comparison

SPEC-PPP-004: Comparative Analysis

Last Updated: 2025-11-16

Observability Platform Comparison

Feature	LangSmith	Helicone	Phoenix Arize	Prop (SQI
Multi-turn Tracking	✅ Threads	✅ Sessions	✅ Sessions	✅ Traje
Trace Hierarchy	✅ Trace → Run	✅ Path syntax	✅ Spans	✅ Traje → Turn
Deployment	❌ Cloud only	✅ Self-hosted	✅ Self-hosted	✅ Local SQLite
License	Proprietary	Apache 2.0 (OSS)	Apache 2.0 (OSS)	N/A (pr code)
Pricing	$39/mo+	Free (OSS)	Free (OSS)	**FREE**
Integration	LangChain required	HTTP proxy	OpenTelemetry	**Native**
Latency Overhead	50-100ms	10-20ms	30-50ms	**<1ms (async)**
Question Effort	❌ None	❌ None	❌ None	✅ Low/Me
Preference Violations	❌ None	❌ None	❌ None	✅ Full tracking
Interaction Scoring	❌ None	❌ None	⚠️ Coherence	✅ R_{Proa} + R_{Pers}
Session-level Metrics	⚠️ Multi-turn evals	⚠️ Basic	✅ Comprehensive	✅ PPP formula
CLI-Specific Features	❌ None	❌ None	❌ None	✅ SPEC tracking
Storage Location	Cloud	Cloud/local	Local	**Local o**
Privacy	⚠️ External	⚠️ Depends	✅ Self-hosted	✅ **100%**
Rust Support	⚠️ Limited	⚠️ HTTP API	⚠️ Python-first	✅ **Nativ**
PPP Compliance	10%	15%	20%	**100%**

Best For	LangChain apps	Multi-provider apps	Research/analysis	**PPP CI tools**

Winner: **Proposed (SQLite)** - Only solution meeting all PPP requirements with zero external dependencies.

Async SQLite Crate Comparison

Feature	tokio-rusqlite	async-sqlite	sqlx	rusqlite (sync)
Maturity	✅ Mature	⚠ Young	✅ Very mature	✅ Very mature
Downloads/Month	23,000	<1,000	500,000+	1,000,000+
License	MIT	MIT	MIT/Apache 2.0	MIT
Async Runtime	tokio only	tokio + async_std	tokio + async_std	N/A (sync)
API Style	Callback-based	Direct async	SQL builder	Direct sync
Compile-time Checks	✖ None	✖ None	✅ Full	✖ None
Connection Pooling	⚠ Manual	✖ None	✅ Built-in	⚠ Manual
Migrations	✖ None	✖ None	✅ Built-in	✖ None
Binary Size	+50KB	+50KB	+2MB	+500KB
Latency (single op)	~0.5ms	~0.5ms	~0.3ms	~0.1ms
Latency (batched)	~0.1ms	~0.1ms	~0.05ms	~0.02ms
Learning Curve	⚠ Medium	✅ Low	⚠ Medium	✅ Low
Thread Safety	✅ Yes	✅ Yes	✅ Yes	⚠ Manual
Error Handling	✅ Good	⚠ Basic	✅ Excellent	✅ Good
Documentation	✅ Good	⚠ Limited	✅ Excellent	✅ Excellen
Community	⚠ Medium	⚠ Small	✅ Large	✅ Very large
Best For	Async logging	Simple cases	Full apps	Sync batch ops

Recommendation: **tokio-rusqlite** for Phase 1-2 (proven, async-friendly), consider **sqlx** for Phase 3 if migrations needed.

Database Schema Pattern Comparison

Query

Pattern	Tables	Complexity	Storage	Extensibility	S
Flat Message History	1	✅ Low	✅ Compact	❌ Poor	❌
Turn-Based (Proposed)	4	⚠ Medium (JOINs)	⚠ Normalized	✅ Excellent	✅
JSON Blob	1	❌ High (no index)	❌ Large	⚠ Good	⚠
Hierarchical Spans	2	❌ Very high (CTE)	⚠ Medium	✅ Excellent	⚠

Details:

1. Flat Message History

Schema:

```
CREATE TABLE messages (
    id INTEGER PRIMARY KEY,
    session_id TEXT,
    timestamp TEXT,
    role TEXT,
    content TEXT
);
```

Pros: - Simplest schema (1 table) - Easy chronological queries - Minimal storage overhead

Cons: - No turn grouping (prompt + response = 2 rows) - Can't store metadata (effort, violations) - Hard to calculate R_{Proact} (questions not tracked)

PPP Compliance: 20% (basic conversation history only)

---

2. Turn-Based with Metadata (Proposed)

Schema:

```
CREATE TABLE trajectories (
    id INTEGER PRIMARY KEY,
    spec_id TEXT NOT NULL,
    agent_name TEXT NOT NULL,
    run_id TEXT,
    created_at TEXT DEFAULT (datetime('now')),
    INDEX idx_spec_agent (spec_id, agent_name)
);

CREATE TABLE trajectory_turns (
    id INTEGER PRIMARY KEY,
    trajectory_id INTEGER NOT NULL,
    turn_number INTEGER NOT NULL,
    prompt TEXT NOT NULL,
    response TEXT NOT NULL,
    token_count INTEGER,
    latency_ms INTEGER,
```

```
            FOREIGN KEY (trajectory_id) REFERENCES trajectories(id) ON
DELETE CASCADE
        );

        CREATE TABLE trajectory_questions (
            id INTEGER PRIMARY KEY,
            turn_id INTEGER NOT NULL,
            question_text TEXT NOT NULL,
            effort_level TEXT,  -- 'low', 'medium', 'high'
            FOREIGN KEY (turn_id) REFERENCES trajectory_turns(id) ON DELETE
CASCADE
        );

        CREATE TABLE trajectory_violations (
            id INTEGER PRIMARY KEY,
            turn_id INTEGER NOT NULL,
            preference_name TEXT NOT NULL,
            expected TEXT NOT NULL,
            actual TEXT NOT NULL,
            severity TEXT NOT NULL,
            FOREIGN KEY (turn_id) REFERENCES trajectory_turns(id) ON DELETE
CASCADE
        );
```

Pros: - Turn-level granularity (prompt + response = 1 row) - Normalized (no duplication) - Supports all PPP calculations - Extensible (add tables without migration)

Cons: - More complex queries (requires JOINs) - 4 tables to manage - Slightly higher write overhead

PPP Compliance: 100% (supports all 20 preferences + formulas)

Example Query (R_{Proact} calculation):

```
        SELECT
            CASE
                WHEN COUNT(q.id) = 0 THEN 0.05
                WHEN COUNT(CASE WHEN q.effort_level != 'low' THEN 1 END) = 0
THEN 0.05
                ELSE -0.1 * COUNT(CASE WHEN q.effort_level = 'medium' THEN 1
END)
                     -0.5 * COUNT(CASE WHEN q.effort_level = 'high' THEN 1
END)
            END AS r_proact
        FROM trajectory_turns t
        LEFT JOIN trajectory_questions q ON t.id = q.turn_id
        WHERE t.trajectory_id = ?;
```

---

3. JSON Blob Storage

Schema:

```
        CREATE TABLE trajectories (
            id INTEGER PRIMARY KEY,
            spec_id TEXT,
            agent_name TEXT,
            turns_json TEXT  -- JSON array: [{prompt, response, questions,
violations}, ...]
```

```
);
```

Example JSON:

```
{
  "turns": [
    {
      "prompt": "Implement OAuth",
      "response": "Which provider?",
      "questions": [{"text": "Which provider?", "effort": "low"}],
      "violations": []
    }
  ]
}
```

Pros: - Flexible schema (no migrations) - Single INSERT per trajectory - Easy to serialize/deserialize

Cons: - SQLite JSON support limited (no indexing in <3.38) - Large storage footprint (JSON overhead ~30%) - Can't enforce foreign keys - Hard to query (requires `json_extract()`)

PPP Compliance: 50% (can store data, but hard to calculate scores)

---

4. Hierarchical with Parent/Child Spans (OpenTelemetry)

Schema:

```
CREATE TABLE spans (
    id INTEGER PRIMARY KEY,
    parent_id INTEGER,
    span_type TEXT,  -- 'conversation', 'turn', 'llm_call',
'tool_use'
    start_time TEXT,
    end_time TEXT,
    attributes_json TEXT,
    FOREIGN KEY (parent_id) REFERENCES spans(id)
);
```

Example Hierarchy:

```
Span (type: conversation, id: 1)
  ├── Span (type: turn, id: 2, parent: 1)
  │     ├── Span (type: llm_call, id: 3, parent: 2)
  │     └── Span (type: tool_use, id: 4, parent: 2)
  └── Span (type: turn, id: 5, parent: 1)
```

Pros: - Industry standard (OpenTelemetry) - Supports nested operations (tool calls, retries) - Migration path to observability platforms

Cons: - Overkill for CLI (no nested tools) - Complex queries (recursive CTEs) - Higher storage overhead (~2x)

PPP Compliance: 60% (can model turns, but not optimized for PPP formulas)

Example Query (recursive CTE to get all turns):

```
WITH RECURSIVE turn_hierarchy AS (
    SELECT * FROM spans WHERE span_type = 'conversation' AND id = ?
    UNION ALL
    SELECT s.* FROM spans s
    JOIN turn_hierarchy th ON s.parent_id = th.id
)
SELECT * FROM turn_hierarchy WHERE span_type = 'turn';
```

Performance Strategy Comparison

Strategy	Latency (per turn)	Throughput	Complexity	Concurrency
Sync (single)	~1ms	1,000/sec	✅ Low	❌ Blocks
Sync (batched)	~0.067ms	15,000/sec	⚠️ Medium	❌ Blocks
Async (single)	~0.5ms	2,000/sec	⚠️ Medium	✅ Good
Async (batched)	**~0.1ms**	**10,000/sec**	⚠️ Medium	✅ Excellent
Async (WAL mode)	~0.02ms	50,000/sec	❌ High	✅ Excellent

Details:

1. Synchronous Single Insert

Implementation:

```
use rusqlite::Connection;

let conn = Connection::open("trajectory.db")?;
conn.execute(
    "INSERT INTO trajectory_turns (trajectory_id, prompt, response)
VALUES (?1, ?2, ?3)",
    params![traj_id, prompt, response],
)?;
```

Performance: - Latency: ~1ms per turn - Throughput: ~1,000 turns/sec - Blocking: Yes (blocks tokio runtime)

Verdict: ❌ **Avoid** - Blocks async runtime, too slow for production.

---

2. Synchronous Batched Insert

Implementation:

```
let tx = conn.transaction()?;
for turn in turns {
    tx.execute(
        "INSERT INTO trajectory_turns (trajectory_id, prompt,
```

```
response) VALUES (?1, ?2, ?3)",
                params![turn.traj_id, turn.prompt, turn.response],
            )?;
        }
        tx.commit()?;
```

Performance: - Latency: ~0.067ms per turn (15,000/sec) - Throughput: 15,000 turns/sec - Blocking: Yes (but brief)

Verdict: ⚠ **Acceptable** - Good for background batch jobs, but still blocks.

---

3. Asynchronous Single Insert

Implementation:

```
        use tokio_rusqlite::Connection;

        let conn = Connection::open("trajectory.db").await?;
        conn.call(|conn| {
            conn.execute(
                "INSERT INTO trajectory_turns (trajectory_id, prompt,
response) VALUES (?1, ?2, ?3)",
                params![traj_id, prompt, response],
            )
        }).await?;
```

Performance: - Latency: ~0.5ms per turn (thread pool overhead) - Throughput: ~2,000 turns/sec - Blocking: No (async)

Verdict: ⚠ **Acceptable** - Good for low-volume real-time logging.

---

4. Asynchronous Batched Insert (RECOMMENDED)

Implementation:

```
        use tokio::sync::mpsc;
        use tokio_rusqlite::Connection;

        // Channel for buffered writes
        let (tx, mut rx) = mpsc::channel::<Turn>(100);

        // Background writer task
        tokio::spawn(async move {
            let conn = Connection::open("trajectory.db").await?;
            let mut buffer = Vec::new();

            loop {
                tokio::select! {
                    Some(turn) = rx.recv() => {
                        buffer.push(turn);
                        if buffer.len() >= 10 {
                            flush_batch(&conn, &buffer).await?;
                            buffer.clear();
                        }
                    }
                    _ = tokio::time::sleep(Duration::from_millis(500)) => {
                        if !buffer.is_empty() {
```

```
                        flush_batch(&conn, &buffer).await?;
                        buffer.clear();
                    }
                }
            }
        }
    });

    async fn flush_batch(conn: &Connection, turns: &[Turn]) ->
Result<()> {
        conn.call(move |conn| {
            let tx = conn.transaction()?;
            for turn in turns {
                tx.execute(
                    "INSERT INTO trajectory_turns (trajectory_id,
prompt, response) VALUES (?1, ?2, ?3)",
                    params![turn.traj_id, turn.prompt, turn.response],
                )?;
            }
            tx.commit()
        }).await
    }
```

Performance: - Latency: ~0.1ms per turn (amortized) - Throughput: ~10,000 turns/sec - Blocking: No (async + buffered) - Lock contention: Minimal (batched writes)

Verdict: ✅ **RECOMMENDED** - Best balance of latency, throughput, and concurrency.

---

5. Asynchronous with WAL Mode

Implementation:

```
    let conn = Connection::open("trajectory.db").await?;
    conn.call(|conn| {
        conn.execute("PRAGMA journal_mode=WAL", [])?;
        conn.execute("PRAGMA synchronous=NORMAL", [])?;
        Ok(())
    }).await?;
```

Performance: - Latency: ~0.02ms per turn - Throughput: 50,000+ turns/sec - Blocking: No (WAL allows concurrent reads)

Pros: - Fastest option (10x faster than batched) - Concurrent reads during writes - No lock contention

Cons: - More complex (WAL files to manage) - Slightly larger disk footprint (~30%)

Verdict: ⚠ **Future optimization** - Use if Phase 1-2 benchmarks show bottleneck.

Integration Strategy Comparison

Approach	Complexity	Reuse	Migration	Latency	B F

Extend consensus_db.rs	✅ Low	✅ High	✅ Easy	**<1ms**	**Pha 1-2**
Separate Database	⚠ Medium	⚠ Low	⚠ Manual	<1ms	Isol nee
MCP Server	❌ High	❌ None	❌ Hard	~5ms	Ext tool

Details:

1. Extend consensus_db.rs (RECOMMENDED)

Approach: Add trajectory tables to existing SQLite database managed by `consensus_db.rs`.

Implementation:

```rust
// codex-rs/tui/src/chatwidget/spec_kit/consensus_db.rs

pub fn init_trajectory_tables(conn: &Connection) -> Result<()> {
    conn.execute(
        "CREATE TABLE IF NOT EXISTS trajectories (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            spec_id TEXT NOT NULL,
            agent_name TEXT NOT NULL,
            run_id TEXT,
            created_at TEXT NOT NULL DEFAULT (datetime('now'))
        )",
        [],
    )?;

    conn.execute(
        "CREATE INDEX IF NOT EXISTS idx_trajectories_spec_agent
         ON trajectories(spec_id, agent_name)",
        [],
    )?;

    // ... other tables

    Ok(())
}
```

Pros: - ✅ Reuses existing connection pool - ✅ Single database file (simpler deployment) - ✅ Can JOIN with consensus_artifacts (via run_id) - ✅ Minimal code changes

Cons: - ⚠ Couples trajectory logging to consensus DB - ⚠ Migrations must coordinate with existing schema

Verdict: ✅ **RECOMMENDED** - Simplest, fastest, most maintainable.

---

2. Separate Database

Approach: Create new `trajectory.db` separate from `consensus.db`.

Implementation:

```rust
// New module: codex-rs/tui/src/chatwidget/spec_kit/trajectory_db.rs
```

```rust
pub struct TrajectoryDb {
    conn: Arc<Mutex<Connection>>,
}

impl TrajectoryDb {
    pub fn open() -> Result<Self> {
        let path = dirs::data_dir()
            .unwrap()
            .join("codex-tui")
            .join("trajectory.db");
        let conn = Connection::open(path)?;
        init_tables(&conn)?;
        Ok(Self { conn: Arc::new(Mutex::new(conn)) })
    }
}
```

Pros: - ✅ Clean separation of concerns - ✅ Independent migrations - ✅ Can be disabled without affecting consensus

Cons: - ❌ Duplicate connection management - ❌ Can't JOIN across databases (require application-level merge) - ❌ More files to manage

Verdict: ⚠ **Alternative** - Use only if trajectory logging needs to be independently deployable.

---

3. MCP Server (NOT RECOMMENDED)

Approach: Create new MCP server for trajectory logging (similar to consensus MCP).

Implementation:

```rust
// New crate: codex-rs/mcp-servers/mcp-trajectory-logger

#[mcp_tool]
async fn log_turn(
    trajectory_id: i64,
    prompt: String,
    response: String,
) -> Result<()> {
    // ... SQLite insert
}
```

Pros: - ✅ Reusable by other tools (not just codex-tui) - ✅ Standard MCP interface

Cons: - ❌ 5x slower (~5ms vs <1ms) - ❌ More complex (JSON-RPC overhead) - ❌ Requires MCP connection management - ❌ No benefit for CLI use case

Verdict: ❌ **NOT RECOMMENDED** - Unnecessary overhead for local logging.

Cost Analysis

Infrastructure Costs

Approach	Setup Cost	Monthly Cost	Storage Cost	Latency Cost
LangSmith	0 hours	$39+	N/A (cloud)	High (50-100ms)
Helicone (cloud)	0 hours	$0 (OSS)	N/A (cloud)	Medium (10-20ms)
Helicone (self-hosted)	4 hours	$5 (VPS)	<1GB/year	Medium (10-20ms)
Phoenix (self-hosted)	4 hours	$5 (VPS)	<1GB/year	Medium (30-50ms)
SQLite (local)	2 hours	**$0**	**<100MB/year**	**Low (<1ms)**

Assumptions (1000 sessions/month, avg 20 turns/session): - Storage: 500 bytes/turn = 10MB/month = 120MB/year - Compute: Local SQLite uses existing machine (no incremental cost)

Winner: SQLite (local) - Zero ongoing cost, minimal storage.

Development Effort Costs

Task	Complexity	Estimated Effort	Dependencies
1. Schema Design	⚠ MEDIUM	4 hours	None
2. Extend consensus_db.rs	✅ LOW	2 hours	consensus_db.rs
3. Async Logging API	⚠ MEDIUM	6 hours	tokio-rusqlite
4. Question Effort Classifier	❌ HIGH	8 hours	SPEC-PPP-001
5. Preference Violation Detector	⚠ MEDIUM	6 hours	SPEC-PPP-002
6. R_{Proact} Calculator	⚠ MEDIUM	4 hours	SPEC-PPP-003
7. R_{Pers} Calculator	⚠ MEDIUM	4 hours	SPEC-PPP-002, 003
8. Integration Tests	⚠ MEDIUM	4 hours	Test harness
9. Documentation	✅ LOW	2 hours	None

Total Effort: ~40 hours (~1 week for 1 engineer)

Phase 1 Target (Schema + basic logging): ~12 hours (1.5 days)

Performance Benchmarks (Estimated)

Component	Operation	Latency	Throughput	Bottleneck
Schema Creation	Init tables	<1ms	N/A	Disk I/O
Insert Turn (sync)	Single INSERT	1ms	1,000/sec	Lock contention
Insert Turn (async)	Single INSERT	0.5ms	2,000/sec	Thread pool
Insert Turn (batched)	Batched INSERT	**0.1ms**	**10,000/sec**	Minimal
Query Trajectory	SELECT with JOINs	2-5ms	200/sec	JOIN complexity
Calculate R_{Proact}	Aggregate query	3-8ms	100/sec	COUNT + CASE
Calculate R_{Pers}	Aggregate query	3-8ms	100/sec	COUNT + violations
Full Weighted Score	Multi-query	10-20ms	50/sec	Multiple JOINs

Expected Overhead (per agent execution): - Logging: ~0.1ms/turn × 20 turns = 2ms - Scoring: ~20ms (once at end) - **Total**: ~22ms (0.1% of typical agent execution time)

Recommendations Summary

Decision	Recommended Option	Alternative	Rationale
Observability Platform	Local SQLite	Phoenix Arize (self-hosted)	Zero cost, <1ms overhead, 100% privacy
Async SQLite Crate	tokio-rusqlite	sqlx	Proven (23k/mo), async-friendly, lightweight
Schema Pattern	Turn-based (4 tables)	JSON blob	Normalized, supports PPP formulas, indexable
Performance Strategy	Async batched	WAL mode	0.1ms/turn, 10K/sec throughput, simple
Integration	Extend consensus_db.rs	Separate DB	Reuses connection, single file,

			simplest
Buffering	5-10 turns, 500ms flush	Immediate write	Reduces lock contention, <1ms amortized
Indexes	spec_id, agent_name, trajectory_id	None	Fast lookups for scoring queries
WAL Mode	Phase 3 optimization	Default	Defer until benchmarks show bottleneck

---

Next Steps

1. **Validate schema design** with trajectory examples from real agent runs
2. **Prototype** async batching with tokio-rusqlite (benchmark actual latency)
3. **Benchmark** scoring queries with 1K+ trajectories (validate <20ms target)
4. **Phase 1 implementation**: Schema + basic logging (defer scoring to Phase 2)
5. **Phase 2 integration**: Connect to SPEC-PPP-003 weighted consensus