Phase 1C

Perfect—here's the fold-in for Phase 1 / Mission 1C. It settles the deterministic bus design and gives you drop-in code + tests that match the research you gathered.

1C — Decisions (locked)

- **API (minimal, testable): publish(msg), history(), clear().**

Extras for observability: historyEnvelopes(), size(), capacity().

- **Determinism & ordering: synchronous publish; global FIFO via a monotonic seq counter (single process, single thread). Unit tests assert call order.**
- **Causality metadata (prep for later phases): optional causationId, correlationId, and ts. We store them in an envelope but keep history() returning just Message[] to avoid ripples.**
- **Memory safety: bounded log with a small circular buffer (default 512). Oldest entries drop when capacity is exceeded. clear() for tests.**
- **When to add IDs: we add seq now (free determinism). Keep causationId/correlationId optional until Phase 3+ (PR automation / tracing). Avoid asserting on timestamps in tests.**

———

Repo changes (copy-paste)

1) Deterministic in-memory bus (bounded FIFO + envelopes)

apps/orchestrator/src/bus/inMemoryBus.ts

```
import type { Message } from '../schema/messages';
export interface Envelope {
  seq: number;           // monotonic sequence for global FIFO
  turn: number;          // redundant but convenient for queries
  msg: Message;          // the actual payload (schema-validated)
  causationId?: string;  // optional: parent message id
  correlationId?: string; // optional: conversation root id
  ts?: number;           // optional: timestamp (avoid in assertions)
}
/**
 * Deterministic, single-process, synchronous event bus with a bounded log.
 * - publish(msg): appends to a circular buffer, returns seq
 * - history(): returns Message[] only (stable public API)
 * - historyEnvelopes(): returns Envelope[] for tracing/telemetry
 * - clear(): resets state (useful for tests)
 */
export class InMemoryBus {
  private log: Envelope[] = [];
  private seq = 0;
  constructor(private readonly maxEntries = 512) {}
  publish(
    msg: Message,
    meta?: Partial<Omit<Envelope, 'seq' | 'turn' | 'msg'>>
  ): number {
    const env: Envelope = {
      seq: ++this.seq,
      turn: msg.turn,
      msg,
      causationId: meta?.causationId,
      correlationId: meta?.correlationId,
      ts: meta?.ts ?? 0
    };
    if (this.log.length >= this.maxEntries) {
      // circular buffer: drop oldest to keep memory bounded
      this.log.shift();
    }
    this.log.push(env);
    return env.seq;
  }
  history(): Message[] {
```

```
    // Return payload only (keeps kernel/tests simple & stable)
    return this.log.map((e) => e.msg);
  }
  historyEnvelopes(): Envelope[] {
    // For tracing or later telemetry; do NOT mutate callers
    return this.log.slice();
  }
  size(): number {
    return this.log.length;
  }
  clear(): void {
    this.log = [];
    this.seq = 0;
  }
}
Kernel already imports InMemoryBus; no behavior change required.
———
2) Bus unit tests (order, boundedness, clear)
apps/orchestrator/tests/bus.spec.ts
import { describe, it, expect } from 'vitest';
import { InMemoryBus } from '../src/bus/inMemoryBus';
const mk = (overrides: Partial<{
  role: 'architect'|'builder',
  type: 'propose'|'critique'|'implement'|'verify',
  content: string,
  turn: number
}> = {}) => ({
  role: overrides.role ?? 'architect',
  type: overrides.type ?? 'propose',
  content: overrides.content ?? 'msg',
  turn: overrides.turn ?? 1
});
describe('InMemoryBus', () => {
  it('publishes synchronously in FIFO order', () => {
    const bus = new InMemoryBus(10);
    bus.publish(mk({content: 'A', turn: 1}));
    bus.publish(mk({content: 'B', turn: 1}));
    bus.publish(mk({content: 'C', turn: 1}));
    const h = bus.history().map(m => m.content);
    expect(h).toEqual(['A', 'B', 'C']);
  });
  it('is bounded: drops oldest when capacity exceeded', () => {
    const bus = new InMemoryBus(2);
    bus.publish(mk({content: 'A'}));
    bus.publish(mk({content: 'B'}));
    bus.publish(mk({content: 'C'})); // A should be evicted
    const h = bus.history().map(m => m.content);
    expect(h).toEqual(['B', 'C']);
    expect(bus.size()).toBe(2);
  });
  it('clear() resets log and sequence', () => {
    const bus = new InMemoryBus(5);
    bus.publish(mk({content: 'X'}));
    expect(bus.size()).toBe(1);
    bus.clear();
    expect(bus.size()).toBe(0);
    // After clear, first publish should have seq=1 again
    const seq = bus.publish(mk({content: 'Y'}));
    expect(seq).toBe(1);
```

```
  });
  it('exposes envelopes for tracing without affecting history()', () => {
    const bus = new InMemoryBus(3);
    bus.publish(mk({content: 'A', turn: 1}), { correlationId: 'corr-1' });
    const envs = bus.historyEnvelopes();
    expect(envs[0].seq).toBe(1);
    expect(envs[0].msg.content).toBe('A');
    expect(envs[0].correlationId).toBe('corr-1');
    // history() remains payload-only
    expect(bus.history()[0]).toHaveProperty('role');
    expect(bus.history()[0]).not.toHaveProperty('seq');
  });
});
```

Notes that tie back to the research

- **Synchronous dispatch + registration order ⇒ deterministic FIFO (Node EventEmitter behavior mirrored, but we keep our own history() for testing).**
- **Monotonic sequence ⇒ global total order (single process), simple to assert in tests; analogous to Kafka's per-partition ordering.**
- **Causation/Correlation IDs ⇒ optional tracing hooks (Greg Young pattern) without affecting Phase-1 tests.**
- **Bounded log ⇒ circular buffer prevents unbounded growth during local runs; clear() keeps tests isolated.**
- **No timestamps in assertions ⇒ avoid clock flakiness; use sequence/turn instead.**

1C "done" check

- **InMemoryBus implements publish/history/historyEnvelopes/clear, with bounded FIFO and a monotonic seq.**
- **Tests cover ordering, boundedness, clear(), and envelope access.**
- **Kernel/agents remain deterministic; smoke:p1 still prints CONVERGED.**

When you've committed these, send over 1D and I'll fold in the diff/patch conventions next.