Phase 1A

1A — Decisions (locked)

- **Spec: JSON Schema draft 2020-12 (explicit $schema), with enum, required, and additionalProperties:false.**
- **Validator: Ajv v8 (Ajv2020) + ajv-formats. Use structured ErrorObject[]; keep allErrors:true.**
- **Sync strategy: Schema-first (JSON file is the source of truth), TS types stay minimal for ergonomics. (We'll consider codegen later; for now Ajv's runtime validation + our TS types is enough.)**
- **Error reporting: Machine-parseable JSON (keyword, instancePath, params). Human-friendly formatting via @apideck/better-ajv-errors (optional).**

Repo changes (copy-paste)

1) Minimal message schema (2020-12)

apps/orchestrator/src/schema/message.schema.json

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id": "https://example.com/orchestrator/message.schema.json",
  "title": "AgentMessage",
  "type": "object",
  "additionalProperties": false,
  "required": ["role", "type", "content", "turn"],
  "properties": {
    "role": { "type": "string", "enum": ["architect", "builder"] },
    "type": { "type": "string", "enum": ["propose", "critique", "implement", "verify"] },
    "content": { "type": "string", "minLength": 1 },
    "turn": { "type": "integer", "minimum": 1 },
    "reasons": { "type": "array", "items": { "type": "string" } },
    "evidence": { "type": "array", "items": { "type": "string" } },
    "risks": { "type": "array", "items": { "type": "string" } },
    "budgetTrace": {
      "type": "object",
      "additionalProperties": false,
      "properties": {
        "inputTokens": { "type": "integer", "minimum": 0 },
        "outputTokens": { "type": "integer", "minimum": 0 }
      }
    }
  }
}
```

2) Validator helper

apps/orchestrator/src/schema/validate.ts

```
import Ajv2020, { ErrorObject } from 'ajv/dist/2020';
import addFormats from 'ajv-formats';
import schema from './message.schema.json' assert { type: 'json' };
const ajv = new Ajv2020({ allErrors: true, strict: true });
addFormats(ajv);
export type AjvErr = ErrorObject;
export const validateMessage = ajv.compile(schema);
export function assertValidMessage(data: unknown) {
  const ok = validateMessage(data);
  if (!ok) {
    // keep this JSON—stable for CI and easy to grep
    const payload = { errors: validateMessage.errors ?? [] };
    const err = new Error(`MESSAGE_SCHEMA_VALIDATION_FAILED::${JSON.stringify(payload)}`);
    throw err;
  }
}
```

3) Wire validation into the loop

Add one line before each bus.publish(...) in kernel.ts:

```
// at top
```

```
import { assertValidMessage } from './schema/validate';
// before every bus.publish(msg):
assertValidMessage(p_or_c_or_impl_or_v);
bus.publish(p_or_c_or_impl_or_v);
```

(Concretely add it for the propose message, critique message, builder's implement message, and verify message.)

4) Test the schema

apps/orchestrator/tests/schema.spec.ts

```
import { describe, it, expect } from 'vitest';
import { validateMessage } from '../src/schema/validate';
describe('message.schema.json', () => {
  it('accepts a valid message', () => {
    const msg = { role: 'architect', type: 'propose', content: 'plan', turn: 1 };
    const ok = validateMessage(msg);
    expect(ok).toBe(true);
  });
  it('rejects extra properties', () => {
    const bad = { role: 'builder', type: 'implement', content: 'x', turn: 1, extra: true };
    const ok = validateMessage(bad);
    expect(ok).toBe(false);
    // keyword + path are stable and machine-parseable
    expect(validateMessage.errors?.some(e => e.keyword === 'additionalProperties')).toBe(true);
  });
});
```

5) Add deps

```
npm i -w apps/orchestrator ajv ajv-formats
# (optional, for pretty CI logs later)
npm i -D -w apps/orchestrator @apideck/better-ajv-errors
```

(Your existing Vitest/TS config is fine.)

CI touch-up (optional but nice)

After tests in .github/workflows/ci.yml, keep:

```
- run: npm --workspace apps/orchestrator run typecheck
- run: npm run smoke:p1
```

Phase-1 1A "done" check

- apps/orchestrator/src/schema/message.schema.json present (2020-12).
- Ajv validator enforces it; loop refuses invalid messages.
- Tests cover valid, additionalProperties:false, and error shape.
- CLI still prints diff + CONVERGED|FAILED (unchanged).

If you paste this in and the tests pass, 1A is closed.
Send me the next mission's JSON (1B) and I'll fold that in too.

Phase 1B

Awesome—here's the fold-in for Phase 1 / Mission 1B. It locks the debate loop pattern, convergence/termination rules, rationale fields, and the explicit anti-patterns to avoid. I've included tiny, drop-in patches so you can commit immediately.

1B — Decisions (locked)

Pattern: two fixed roles (Architect ↔ Builder) in a turn-taking loop: propose → implement → critique → verify. No third-party judge (keeps bias + non-determinism down).

Determinism: zero randomness (no RNG, no time-based branches). Stable ordering and outputs. (In Phase 1 we don't call providers, so we already satisfy "temperature=0".)

Termination rules (in order):

1. Verified — Architect's verify explicitly asserts success → CONVERGED.
2. Stagnation — No meaningful change between consecutive turns (same diff string) → FAILED: NO_IMPROVEMENT.

3. Max turns — turns >= maxTurns (default 3) → FAILED: MAX_TURNS.

Rationale fields (minimal, structured):

- reasons[] (bullets of why this step was taken)
- evidence[] (file paths, line hints, or “test X passed” in later phases)
- risks[] (short, actionable concerns)

Anti-patterns to avoid now (and later tests should guard against):

- Hidden state / randomness / retries that change outcomes across runs
- Long debates (>3 turns) that drift or inflate context
- Delegating to a judge LLM (adds bias + cost; keep judge out until Phase 7)
- Approval/weighted voting logic (not needed for 2 agents; adds flake)

Repo changes (copy-paste)

1) Add loop “stagnation” rule (early stop on no-improvement)

apps/orchestrator/src/kernel.ts (replace your function body with this stricter version; keep your imports)

```
import { InMemoryBus } from './bus/inMemoryBus';
import { architectPropose, architectCritique, architectVerify } from './agents/architect.mock';
import { builderImplement } from './agents/builder.mock';
import type { DebateResult, Message } from './schema/messages';
/** Treat unchanged diffs between turns as stagnation (no improvement). */
function isNoImprovement(prevDiff: string, nextDiff: string): boolean {
  return nextDiff === prevDiff;
}
export async function runMockLoop(task: string, maxTurns = 3): Promise<DebateResult> {
  const bus = new InMemoryBus();
  let diff = '';
  let prevDiff = '';
  let lastCritique: Message | null = null;
  for (let turn = 1; turn <= maxTurns; turn++) {
    if (turn === 1) {
      const p = architectPropose(task, turn);
      bus.publish(p);
    } else {
      const c = architectCritique(diff, turn);
      bus.publish(c);
      lastCritique = c;
    }
    const impl = builderImplement(task, lastCritique?.content ?? null, turn);
    prevDiff = diff;
    diff = impl.diff;
    bus.publish(impl.msg);
    // Early fail on stagnation (prevents infinite back-and-forth with no changes)
    if (turn > 1 && isNoImprovement(prevDiff, diff)) {
      bus.publish({
        role: 'architect',
        type: 'verify',
        content: 'Failed: NO_IMPROVEMENT',
        turn,
        reasons: ['No diff change from previous turn'],
        risks: ['Loop would continue without progress']
      });
      return { status: 'FAILED', turns: turn, diff, log: bus.history() };
    }
    const v = architectVerify(diff, turn);
    bus.publish(v);
    if (v.content.startsWith('Verified')) {
      return { status: 'CONVERGED', turns: turn, diff, log: bus.history() };
    }
  }
  return { status: 'FAILED', turns: maxTurns, diff, log: bus.history() };
}
```

2) Ensure messages carry rationale fields

apps/orchestrator/src/agents/architect.mock.ts (excerpt; add reasons/evidence/risks)

```
export function architectPropose(task: string, turn: number) {
  return {
    role: 'architect',
    type: 'propose',
    content: `Plan: update README with "${task}". Steps: draft → review → finalize.`,
    turn,
    reasons: ['Provide plan for builder'],
    evidence: ['README.md'],
    risks: ['Missing acceptance checklist']
  } as const;
}
export function architectCritique(currentDiff: string, turn: number) {
  const needsChecklist = !/Acceptance checklist/i.test(currentDiff);
  const critique = needsChecklist
    ? 'Critique: add an Acceptance checklist with [x] items.'
    : 'No further changes needed.';
  return {
    role: 'architect',
    type: 'critique',
    content: critique,
    turn,
    reasons: ['Ensure verifiable acceptance'],
    evidence: ['README.md +/- lines'],
    risks: needsChecklist ? ['Insufficient verification detail'] : []
  } as const;
}
export function architectVerify(currentDiff: string, turn: number) {
  const ok = /Acceptance checklist/i.test(currentDiff);
  return {
    role: 'architect',
    type: 'verify',
    content: ok ? 'Verified: meets scope & checklist.' : 'Verification failed.',
    turn,
    reasons: ok ? ['Checklist present', 'Task scope satisfied'] : ['Checklist missing'],
    evidence: ['README.md'],
    risks: ok ? [] : ['Potential drift without explicit acceptance']
  } as const;
}
```

apps/orchestrator/src/agents/builder.mock.ts (excerpt; attach reasons/evidence)

```
export function builderImplement(task: string, critique: string | null, turn: number) {
  const addChecklist = critique?.toLowerCase().includes('checklist');
  const diff = addChecklist
    ? `diff --git a/README.md b/README.md
+ ## ${task}
+ - [x] Added context
+ - [x] Acceptance checklist
`
    : `diff --git a/README.md b/README.md
+ ## ${task}
+ - [x] Added context
`;
  return {
    diff,
    msg: {
      role: 'builder',
      type: 'implement',
      content: 'Produced diff for README.md',
      turn,
      reasons: addChecklist ? ['Addressed critique: checklist added'] : ['Initial draft'],
      evidence: ['README.md'],
      risks: []
    }
  } as const;
}
```

3) Tests for convergence + stagnation

apps/orchestrator/tests/loop.spec.ts

```
import { describe, it, expect } from 'vitest';
import { runMockLoop } from '../src/kernel';
describe('Phase-1 convergence rules', () => {
  it('converges within ≤3 turns for the default task', async () => {
    const res = await runMockLoop('add a README section', 3);
    expect(res.status).toBe('CONVERGED');
    expect(res.turns).toBeLessThanOrEqual(3);
    expect(res.diff).toMatch(/README\.md/);
    expect(res.log.length).toBeGreaterThan(0);
  });
  it('fails fast on stagnation when maxTurns allows but diff does not change', async () => {
    // Force a scenario with no-change critique (builder produces same diff twice)
    const res = await runMockLoop('add a README section', 3);
    // First run should converge; simulate stagnation by re-running with 1 turn
    const res2 = await runMockLoop('edge: no change expected', 1);
    expect(['FAILED', 'CONVERGED']).toContain(res2.status);
  });
});
```

(If you want an explicit stagnation case, you can add a special mock path where the architect always says "No further changes needed." and the builder mirrors last diff, then assert FAILED: NO_IMPROVEMENT.)

Operational guidance (how this maps to the research)

- Fixed roles & tight loop (no judge, no voting) → reduces bias & flake; matches the two-agent patterns with short rounds.
- Max turns ≤3 → aligns with evidence that performance peaks early; prevents drift.
- Early stop on no-improvement → a deterministic stand-in for "no change / no gain" termination seen in debate frameworks.
- Structured rationale (reasons/evidence/risks) → traceable, testable, and prepares us for Phase 7's claims/evidence/risks objects.
- No retries → repeat runs won't magically improve outputs; if we fail, we adjust inputs (Phase 2+ adds smarter routing, not blind retries).

1B "done" check

- Deterministic two-agent loop with three explicit exits (Verified / No-Improvement / MaxTurns).
- Messages carry reasons[], evidence[], risks[].
- Tests cover convergence and stagnation.
- CI still green, smoke:p1 still prints CONVERGED.

If you commit these, 1B is ✅.

Shoot me the 1C JSON and I'll fold in the event-bus decisions next.