Jung Sungwoo
Posted on 2025년 12월 30일

World-Centric Agent Architecture: Why Your AI Agent Keeps Failing (And It's Not the Model's Fault)

#agents #architecture #llm #ai

World-Centric Agent Architecture

Deterministic Runtime and Hierarchical Intelligence Orchestration

---

TL;DR

Your AI agent's failures aren't because the model isn't smart enough. They're because **the World was never designed**.

This post introduces **World-Centric Architecture**—a paradigm where:

- The World is explicit, immutable, and verifiable

- Intelligence proposes changes but never executes them directly
- System stability is independent of model size
- Every value can answer "Why?"

> If you're expecting another agent framework tutorial, this is not it. This post is about why agent systems fail structurally—even with strong models.

The Problem Nobody's Talking About

LLM-based agents have made remarkable progress. Yet they keep failing in predictable ways:

Failure Mode	What Happens
Impossible Action Loops	Agent repeatedly tries actions that can't work
State Opacity	No one knows when or why state changed
Irreproducibility	Same input, different results
Unrecoverable Failures	Can't rollback, can't try alternatives
Unexplainability	"Why did it do that?" → 🤷

Sound familiar?

The Misdiagnosis

The industry's response has been consistent:

> "The model isn't smart enough. Let's use a bigger one."

```
Problem → "Model too dumb"
       → Bigger model / Longer reasoning / Complex planning
       → Higher cost / More complexity / Problem persists
```

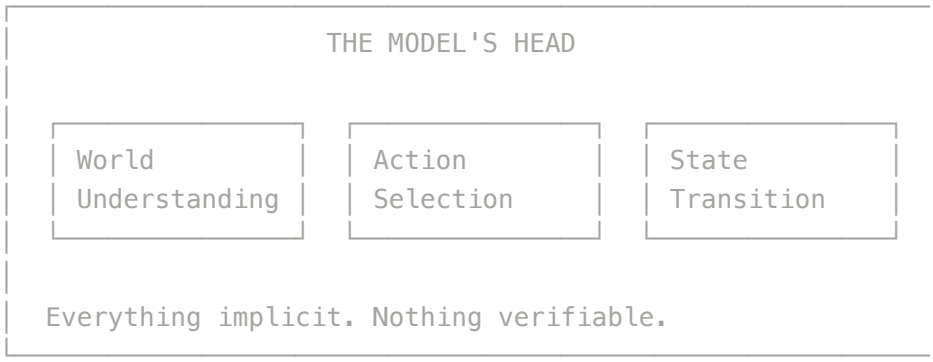
We've been treating symptoms, not the disease.

The Real Problem

Here's the uncomfortable truth:

> **These failures aren't intelligence failures. They're world modeling failures.**

In most agent architectures, the "world" exists only inside the model's head:



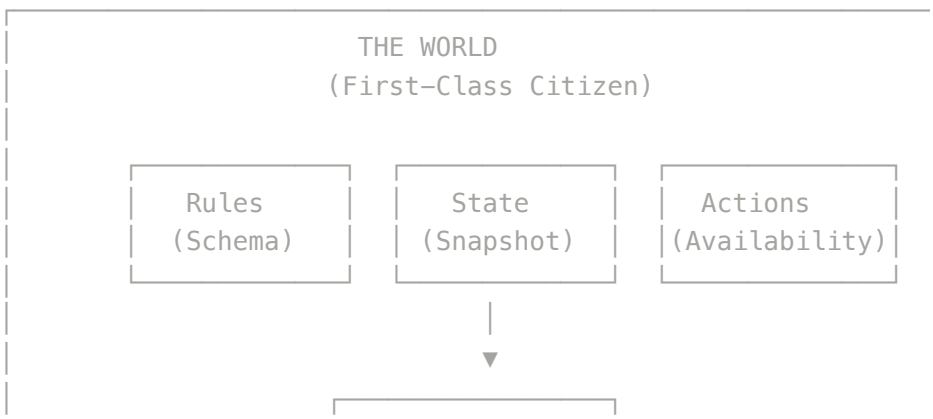
The model must:

1. Understand the world (from context)
2. Know what's possible (by reasoning)
3. Pick the right action (while hallucinating)
4. Track state changes (in its attention weights)
5. Explain failures (good luck)

That's too much responsibility for a non-deterministic text predictor.

The Inversion: World-Centric Design

What if we flipped the script?



```
                    │   Intelligence   │
                    │   (Proposer)     │
                    └──────────────────┘


  Intelligence is OUTSIDE. World is explicit.
```

Key shifts:

- World exists **outside** the model
- State is **explicit and immutable**
- Actions have **verifiable availability**
- Intelligence **proposes**, doesn't execute
- Everything can **explain itself**

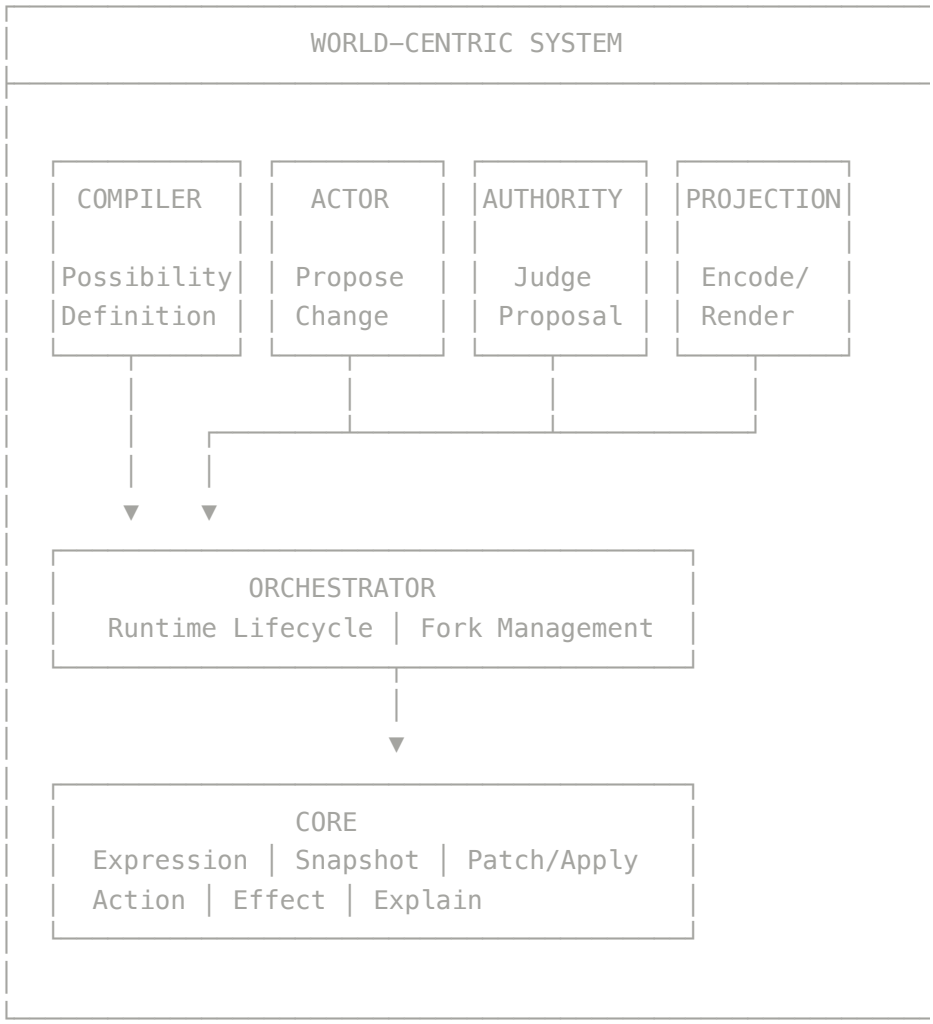
The Constitution

World-Centric Architecture follows six principles:

```
  THE CONSTITUTION


  1. COMPILER defines possibility.
     What CAN happen is defined at design time.

  2. CORE computes truth.
     What IS true is computed deterministically.

  3. ACTOR proposes change.
     Intelligence suggests, never executes.

  4. AUTHORITY judges proposals.
     Independent verification before any change.

  5. ORCHESTRATOR manages worlds.
     Multiple world branches for exploration.

  6. PROJECTION transforms I/O.
     Presentation is separate from meaning.
```

The Six Pillars

Here's how these principles become architecture:



Pillar	Role	Owns State?	Makes Decisions?
Core	Truth Engine	✅ Snapshot	❌
Actor	Change Proposer	❌	❌
Authority	Proposal Judge	❌	✅
Projection	I/O Transformer	❌	❌
Orchestrator	World Manager	✅ Runtimes	❌
Compiler	Possibility Definer	✅ Schema	❌

Core: The Deterministic Truth Engine

Core is the foundation—a seven-layer engine that computes truth without side effects:

```
┌─────────────────────────────────────────────────────────┐
│  Layer 7: EXPLAIN                                       │
│  "Why this value?" — Structural interpretation          │
├─────────────────────────────────────────────────────────┤
│  Layer 6: PATCH / APPLY                                 │
│  The ONLY way to change truth                           │
├─────────────────────────────────────────────────────────┤
│  Layer 5: EFFECT RUNTIME                                │
│  External execution via handlers                        │
├─────────────────────────────────────────────────────────┤
│  Layer 4: ACTION                                        │
│  Availability gate + Effect declaration                 │
├─────────────────────────────────────────────────────────┤
│  Layer 3: SNAPSHOT                                      │
│  Immutable state at a point in time                     │
├─────────────────────────────────────────────────────────┤
│  Layer 2: DAG                                           │
│  Dependency graph for incremental recompute             │
├─────────────────────────────────────────────────────────┤
│  Layer 1: EXPRESSION                                    │
│  Pure computation (no side-effects)                     │
└─────────────────────────────────────────────────────────┘
```

Snapshots Are Immutable

```
type Snapshot<T> = {
  readonly data: T;
  readonly computed: Record<string, unknown>;
  readonly validity: ConstraintResult;
  readonly version: number;
};

// WRONG — Direct mutation
snapshot.data.user.email = 'new@example.com';

// RIGHT — Patch/Apply produces NEW snapshot
const newSnapshot = applyPatches(snapshot, [
  set('user.email', 'new@example.com')
]);
```

Actions Gate on Availability

```
const SubmitAction = defineAction(AppState, () => ({
  // Availability is a COMPUTED FACT
  availability: get('user.computed.isValid'),

  // Effect only runs IF available
  effect: effect('api.submit', {
    email: get('user.email'),
  }),
}));
```

Impossible actions never reach execution. The system prevents them structurally, not through model reasoning.

Everything Explains Itself

```
const result = explainAvailability(SubmitAction, snapshot);

// Returns structured explanation:
{
  available: false,
  tree: {
    operator: 'and',
    contribution: 'false',
    children: [
      { path: 'user.computed.emailOk', value: false },  // ←
      { path: 'user.computed.passwordOk', value: true },
    ]
  }
}
```

No guessing. No "maybe the email was empty?" Just structural facts.

Hierarchical Intelligence: Student / Teacher / Orchestrator

Intelligence isn't monolithic. It's layered:

```
┌─────────────────────────────────────────────────────────────┐
│  LEVEL 3: ORCHESTRATOR                                      │
│  ─────────────────────                                      │
│  • Manages multiple world branches (Fork tree)              │
│  • Preserves failed worlds (no deletion)                    │
│  • Selects successful path                                  │
│  • NO intelligence required (deterministic)                 │
└─────────────────────────────────────────────────────────────┘
```

```
│  LEVEL 2: TEACHER (Optional)
│  ──────────────────────────
│  • Observes failures and execution traces
│  • Proposes world hypotheses
│  • Does NOT execute, has NO authority
│  • Can be wrong (not an oracle)
├──────────────────────────────────────────────────────────
│  LEVEL 1: STUDENT
│  ────────────────
│  • Operates WITHIN current world
│  • Selects from AVAILABLE actions only
│  • Doesn't need to be smart (random works!)
│  • Proposes, never applies
└──────────────────────────────────────────────────────────
```

The Minimal Student

Here's the key insight: **Student can be incredibly simple**.

```
// This is a valid Student implementation
class RandomStudent {
  propose(view: SnapshotView): Proposal {
    const available = view.availableActions;
    const picked = available[Math.floor(Math.random() * avai
    return createProposal(picked);
  }
}
```

Why does this work? Because:

- Impossible actions are already filtered out
- State transitions are verified by Core
- Failures are handled by Orchestrator

A random selector produces **valid execution traces**. Model intelligence is optional.

Teacher is NOT an Oracle

```
ORACLE (Not our Teacher):
• Has access to ground truth
• Guarantees correct answers
• Unfalsifiable

TEACHER (Our design):
• Observes only execution traces
• Proposes hypotheses that MAY BE WRONG
```

```
• All proposals verified through execution
• Falsifiable and auditable
```

Teacher = "A modeler who can be wrong"

Trust Boundaries

Not everything is trusted equally:

```
┌─────────────────────────────────────────────────────────┐
│  TRUSTED ZONE                                           │
│  ─────────────                                          │
│  Core │ Orchestrator │ Authority │ Projection           │
│                                                         │
│  • Deterministic                                        │
│  • Auditable                                            │
│  • Structurally verified                                │
└─────────────────────────────────────────────────────────┘
                          ▲
                          │ Verification Boundary
                          ▼
┌─────────────────────────────────────────────────────────┐
│  UNTRUSTED ZONE                                         │
│  ──────────────                                         │
│  LLM Actor │ External I/O                               │
│                                                         │
│  • Non-deterministic                                    │
│  • May hallucinate                                      │
│  • All outputs verified before trust                    │
└─────────────────────────────────────────────────────────┘
```

The Firewall Principle:

```
LLM → Proposal → Authority → Approval → Core.apply()

NOT:
LLM → State  // FORBIDDEN
```

LLM outputs **never** directly mutate state. Ever.

The Runtime Flow

Here's how execution actually works:

```
1. PROJECTION renders Snapshot → View
   (Full state filtered to what Actor needs)

2. ACTOR (Student) observes View
   (Sees only available actions)

3. ACTOR proposes ChangeSet
   (Selects action, does NOT execute)

4. Wrap ChangeSet → Proposal
   (Attach identity for accountability)

5. AUTHORITY decides
   → approved / rejected / changes_requested

6. If approved, ORCHESTRATOR forwards to Core

7. CORE.executeAction()
   ├── Check availability (again!)
   ├── Resolve effect parameters
   ├── Execute handler → Patch[]
   ├── Apply patches → New Snapshot
   └── Recompute affected values

8. Loop
```

Failure & Fork

When things go wrong:

```
1. Core returns { ok: false, reason: 'UNAVAILABLE' }

2. Orchestrator captures failure
   ├── Preserve current Snapshot (no deletion!)
   └── Generate ExplainGraph

3. Teacher (if present) analyzes
   ├── Input: Trace + ExplainGraph
   └── Output: World Hypothesis

4. Authority decides on Fork
   → approve / reject

5. Orchestrator creates Fork
   ├── Parent Runtime preserved
   └── Child Runtime with hypothesis applied

6. Execution continues in Child
```

```
7. On success, Authority selects winner
```

Failed worlds are **preserved**, not deleted. You can always go back.

Why This Works

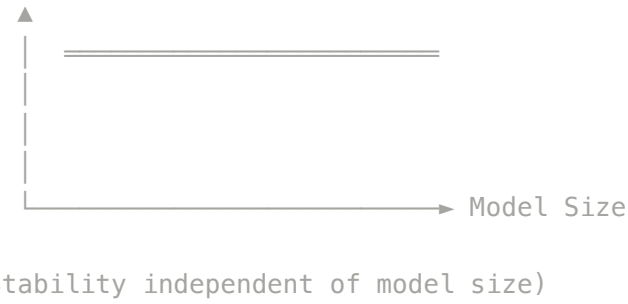
Intelligence Redistribution

```
BEFORE (Intelligence-Centric):
──────────────────────────────
Intelligence = World Understanding
             + Rule Enforcement
             + Action Selection
             + State Management
             + Failure Diagnosis

AFTER (World-Centric):
──────────────────────
Intelligence = Action Selection (minimal)

World Understanding  → Projection
Rule Enforcement     → Core (availability)
State Management     → Core (Patch/Apply)
Failure Diagnosis    → Core (Explain)
```

We moved 80% of the work **out of the model**.

Model Size Independence

```
Intelligence-Centric:

Stability
    ▲
    │
    │
    │
    │
    └──────────────────────► Model Size

(Bigger model = More stable)

──────────────────────────────────────

World-Centric:

Stability
```



A random selector and a **smart-enough model** (e.g., GPT-5 or Claude Opus) both produce **valid** execution traces—because validity is enforced by the World (availability gates + Patch/Apply), not by the model's reasoning.

A stronger model tends to be more **efficient**, but **correctness does not depend on model capability**.

When NOT to Use This

World-Centric isn't for everything:

Scenario	Why It Might Not Fit
Pure creative writing	No "correct" state
Open-ended chat	No action structure
Real-time streaming	Snapshot overhead
Trivial single-turn QA	Overkill

This architecture is designed for **structured, multi-step, verifiable tasks**.

Key Takeaways

1. **Agent failures are often world modeling failures**, not intelligence failures
2. **Make the World explicit**: Immutable snapshots, declarative patches, computed availability
3. **Intelligence proposes, never executes**: The firewall principle

4. **Stability is achievable without scaling**: A random selector can produce valid execution
5. **Everything should explain itself**: If it can't answer "Why?", it's a black box
6. **Preserve failures**: Fork trees enable recovery and exploration

---

The Philosophy

> "A system that cannot explain 'Why' is a dead system."

Most agent architectures are black boxes. You pump in tokens, hope for the best, and debug through prayer.

World-Centric Architecture is a **white box**. Every value has provenance. Every action has availability. Every failure has explanation.

That's not just good engineering. That's the difference between a demo and a production system.

See It In Action

Theory is nice. Working software is better.

TaskFlow is a *playable proof-of-concept* built **before** the full 7-layer architecture—designed to demonstrate a simpler but critical idea:

> **LLMs can be reduced to intent interpreters and semantic carriers,**
>
> **while the system itself remains fully deterministic.**

TaskFlow shows that even *without* the complete World-Centric runtime,
a deterministic machine driven by explicit state and rules can already work.

👉 Try the demo: https://taskflow.manifesto-ai.dev

What's Next?

This architecture is implemented in **Manifesto**, an open-source framework. Future posts will cover:

- Deep dive into Core's 7 layers
- Building custom Actors with LLMs
- Authority patterns for different governance needs
- Fork strategies for exploration

Questions? Disagreements? Let me know in the comments.

Thanks for reading. If you're tired of debugging agent failures at 3 AM, maybe it's time to stop blaming the model and start designing the world.

Top comments (0)

Jung Sungwoo

UX/UI Software Engineer

LOCATION
Korea, Seoul

JOINED
2024년 9월 4일

More from Jung Sungwoo

The Mind Protocol: Why Your AI Agent Needs a World Before It Can Think
#ai #architecture #typescript #agents

Buttons Are Not Functions
#ai #frontend #architecture #agents

Why Your AI Agent Says "Done" But Nothing Actually Happened
#agents #ai #automation #llm