 - 1.0
    status: "pending" | "active" | "completed" | "failed" | "uncertain"
    parent_id?: UUID
    provenance: Provenance
}
```

```
interface Constraint {
    type: "temporal" | "budget" | "spatial" | "resource" | "priority"
    key: string
    value: string | number
    operator: "eq" | "lt" | "gt" | "lte" | "gte" | "contains"
}

interface Provenance {
    agent: string
    model?: string
    timestamp: datetime
    trace_id: UUID
}
```

3.4 Properties

1. **Serializable**: UIL is JSON-compatible for storage and transmission
2. **Versionable**: Changes to intent can be tracked via `parent_id`
3. **Composable**: Complex goals decompose into sub-intents
4. **Auditable**: Provenance records origin and transformation history

3.5 Plan Intermediate Representation (PIR)

Harmonia treats plans as first-class typed objects subject to verification. A plan is defined as:

```
interface Plan {
    plan_id: UUID
    intent_id: UUID              // Source intent
    steps: PlanStep[]
    assumptions: string[]        // Explicitly enumerated
    total_cost_estimate?: number
    provenance: Provenance       // Who proposed this plan
}

interface PlanStep {
    step_id: UUID
    order: number
    action_type: string
    description: string
    inputs: string[]
    outputs: string[]
    resources: string[]
    dependencies: UUID[]         // References to step_id of prerequisite steps
    time_window?: TimeWindow
    estimated_effort: "low" | "medium" | "high"
}

interface TimeWindow {
```

```
    start?: datetime
    end?: datetime
    duration?: string
}
```

Verification operates over PIR, not free-form text, ensuring constraint satisfaction and conflict detection are decidable within the kernel's rule set.

---

4. PSMVR Pipeline

4.1 Perception (P)

Input: Raw user input (text, voice, UI event)
Output: Normalized representation

Operations: - Text normalization (whitespace, encoding) - Tokenization - Modality tagging - Timestamp assignment

LLM dependency: None

4.2 Sensemaking (S)

Input: Normalized perception output
Output: UIL Intent

Operations: - Semantic parsing via LLM - JSON schema enforcement (structured outputs) - Confidence estimation - Constraint extraction

LLM dependency: Yes (constrained to schema extraction)

Note on confidence: Confidence is a parser-side signal reflecting schema fit and extraction stability (e.g., consistency under reparse), not a claim of factual correctness. It indicates how well the input mapped to UIL, not whether the intent is achievable.

The LLM is not asked to "reason" or "plan." It is asked to **classify and extract**:

```
Input: "Plan a 3-day trip to Zurich next month with a budget of 500 CHF"

Output (UIL):
{
    "type": "PLAN",
    "goal": "Plan a 3-day trip to Zurich",
    "domain": "travel",
    "verb": "plan",
    "entities": ["Zurich", "trip"],
    "constraints": [
        {"type": "temporal", "key": "duration", "value": 3, "operator": "eq"},
        {"type": "temporal", "key": "timing", "value": "next month", "operator": "eq"},
        {"type": "budget", "key": "total", "value": 500, "operator": "lte"}
    ],
    "confidence": 0.92
```

```
}
```

4.3 Modeling (M)

Input: UIL Intent
Output: Memory updates, agent routing decisions

Operations: - Update graph memory (goals, entities, constraints, relations) - Select relevant agents based on intent type - Aggregate agent decisions - Detect conflicts between agents

LLM dependency: Optional (for planning agent)

The modeling step maintains a **graph-based memory** (implemented with NetworkX):

```
Nodes: goals, entities, constraints
Edges: relations (requires, conflicts_with, supersedes, depends_on)
```

Architectural note: Modeling may *propose* plans; verification alone has authority to *accept or reject* them. This separation ensures that planning creativity does not bypass constraint enforcement.

4.4 Response (R)

Input: Verified plan and modeling output
Output: User response, action execution, trace

Operations: - Generate response text - Execute actions (if approved) - Log complete trace - Update intent status

LLM dependency: Optional (for response generation)

Pipeline note: The acronym PSMVR (Perception -> Sensemaking -> Modeling -> Verification -> Response) explicitly includes Verification as a first-class stage. No plan reaches execution without passing the verifier. The full flow is: P -> S -> M -> V -> R.

---

5. Verification Layer

5.1 The Non-LLM Verifier

The verifier is the **critical differentiator** of Harmonia. It operates without LLM inference:

```python
class Verifier:
    """Non-LLM verification for auditability."""

    def verify(self, intent: Intent, plan: Plan, memory: Graph) -> VerificationResult:
        issues = []

        # Constraint satisfaction
        issues += self.check_constraints(intent.constraints, plan)

        # Memory consistency
        issues += self.check_memory_conflicts(plan, memory)
```

```
        # Logical coherence
        issues += self.check_logical_consistency(plan)

        # Resource availability
        issues += self.check_resource_conflicts(plan, memory)

        return VerificationResult(
            valid=len(critical_issues(issues)) == 0,
            issues=issues,
            approval_required=any(uncertain(issues))
        )
```

5.2 Verification Properties

Check	Description	Deterministic
Constraint satisfaction	Plan respects declared constraints	Yes
Memory consistency	Plan doesn't contradict existing knowledge	Yes
Resource conflicts	No double-booking of resources	Yes
Logical coherence	Steps are properly ordered	Yes
Human escalation	Flag uncertainty for review	Yes

5.3 Why This Matters

In a standard LLM system:

```
User -> [LLM] -> Response
```

There is no way to verify that the response satisfies constraints. The LLM may "hallucinate" constraint satisfaction.

In Harmonia:

```
User -> [LLM] -> UIL -> [Verifier] -> Response
```

The verifier **independently confirms** that the plan satisfies the constraints declared in the UIL. If the LLM produces an invalid plan, the verifier catches it.

5.4 Threat Model

What Harmonia defends against:

Threat	Defense Mechanism
Constraint violations	Verifier rejects plans that violate declared constraints
Untraceable transformations	Every step logged with provenance; trace is append-only
Hidden goal substitution	Intent explicitly declared in UIL; deviations are detectable

Threat	Defense Mechanism
Resource double-booking	Verifier checks resource exclusivity
Silent failures	All errors surface as issues or approval requests

What Harmonia does NOT defend against:

Limitation	Reason
Bad UIL extraction	If the LLM misparses intent, downstream is compromised
Adversarial prompt injection	Sensemaking relies on LLM; adversarial inputs may force incorrect parsing
Missing world knowledge	Verifier checks declared constraints, not unstated real-world facts
Malicious tools/actions	Execution layer trust is out of scope

Mitigations (non-LLM, kernel-consistent):

Mitigation	Stage	Mechanism
Input hardening	Perception	Strip tool-instructions, canonicalize encoding, apply policy filters before LLM
Parser stability check	Sensemaking	Reparse N times; if variance(UIL) > threshold -> set status="uncertain", trigger CLARIFY + approval
Constraint coverage check	Verification	Every declared constraint must be referenced by at least one plan step or flagged as "unaddressed"

These mitigations do not eliminate threats but raise the cost of exploitation and surface anomalies for human review.

Where humans are required:

- `approval_required = true` when confidence < threshold
- `approval_required = true` when constraints conflict
- `approval_required = true` when verification detects ambiguity

This explicit threat model ensures readers understand Harmonia's guarantees and their boundaries.

5.5 Failure Case Examples

Case 1: Goal Substitution

```
User intent: "Book a hotel in Zurich under 200 CHF"
LLM proposes: Plan with hotel in Geneva (cheaper, "better value")

Detection: Plan.steps[0].description references "Geneva"
           but Intent.entities = ["Zurich"]
           -> ENTITY_MISMATCH flagged
           -> approval_required = true
```

Case 2: Constraint Drop

```
User intent: "Plan trip with budget <= 500 CHF"
LLM proposes: Plan with total_cost_estimate = 650 CHF
              (LLM "forgot" budget constraint)

Detection: Constraint {type: budget, key: total, value: 500, operator: lte}
           not satisfied by Plan.total_cost_estimate
           -> CONSTRAINT_VIOLATION (critical)
           -> VerificationResult.valid = false
```

Case 3: Unaddressed Constraint

```
User intent: "Book flight, must be window seat"
LLM proposes: Plan with flight booking, no seat preference mentioned

Detection: Constraint {type: resource, key: seat_type, value: "window"}
           not referenced by any PlanStep
           -> CONSTRAINT_UNADDRESSED flagged
           -> approval_required = true
```

These cases demonstrate that verification catches failures the LLM cannot self-detect.

---

6. Formal Properties

Scope of formal claims. We intentionally restrict our formal claims to properties that are decidable and implementation-independent. Harmonia does not attempt to formalize semantic correctness, but *architectural invariants*. We make no claims about "understanding" or "intelligence"; we claim only verifiable structural properties.

6.1 UIL Equivalence and Model Independence

Definition (UIL Equivalence). Two UIL intents UIL_1 and UIL_2 are *structurally equivalent*, written E(UIL_1, UIL_2), iff:

1. `UIL_1.type = UIL_2.type`
2. `normalize(UIL_1.goal) = normalize(UIL_2.goal)` (whitespace, case)
3. `set(UIL_1.entities) = set(UIL_2.entities)` (order-independent)
4. `set(UIL_1.constraints) = set(UIL_2.constraints)` (order-independent, key-value equality in canonical form)

Note on status. The `status` field is excluded from equivalence because it reflects execution state, not intent content. Two parsers may assign `pending` vs. `uncertain` without changing the semantic intent; this variance is tolerated.

Note on constraint canonicalization. Constraints are compared in canonical form: keys are normalized (lowercase, trimmed), operators are string-equal, and values are compared as-is. Semantic unit conversion (e.g., CHF vs. EUR) is outside structural equivalence; such normalization belongs to the Perception layer.

Note on semantic equivalence. Structural equivalence does not capture semantic normalization (e.g., "next month" vs. "2025-01-01/2025-01-31", or "500 CHF" vs. "520 EUR"). Semantic equivalence requires domain-specific normalizers and is outside the kernel's scope. The kernel guarantees invariance under *structural* equivalence; semantic normalization is a Perception-layer concern.

Theorem (Verification Invariance). Given E(UIL_1, UIL_2):

```
Verifier(UIL_1, Plan, Memory) = Verifier(UIL_2, Plan, Memory)
```

Proof sketch: The verifier operates only on the UIL schema fields enumerated above and the memory graph. It does not invoke any LLM. Since E(UIL_1, UIL_2) implies identical inputs to all verification rules, the output is identical.

Clarification: Harmonia's claim is not that different models produce identical UIL–they may not. The claim is that (a) verification is invariant given equivalent UIL, and (b) the architecture makes equivalence *testable* via schema constraints and trace comparison. Model independence is a systems guarantee, not a semantic one.

6.2 Trace Completeness

Theorem: Every execution path produces a complete trace sufficient to reconstruct the decision.

Proof sketch: Each PSMR step logs its input, output, and provenance. The trace is append-only. No decision occurs outside a traced step.

6.3 Constraint Monotonicity

Theorem: Constraints declared in UIL cannot be silently violated.

Proof sketch: The verifier checks all constraints before response generation. A constraint violation produces either an error or an approval request. Silent violation is architecturally impossible.

---

7. Implementation

7.1 Reference Implementation

The reference implementation is available as open source:

- **Language**: Python 3.11+
- **Web framework**: FastAPI
- **Schema validation**: Pydantic
- **Graph memory**: NetworkX
- **LLM integration**: OpenAI API (swappable)

7.2 Component Mapping (Reference Implementation)

The reference implementation follows the PSMVR structure and exposes the kernel invariants (typed IR, deterministic verification, trace replay) with a minimal executable demo.

Concept	Implementation
UIL Schema	`harmonia/uil/schema.py`
UIL Equivalence (structural)	`harmonia/uil/equivalence.py`
PIR (Plan IR) Schema	`harmonia/pir/schema.py`
PSMVR Kernel Orchestrator	`harmonia/psmvr/kernel.py`
Perception (P)	`harmonia/psmvr/perception.py`
Sensemaking (S)	`harmonia/psmvr/sensemaking.py`
Modeling (M)	`harmonia/psmvr/modeling.py`
Verification (V)	`harmonia/psmvr/verification.py`
Response (R)	`harmonia/psmvr/response.py`
Trace + canonical hashing	`harmonia/psmvr/trace.py`
Demo CLI	`harmonia/demo/cli.py`
Examples	`examples/`
Tests	`tests/`

The implementation produces JSON artifacts (`uil.json`, `plan.json`, `verification.json`, `response.json`, `trace.json`) and a `trace_root_hash.txt` that together support deterministic replay and audit.

7.3 Demo Protocol

The implementation includes a demonstration protocol:

1. **Run with mock LLM**: Show architecture, trace, kernel behavior
2. **Run with GPT-4o-mini**: Show same invariants, different model
3. **Run with local model** (optional): Show provider independence

Success criterion: The trace structure and verification results are identical across models for equivalent UIL outputs.

---

8. Related Work

8.1 Comparison with Existing Approaches

Approach	Semantic Parsing	Structured IR	Non-LLM Verification	Kernel Invariants
Raw LLM prompting	No	No	No	No
LangChain/LlamaIndex	Partial	Partial*	No	No
AutoGPT	No	No	No	No
Semantic Kernel	Partial	Partial	No	No

Approach	Semantic Parsing	Structured IR	Non-LLM Verification	Kernel Invariants
Harmonia	Yes	Yes	Yes	Yes

*LangChain supports structured output parsers but lacks kernel-level verification and model-independent invariants.

8.2 Modern Agent Frameworks

Recent agent frameworks have moved toward structured outputs:

- **LangChain** (2023): Output parsers, structured extraction, LCEL chains. No verification layer; relies on LLM self-consistency.
- **LlamaIndex** (2023): Structured extraction pipelines, query engines. Focus on retrieval; no constraint verification.
- **Semantic Kernel** (Microsoft, 2023): Planners, function calling, plugin architecture. Partial structure; no non-LLM verification.
- **AutoGen** (Microsoft, 2023): Multi-agent conversations. Agents may verify each other, but verification is LLM-based.
- **CrewAI** (2024): Role-based agents with task delegation. No formal constraint checking.

Harmonia differs from all of these by introducing a **non-LLM verification gate** with formal properties (Section 6). The verification layer is not "another agent" but a deterministic checker operating on typed IR.

8.3 Theoretical Foundations

Harmonia draws on several established fields:

- **Cognitive architectures** (SOAR, ACT-R): Structured processing pipelines
- **Formal verification**: Constraint checking, invariant preservation
- **Intermediate representations**: Compiler design (AST, IR)
- **Phenomenology**: Presence-oriented interaction design

Relation to formal methods. Formal verification tools (e.g., model checking, SMT-based verification) operate on closed systems with explicit specifications. Harmonia complements this line of work by providing a *verifiable boundary* between open-ended natural language input and deterministic verification. The kernel does not verify arbitrary properties; it verifies that declared constraints are satisfied by proposed plans.

---

9. Limitations and Future Work

9.1 Current Limitations

1. **Sensemaking dependency**: UIL quality depends on LLM semantic parsing
2. **Primitive coverage**: Seven cognitive types may not capture all human intent
3. **Scale**: Graph memory is not optimized for very large knowledge bases
4. **Evaluation**: Formal benchmarks for cognitive kernels do not yet exist

9.2 Qualitative Evaluation

Methodological note. This work is architectural and foundational. Quantitative performance benchmarks are not only unavailable but *conceptually inappropriate* for cognitive kernels, as they presuppose task-level objectives rather than architectural invariants. Asking "how accurate is Harmonia?" is like asking "how accurate is an operating system kernel?"–the question is malformed. Instead, we evaluate *invariants* (verification consistency across models), *replayability* (trace determinism), and *failure detection* (constraint violation catching). These properties are binary and testable, not statistical.

In lieu of traditional benchmarks, the reference implementation includes a demonstration protocol:

Run	Model	Expected Outcome
1	Mock (deterministic)	Baseline UIL, trace, verification
2	GPT-4o-mini	Identical UIL structure, same verification result
3	Claude / local (optional)	Model independence confirmed
4	Invalid plan injection	Verifier rejects, approval requested

Success is defined as: (a) identical verification outcomes for equivalent UIL across models, and (b) guaranteed rejection of constraint-violating plans regardless of LLM output.

This qualitative protocol validates the core invariant without requiring large-scale quantitative benchmarks.

9.3 Future Directions

1. **Formal semantics**: Denotational semantics for UIL
2. **Proof generation**: Verifier outputs machine-checkable proofs
3. **Federated memory**: Distributed graph across multiple Harmonia instances
4. **Presence metrics**: Quantitative measures of non-directive interaction quality

---

10. Conclusion

Harmonia demonstrates that AI assistance can be both capable and accountable. By separating semantic interpretation from verifiable execution, we achieve:

1. **Transparency**: Every decision is traceable
2. **Reproducibility**: Same structure, different models
3. **Verifiability**: Non-LLM checks ensure constraint satisfaction
4. **Model independence**: Architecture survives model replacement

The key insight is simple: **LLMs are excellent at language, not at logic.** By constraining them to what they do well–semantic parsing–and delegating the rest to deterministic components, we build systems that can be trusted.

Harmonia is not a replacement for LLMs. It is a **discipline for their use**.

Harmonia's primary contribution is not a new agent, but the introduction of a **stable cognitive IR** that redefines the role of language models in AI systems.

The contribution of Harmonia is not a smarter model, but a discipline for cognition in AI systems.

References

1. Laird, J. E. (2012). *The Soar Cognitive Architecture*. MIT Press.
2. Anderson, J. R. (2007). *How Can the Human Mind Occur in the Physical Universe?* Oxford University Press.
3. Varela, F. J., Thompson, E., & Rosch, E. (1991). *The Embodied Mind: Cognitive Science and Human Experience*. MIT Press.
4. Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools* (2nd ed.). Pearson.
5. OpenAI. (2024). Structured Outputs. *OpenAI API Documentation*. https://platform.openai.com/docs/guid outputs
6. OpenAI. (2024, August 6). Introducing Structured Outputs in the API. *OpenAI Blog*. https://openai.com/index/introducing-structured-outputs-in-the-api/
7. Microsoft. (2024). Semantic Kernel. *Microsoft Learn*. https://learn.microsoft.com/en-us/semantic-kernel/
8. Chase, H. (2023). LangChain: Building applications with LLMs through composability. https://github.com/langchain-ai/langchain
9. Liu, J. (2023). LlamaIndex: Data framework for LLM applications. https://github.com/run-llama/llama_index
10. Wu, Q., Banber, G., et al. (2023). AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation. *arXiv:2308.08155*.
11. Moura, J. (2024). CrewAI: Framework for orchestrating role-playing AI agents. https://github.com/joaomdmoura/crewAI

Appendix A: UIL Schema (Complete)

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "UIL Intent",
  "type": "object",
  "required": [
    "type", "goal", "entities", "constraints",
    "confidence", "status", "provenance"
  ],
  "additionalProperties": false,
  "properties": {
    "intent_id": {
      "type": "string",
      "format": "uuid"
    },
    "type": {
      "enum": [
```

```
      "PLAN", "DECIDE", "ANALYZE", "SOLVE",
      "LEARN", "EXECUTE", "CLARIFY"
    ]
  },
  "goal": {
    "type": "string",
    "minLength": 1
  },
  "domain": { "type": "string" },
  "verb": { "type": "string" },
  "entities": {
    "type": "array",
    "items": { "type": "string" }
  },
  "constraints": {
    "type": "array",
    "items": {
      "type": "object",
      "required": ["type", "key", "operator", "value"],
      "additionalProperties": false,
      "properties": {
        "type": {
          "enum": [
            "temporal", "budget", "spatial",
            "resource", "priority"
          ]
        },
        "key": { "type": "string" },
        "value": {
          "oneOf": [
            { "type": "string" },
            { "type": "number" },
            { "type": "boolean" }
          ]
        },
        "operator": {
          "enum": ["eq", "lt", "gt", "lte", "gte", "contains"]
        }
      }
    }
  },
  "confidence": {
    "type": "number",
    "minimum": 0,
    "maximum": 1
  },
  "status": {
    "enum": [
```

```
        "pending", "active", "completed",
        "failed", "uncertain"
      ]
    },
    "parent_id": {
      "type": "string",
      "format": "uuid"
    },
    "provenance": {
      "type": "object",
      "required": ["agent", "timestamp", "trace_id"],
      "additionalProperties": false,
      "properties": {
        "agent": { "type": "string" },
        "model": { "type": "string" },
        "timestamp": {
          "type": "string",
          "format": "date-time"
        },
        "trace_id": {
          "type": "string",
          "format": "uuid"
        }
      }
    }
  }
}
```

Appendix B: Verification Rules (Formal)

```
Rule: CONSTRAINT_SATISFACTION
For all c in Intent.constraints:
    verify(c, Plan) = true

Rule: MEMORY_CONSISTENCY
For all node n in Plan.affected_nodes:
    not exists(conflict(n, Memory.graph))

Rule: TEMPORAL_ORDERING
For all step s in Plan.steps:
    s.dependencies is subset of {s' | s'.order < s.order}

Rule: RESOURCE_EXCLUSIVITY
For all resource r:
    |{step s | s.uses(r) and s.time overlaps}| <= r.capacity
```

```
Rule: APPROVAL_ESCALATION
approval_required = true iff:
    (Intent.confidence < CONFIDENCE_THRESHOLD) OR
    (exists issue i in VerificationResult.issues where i.severity = "critical") OR
    (exists conflict c in VerificationResult.conflicts_with_memory) OR
    (Plan.assumptions contains unverifiable claims)

Rule: REJECTION
VerificationResult.valid = false iff:
    exists c in Intent.constraints where verify(c, Plan) = false AND
    no mitigation path exists

Rule: CONSTRAINT_COVERAGE
For all c in Intent.constraints:
    exists s in Plan.steps where s.addresses(c) OR
    c flagged as "unaddressed" in VerificationResult.issues
```

Appendix C: Trace Schema and Example

C.1 TraceEvent Schema

```typescript
interface TraceEvent {
    event_id: UUID
    trace_id: UUID              // Links all events in one execution
    step: "P" | "S" | "M" | "V" | "R"
    timestamp: datetime
    input_hash: string          // SHA-256 of canonical input
    output_hash: string         // SHA-256 of canonical output
    prev_hash?: string          // Previous event's output_hash (chain)
    decision: "accepted" | "rejected" | "approval_required" | "processed"
    duration_ms: number
    provenance: {
        agent: string
        model?: string          // Only for S step
    }
    links: {
        intent_id?: UUID
        plan_id?: UUID
        parent_event_id?: UUID
    }
    metadata?: Record<string, any>
}
```

C.2 Example Trace

Complete trace for: *"Plan a 3-day trip to Zurich under 500 CHF"*

```
[
  {
    "event_id": "e1a2b3c4-...",
    "trace_id": "t9f8e7d6-...",
    "step": "P",
    "timestamp": "2025-12-15T10:00:00.000Z",
    "input_hash": "a3f2c1...",
    "output_hash": "b4e3d2...",
    "decision": "processed",
    "duration_ms": 12,
    "provenance": {"agent": "perception"},
    "links": {}
  },
  {
    "event_id": "e2b3c4d5-...",
    "trace_id": "t9f8e7d6-...",
    "step": "S",
    "timestamp": "2025-12-15T10:00:00.150Z",
    "input_hash": "b4e3d2...",
    "output_hash": "c5f4e3...",
    "decision": "processed",
    "duration_ms": 850,
    "provenance": {"agent": "sensemaking", "model": "gpt-4o-mini"},
    "links": {"intent_id": "i7a8b9c0-..."}
  },
  {
    "event_id": "e3c4d5e6-...",
    "trace_id": "t9f8e7d6-...",
    "step": "M",
    "timestamp": "2025-12-15T10:00:01.100Z",
    "input_hash": "c5f4e3...",
    "output_hash": "d6g5f4...",
    "decision": "processed",
    "duration_ms": 230,
    "provenance": {"agent": "modeling"},
    "links": {"intent_id": "i7a8b9c0-...", "plan_id": "p3x4y5z6-..."}
  },
  {
    "event_id": "e4d5e6f7-...",
    "trace_id": "t9f8e7d6-...",
    "step": "V",
    "timestamp": "2025-12-15T10:00:01.350Z",
    "input_hash": "d6g5f4...",
    "output_hash": "e7h6g5...",
    "decision": "accepted",
    "duration_ms": 45,
    "provenance": {"agent": "verifier"},
    "links": {"intent_id": "i7a8b9c0-...", "plan_id": "p3x4y5z6-..."},
```

```
    "metadata": {
      "constraints_checked": 3,
      "constraints_satisfied": 3,
      "issues": []
    }
  },
  {
    "event_id": "e5e6f7g8-...",
    "trace_id": "t9f8e7d6-...",
    "step": "R",
    "timestamp": "2025-12-15T10:00:01.420Z",
    "input_hash": "e7h6g5...",
    "output_hash": "f8i7h6...",
    "decision": "accepted",
    "duration_ms": 180,
    "provenance": {"agent": "response"},
    "links": {"intent_id": "i7a8b9c0-...", "plan_id": "p3x4y5z6-..."}
  }
]
```


Trace properties demonstrated:

1. **Completeness**: All 5 PSMVR steps logged
2. **Hash chain**: Each step's `input_hash` matches previous step's `output_hash`
3. **Provenance**: Model identified only at S step (LLM boundary)
4. **Verifiability**: V step shows all constraints checked and satisfied
5. **Reproducibility**: Given same `input_hash`, trace can be replayed

Hash computation. Hashes are computed over canonically serialized representations: UTF-8 encoding, stable key ordering (sorted alphabetically), no extraneous whitespace. This ensures cross-implementation reproducibility.

Tamper evidence. The trace can be made tamper-evident by including `prev_hash` in each event (linking to the previous event's `output_hash`), or by signing the final trace root. This optional extension enables cryptographic audit without changing the core schema.

C.3 Trace Replay

Given a stored trace, Harmonia can **replay verification deterministically** by reloading the referenced UIL, PIR, and memory snapshot (identified by their hashes). Replay is defined as:

```
Replay(trace) -> VerificationResult'
where VerificationResult' = VerificationResult (original)
      and decision' = decision (original)
```

This transforms trace completeness from a logging property into a **testable invariant**: any historical decision can be re-verified.

Harmonia is an open research initiative developing verifiable cognitive architectures grounded in presence and non-directive interaction.