converge-core

A FORMALLY-GROUNDED MULTI-AGENT RUNTIME

The Problem with Agent Systems

Current multi-agent frameworks suffer from:

- **Non-determinism** — Same input, different outputs
- **No convergence guarantee** — Workflows can loop forever
- **Agent drift** — LLMs deviate from intended behavior
- **Untraceable decisions** — "Why did it do that?" has no answer
- **Partial failures** — Inconsistent state after errors

We can do better.

The Converge Approach

A runtime built on **fixed-point semantics**:

```
Root Intent
   ↓
Fan-out (agents propose)
   ↓
Validate + enforce invariants
   ↓
Serial commit (append-only Context)
   ↓
Repeat until fixed point
   ↓
Explainable result + audit trail
```

Agents propose. The engine decides.

The 9 Axioms

MATHEMATICAL FOUNDATIONS

Axioms 1-5

#	Axiom	Formula	Guarantee
1	**Monotonicity**	$\mathtt{ctx} \subseteq \mathtt{step(ctx)}$	Facts never lost
2	**Determinism**	$\mathtt{step(ctx)} = \mathtt{step(ctx)}$	Reproducible
3	**Idempotency**	$\mathtt{agent(ctx)} = \mathtt{agent(agent(ctx))}$	Safe retries
4	**Commutativity**	$\mathtt{a(b(ctx))} = \mathtt{b(a(ctx))}$	Order-independent
5	**Termination**	$\exists \mathtt{n}: \mathtt{step}^{\mathtt{n}}\mathtt{(ctx)} = \mathtt{step}^{\mathtt{n+1}}\mathtt{(ctx)}$	Always halts

Axioms 6-9

#	Axiom	Formula	Guarantee
6	**Consistency**	`¬∃(f, ¬f) ∈ ctx`	No contradictions
7	**Starvation Freedom**	`enabled(a) ⇒ ◇runs(a)`	Fair scheduling
8	**Confluence**	`ctx₁ ∪ ctx₂ → ctx*`	Merges converge
9	**Observability**	`∀effect: logged(effect)`	Full audit trail

These aren't guidelines — they're enforced by the type system.

Core Architecture

TYPE-SAFE BY DESIGN

The Context: Append-Only Truth

```rust
pub struct Context {
    facts: FactStore,        // Immutable, append-only
    clock: LamportClock,     // Logical ordering
    trace: ExecutionTrace,   // Full history
}

impl Context {
    pub fn derive(&self, new_facts: Vec<Fact>) -> Context {
        // Old context unchanged, new context returned
        Context {
            facts: self.facts.extend(new_facts),
            clock: self.clock.tick(),
            trace: self.trace.append(new_facts),
        }
    }
}
```

Immutability guarantees monotonicity.

Facts: Typed, Versioned, Traceable

```rust
pub struct Fact {
    pub id: FactId,
    pub kind: FactKind,
    pub payload: Value,
    pub provenance: Provenance,  // Who created this?
    pub timestamp: LamportTime,
    pub supersedes: Option<FactId>,
}

pub enum FactKind {
    Seed,            // Human-provided input
    Derived,         // Agent-computed
    ProposedFact,    // LLM suggestion (untrusted)
    Validated,       // Promoted from ProposedFact
}
```

Agents: Pure Functions Over Context

```rust
pub trait Agent: Send + Sync {
    fn name(&self) -> &str;

    fn can_run(&self, ctx: &Context) -> bool;

    fn run(&self, ctx: &Context) -> AgentResult;
}

pub enum AgentResult {
    NoOp,                           // Nothing to do
    Propose(Vec<Fact>),             // New facts to add
    Error(AgentError),              // Recoverable failure
}
```

Agents are pure: `Context → AgentResult`

The Engine: Fixed-Point Execution

```rust
impl Engine {
    pub fn run(&mut self, initial: Context) -> EngineResult {
        let mut ctx = initial;

        loop {
            let proposals = self.fan_out(&ctx);      // Parallel agent execution
            let validated = self.validate(proposals); // Invariant checking
            let next = ctx.derive(validated);         // Immutable update

            if next == ctx {                          // Fixed point reached
                return EngineResult::Converged(next);
            }

            ctx = next;
            self.check_termination()?;               // Cycle limit
        }
    }
}
```

LLM Integration: Trust Boundaries

```rust
pub struct LlmAgent {
    provider: Arc<dyn LlmProvider>,
    validator: Arc<dyn FactValidator>,
}

impl Agent for LlmAgent {
    fn run(&self, ctx: &Context) -> AgentResult {
        let response = self.provider.complete(ctx)?;

        // LLM output is NEVER directly trusted
        let proposals = response.facts.into_iter()
            .map(|f| f.as_proposed())  // Mark as ProposedFact
            .collect();

        AgentResult::Propose(proposals)
    }
}
```

LLMs propose. Validators decide.

Validation: The Trust Layer

```rust
pub trait FactValidator: Send + Sync {
    fn validate(&self, fact: &Fact, ctx: &Context) -> ValidationResult;
}

pub enum ValidationResult {
    Accept,                        // Promote to Validated
    Reject(RejectionReason),       // Discard with reason
    NeedsReview,                   // Human-in-the-loop
}

// Built-in validators
pub struct SchemaValidator;      // Type checking
pub struct InvariantValidator;   // Business rules
pub struct ConsistencyValidator; // No contradictions
```

Key Features

WHAT MAKES CONVERGE-CORE DIFFERENT

1. Merkle-Based Integrity

```
pub struct FactStore {
    root: MerkleRoot,
    nodes: HashMap<FactId, MerkleNode>,
}

impl FactStore {
    pub fn verify(&self) -> bool {
        self.compute_root() == self.root
    }

    pub fn proof(&self, fact_id: &FactId) -> MerkleProof {
        // Generate proof that fact exists in store
    }
}
```

Tamper-evident audit trail. Cryptographic guarantees.

2. Parallel Agent Execution

```rust
impl Engine {
    fn fan_out(&self, ctx: &Context) -> Vec<AgentResult> {
        self.agents
            .par_iter()                      // Rayon parallel iterator
            .filter(|a| a.can_run(ctx))
            .map(|a| a.run(ctx))
            .collect()
    }
}
```

Commutativity axiom enables safe parallelism.

3. Time-Travel Debugging

```
impl ExecutionTrace {
    pub fn replay_to(&self, cycle: usize) -> Context {
        self.snapshots[..=cycle]
            .iter()
            .fold(Context::empty(), |ctx, facts| ctx.derive(facts))
    }

    pub fn diff(&self, from: usize, to: usize) -> Vec<Fact> {
        // Show exactly what changed between cycles
    }
}
```

Replay any execution. Debug any decision.

4. Pluggable LLM Providers

```rust
pub trait LlmProvider: Send + Sync {
    fn complete(&self, prompt: &Prompt) -> LlmResult<Response>;
    fn embed(&self, text: &str) -> LlmResult<Embedding>;
}

// Implementations
pub struct AnthropicProvider;   // Claude
pub struct OpenAIProvider;      // GPT-4
pub struct OllamaProvider;      // Local models
pub struct MockProvider;        // Testing
```

Swap providers without changing agent code.

5. Invariant DSL

```
invariant! {
    name: "no_negative_balance",
    description: "Account balance must never go negative",
    check: |ctx| {
        ctx.facts_of_type::<Balance>()
            .all(|b| b.amount >= 0)
    }
}

invariant! {
    name: "invoice_requires_delivery",
    description: "Cannot invoice without delivery proof",
    check: |ctx| {
        ctx.facts_of_type::<Invoice>()
            .all(|inv| ctx.has_delivery_for(&inv.work_id))
    }
}
```

Performance

BUILT FOR PRODUCTION

Benchmarks

Operation	Throughput	Latency (p99)
Fact insertion	100K/sec	0.1ms
Context derivation	50K/sec	0.2ms
Agent execution (pure)	10K/sec	1ms
Agent execution (LLM)	100/sec	500ms
Merkle verification	1M/sec	0.01ms

Measured on M2 MacBook Pro, single-threaded

Memory Model

```
Context Size vs Facts:
  1K facts   →   ~100 KB
  10K facts  →   ~1 MB
  100K facts →   ~10 MB
  1M facts   →   ~100 MB

Derivation is O(n) where n = new facts only
Full context never copied, only extended
```

Structural sharing via immutable data structures.

Getting Started

FROM ZERO TO RUNNING

Installation

```toml
# Cargo.toml
[dependencies]
converge-core = "0.1"
converge-provider = "0.1"  # LLM providers
tokio = { version = "1", features = ["full"] }
```

```
# Or use the CLI
cargo install converge-cli
converge new my-project
```

Hello, Converge

```rust
use converge_core::{Engine, Context, Agent, Fact};

struct HelloAgent;

impl Agent for HelloAgent {
    fn name(&self) -> &str { "hello" }
    fn can_run(&self, ctx: &Context) -> bool {
        !ctx.has_fact_of_type::<Greeting>()
    }
    fn run(&self, _ctx: &Context) -> AgentResult {
        AgentResult::Propose(vec![
            Fact::derived("greeting", Greeting { message: "Hello, Converge!" })
        ])
    }
}

fn main() {
    let mut engine = Engine::new();
    engine.register(HelloAgent);
    let result = engine.run(Context::empty());
    println!("{:?}", result);
}
```

Project Structure

```
my-project/
├── Cargo.toml
├── src/
│   ├── main.rs
│   ├── agents/          # Your agent implementations
│   │   ├── mod.rs
│   │   └── invoice.rs
│   ├── facts/           # Domain fact types
│   │   ├── mod.rs
│   │   └── money.rs
│   └── invariants/      # Business rules
│       └── mod.rs
└── specs/               # Converge Truths (.truth files)
    └── money.truth
```

Documentation & Resources

- **Docs**: docs.rs/converge-core
- **Examples**: github.com/kpernyer/converge-core/examples
- **Discord**: discord.gg/converge
- **Blog**: converge.zone/signals

Let's Talk

Kenneth Pernyer · Founder

LinkedIn Twitter GitHub

crates.io converge-core hex.pm converge_ledger ghcr.io converge-ledger