recommendations

SPEC-PPP-004: Implementation Recommendations

Last Updated: 2025-11-16 **Target**: Trajectory logging infrastructure for PPP framework **Total Effort**: ~40 hours (1 week, 1 engineer)

Executive Summary

Implement PPP trajectory logging using **local SQLite extension** (NOT MCP server) with **async batching** for <1ms overhead. Extend existing `consensus_db.rs` to add 4 new tables tracking multi-turn conversations, question effort, and preference violations.

Key Decisions: 1. ✅ SQLite (NOT MCP server) - 5x faster, zero cost, simpler deployment 2. ✅ tokio-rusqlite - Proven async SQLite library (23k downloads/month) 3. ✅ Async batching - Buffer 5-10 turns, flush every 500ms (0.1ms/turn overhead) 4. ✅ Extend consensus_db.rs - Reuse existing connection, single database file 5. ✅ Turn-based schema - Normalized 4-table design supporting PPP formulas

---

Phased Rollout

Phase 1: Foundation (12 hours, 1.5 days)

Goal: Basic trajectory logging without scoring

Deliverables: - ✅ SQLite schema (4 tables: trajectories, turns, questions, violations) - ✅ Async logging API with batching - ✅ Integration with existing consensus_db.rs - ✅ Unit tests (CRUD operations)

Capabilities: - Log multi-turn conversations - Track question effort (manual classification) - Record preference violations - Query trajectories by spec_id, agent_name

NOT Included (defer to Phase 2): - Automatic question effort classification - R_{Proact} and R_{Pers} calculation - Weighted consensus integration

---

Phase 2: Scoring Integration (16 hours, 2 days)

Goal: Enable PPP interaction scoring

Deliverables: - ✅ R_{Proact} calculation queries - ✅ R_{Pers} calculation queries - ✅ Integration with SPEC-PPP-003 weighted consensus - ✅ Question effort classifier (heuristic-based, 75-85% accuracy) - ✅ Integration tests (full scoring pipeline)

Capabilities: - Calculate proactivity scores from trajectories - Calculate personalization scores - Use scores in weighted consensus - Automatic question effort detection (basic)

NOT Included (defer to Phase 3): - LLM-based question effort classification (90-95% accuracy) - Observability platform export (OpenTelemetry) - WAL mode optimization

---

Phase 3: Production Hardening (12 hours, 1.5 days)

Goal: Optimize for production workloads

Deliverables: - ✅ WAL mode for concurrent reads - ✅ LLM-based question effort classifier (optional upgrade) - ✅ Archival/cleanup strategy (auto-delete old trajectories) - ✅ Observability export (OpenTelemetry spans, if needed) - ✅ Load testing (10K+ trajectories) - ✅ Performance benchmarks

Capabilities: - <0.02ms/turn latency (WAL mode) - 50,000+ turns/sec throughput - Automatic cleanup (keep last 30 days) - Export to Phoenix Arize (if desired)

Detailed Task Breakdown

Task 1: Database Schema Design (4 hours)

File: `codex-rs/tui/src/chatwidget/spec_kit/consensus_db.rs`

Implementation:

```rust
// Add to existing init_database() function

pub fn init_trajectory_tables(conn: &Connection) -> Result<()> {
    // Table 1: Trajectories (one per agent execution)
    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,  -- Links to consensus_artifacts.run_id
            created_at TEXT NOT NULL DEFAULT (datetime('now')),
            INDEX idx_trajectories_spec_agent (spec_id, agent_name),
            INDEX idx_trajectories_run_id (run_id)
        )",
        [],
    )?;

    // Table 2: Trajectory Turns (one per user-agent exchange)
    conn.execute(
        "CREATE TABLE IF NOT EXISTS trajectory_turns (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
                    trajectory_id INTEGER NOT NULL,
                    turn_number INTEGER NOT NULL,
                    prompt TEXT NOT NULL,
                    response TEXT NOT NULL,
                    token_count INTEGER,
                    latency_ms INTEGER,
                    timestamp TEXT NOT NULL DEFAULT (datetime('now')),
                    FOREIGN KEY (trajectory_id) REFERENCES trajectories(id)
ON DELETE CASCADE,
                    INDEX idx_turns_trajectory (trajectory_id),
                    UNIQUE (trajectory_id, turn_number)
                )",
                [],
            )?;

            // Table 3: Questions Asked (extracted from responses)
            conn.execute(
                "CREATE TABLE IF NOT EXISTS trajectory_questions (
                    id INTEGER PRIMARY KEY AUTOINCREMENT,
                    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,
                    INDEX idx_questions_turn (turn_id)
                )",
                [],
            )?;

            // Table 4: Preference Violations (detected in responses)
            conn.execute(
                "CREATE TABLE IF NOT EXISTS trajectory_violations (
                    id INTEGER PRIMARY KEY AUTOINCREMENT,
                    turn_id INTEGER NOT NULL,
                    preference_name TEXT NOT NULL,  -- e.g., 'no_ask',
'require_json'
                    expected TEXT NOT NULL,         -- e.g., 'no questions',
'valid JSON'
                    actual TEXT NOT NULL,           -- e.g., 'asked
question', 'malformed JSON'
                    severity TEXT NOT NULL,         -- 'minor', 'major',
'critical'
                    FOREIGN KEY (turn_id) REFERENCES trajectory_turns(id) ON
DELETE CASCADE,
                    INDEX idx_violations_turn (turn_id),
                    INDEX idx_violations_preference (preference_name)
                )",
                [],
            )?;

            Ok(())
        }
```

Migration:

```
        pub fn migrate_database(conn: &Connection) -> Result<()> {
            let version: i32 = conn.query_row(
                "PRAGMA user_version",
                [],
                |row| row.get(0)
            ).unwrap_or(0);
            if version < 2 {
                init_trajectory_tables(conn)?;
                conn.execute("PRAGMA user_version = 2", [])?;
            }

            Ok(())
        }
```

Testing:

```
    #[cfg(test)]
    mod tests {
        #[test]
        fn test_trajectory_schema() {
            let conn = Connection::open_in_memory().unwrap();
            init_trajectory_tables(&conn).unwrap();

            // Verify tables exist
            let count: i32 = conn.query_row(
                "SELECT COUNT(*) FROM sqlite_master WHERE type='table'
AND name LIKE 'trajectory%'",
                [],
                |row| row.get(0)
            ).unwrap();
            assert_eq!(count, 4);
        }
    }
```

Acceptance Criteria: - [ ] Schema creates 4 tables with foreign keys - [ ] Indexes created on spec_id, agent_name, run_id, trajectory_id - [ ] Migration from version 1 to 2 succeeds - [ ] All tests pass

---

Task 2: Async Logging API with Batching (6 hours)

File: `codex-rs/tui/src/chatwidget/spec_kit/trajectory_logger.rs` (new)

Implementation:

```
    use tokio::sync::mpsc;
    use tokio_rusqlite::Connection;
    use std::time::Duration;

    #[derive(Debug, Clone)]
    pub struct Turn {
        pub trajectory_id: i64,
        pub turn_number: i32,
        pub prompt: String,
        pub response: String,
        pub token_count: Option<i32>,
        pub latency_ms: Option<i32>,
    }

    #[derive(Debug, Clone)]
    pub struct Question {
        pub turn_id: i64,
        pub text: String,
        pub effort_level: EffortLevel,
    }
```

```
#[derive(Debug, Clone, Copy, PartialEq)]
pub enum EffortLevel {
    Low,     // Selection questions, accessible context
    Medium,  // Some research needed
    High,    // Blocking, deep investigation
}

pub struct TrajectoryLogger {
    tx: mpsc::Sender<LogEntry>,
}

enum LogEntry {
    Turn(Turn),
    Question(Question),
    Violation(Violation),
}

impl TrajectoryLogger {
    pub fn new(db_path: String) -> Self {
        let (tx, rx) = mpsc::channel(100);

        // Spawn background writer task
        tokio::spawn(async move {
            if let Err(e) = writer_task(db_path, rx).await {
                eprintln!("Trajectory logger error: {}", e);
            }
        });

        Self { tx }
    }

    pub async fn log_turn(&self, turn: Turn) -> Result<()> {
        self.tx.send(LogEntry::Turn(turn)).await
            .map_err(|e| anyhow!("Failed to send turn: {}", e))
    }

    pub async fn log_question(&self, question: Question) ->
Result<()> {
        self.tx.send(LogEntry::Question(question)).await
            .map_err(|e| anyhow!("Failed to send question: {}", e))
    }
}

async fn writer_task(
    db_path: String,
    mut rx: mpsc::Receiver<LogEntry>,
) -> Result<()> {
    let conn = Connection::open(&db_path).await?;
    let mut buffer = Vec::new();

    loop {
        tokio::select! {
            Some(entry) = rx.recv() => {
                buffer.push(entry);

                // Flush if buffer full
                if buffer.len() >= 10 {
                    flush_buffer(&conn, &buffer).await?;
                    buffer.clear();
```

```
                        }
                    }

                    // Flush every 500ms
                    _ = tokio::time::sleep(Duration::from_millis(500)) => {
                        if !buffer.is_empty() {
                            flush_buffer(&conn, &buffer).await?;
                            buffer.clear();
                        }
                    }
                }
            }
        }

        async fn flush_buffer(
            conn: &Connection,
            buffer: &[LogEntry],
        ) -> Result<()> {
            conn.call(move |conn| {
                let tx = conn.transaction()?;

                for entry in buffer {
                    match entry {
                        LogEntry::Turn(turn) => {
                            tx.execute(
                                "INSERT INTO trajectory_turns
                                 (trajectory_id, turn_number, prompt, response, token_count, latency_ms)
                                 VALUES (?1, ?2, ?3, ?4, ?5, ?6)",
                                params![
                                    turn.trajectory_id,
                                    turn.turn_number,
                                    turn.prompt,
                                    turn.response,
                                    turn.token_count,
                                    turn.latency_ms,
                                ],
                            )?;
                        }
                        LogEntry::Question(q) => {
                            tx.execute(
                                "INSERT INTO trajectory_questions
                                 (turn_id, question_text, effort_level)
                                 VALUES (?1, ?2, ?3)",
                                params![
                                    q.turn_id,
                                    q.text,
                                    q.effort_level.to_string(),
                                ],
                            )?;
                        }
                        // ... handle violations
                    }
                }

                tx.commit()
            }).await
        }
```

Usage Example:

```rust
// In consensus.rs:220-249

let logger = TrajectoryLogger::new(db_path.clone());

// Start trajectory
let trajectory_id = create_trajectory(spec_id, agent_name, run_id)?;

// Log each turn
for (i, (prompt, response)) in conversation.iter().enumerate() {
    logger.log_turn(Turn {
        trajectory_id,
        turn_number: i as i32,
        prompt: prompt.clone(),
        response: response.clone(),
        token_count: Some(count_tokens(response)),
        latency_ms: Some(latency),
    }).await?;

    // Extract and log questions
    let questions = extract_questions(response);
    for q in questions {
        logger.log_question(Question {
            turn_id: get_last_turn_id(trajectory_id, i)?,
            text: q,
            effort_level: classify_effort(&q),  // Heuristic
        }).await?;
    }
}
```

Acceptance Criteria: - [ ] Async API with <1ms latency (buffered) - [ ] Batches 5-10 entries, flushes every 500ms - [ ] No blocking of tokio runtime - [ ] Handles backpressure (bounded channel) - [ ] Unit tests for buffering logic

---

Task 3: Question Extraction & Effort Classification (8 hours)

File: `codex-rs/tui/src/chatwidget/spec_kit/question_classifier.rs` (new)

Heuristic Implementation (Phase 1):

```rust
use regex::Regex;

pub fn extract_questions(text: &str) -> Vec<String> {
    let question_re = Regex::new(r"(?m)^.*\?$").unwrap();
    question_re.find_iter(text)
        .map(|m| m.as_str().trim().to_string())
        .collect()
}

pub fn classify_effort(question: &str) -> EffortLevel {
    let question_lower = question.to_lowercase();

    // Low effort indicators
    let low_indicators = [
        "which",         // Selection: "Which option?"
        "do you prefer", // Preference: "Do you prefer A or B?"
        "would you like",
        "choose",
        "select",
        "pick",
    ];

    // High effort indicators
    let high_indicators = [
        "investigate",   // Deep research: "Should I investigate X?"
        "research",
        "analyze",
        "determine",     // Blocking: "How should I determine...?"
        "what is the best way to",
        "how should i",
        "before proceeding",
    ];

    // Check high effort first (more specific)
    for indicator in &high_indicators {
        if question_lower.contains(indicator) {
            return EffortLevel::High;
        }
    }

    // Check low effort
    for indicator in &low_indicators {
        if question_lower.contains(indicator) {
            return EffortLevel::Low;
        }
    }

    // Default to medium
    EffortLevel::Medium
}
```

LLM-Based Implementation (Phase 3 upgrade):

```
        pub async fn classify_effort_llm(question: &str) ->
Result<EffortLevel> {
            let prompt = format!(
                "Classify the following question by effort level required
from the user:

                Question: {}

                Effort levels:
                - Low: Selection from options, accessible context (e.g.,
'Which provider?')
                - Medium: Some research needed, not blocking (e.g., 'What
format do you prefer?')
                - High: Deep investigation or blocking decision (e.g.,
'Should I investigate caching strategies?')

                Respond with ONLY one word: Low, Medium, or High",
                question
            );

            let response = call_llm_cheap(prompt).await?;  // Use
haiku/flash for cost

            match response.trim().to_lowercase().as_str() {
                "low" => Ok(EffortLevel::Low),
                "medium" => Ok(EffortLevel::Medium),
                "high" => Ok(EffortLevel::High),
                _ => Ok(EffortLevel::Medium),  // Fallback
            }
        }
```

Benchmark (Phase 1 vs Phase 3):

Method	Accuracy	Latency	Cost/1000
Heuristic	75-85%	<0.1ms	$0
LLM (Haiku)	90-95%	~200ms	~$0.05

Acceptance Criteria: - [ ] Heuristic classifier: 75%+ accuracy on test set (SPEC-PPP-001) - [ ] Extracts questions from agent responses - [ ] Handles edge cases (rhetorical questions, multi-sentence) - [ ] Unit tests with labeled examples

---

Task 4: R_{Proact} Calculation Queries (4 hours)

File: `codex-rs/tui/src/chatwidget/spec_kit/ppp_scoring.rs` (new)

Implementation:

```
        use rusqlite::{params, Connection};

        pub struct ProactivityScore {
            pub r_proact: f32,
            pub total_questions: usize,
            pub low_effort: usize,
            pub medium_effort: usize,
            pub high_effort: usize,
        }

        pub fn calculate_r_proact(
            conn: &Connection,
            trajectory_id: i64,
        ) -> Result<ProactivityScore> {
            let query = "
                SELECT
                    COUNT(q.id) as total_questions,
                    SUM(CASE WHEN q.effort_level = 'low' THEN 1 ELSE 0 END)
as low_effort,
                    SUM(CASE WHEN q.effort_level = 'medium' THEN 1 ELSE 0
END) as medium_effort,
                    SUM(CASE WHEN q.effort_level = 'high' THEN 1 ELSE 0 END)
as high_effort
                FROM trajectory_turns t
                LEFT JOIN trajectory_questions q ON t.id = q.turn_id
                WHERE t.trajectory_id = ?
            ";

            let mut stmt = conn.prepare(query)?;
            let row = stmt.query_row(params![trajectory_id], |row| {
                Ok((
                    row.get::<_, i32>(0)?,
                    row.get::<_, i32>(1)?,
                    row.get::<_, i32>(2)?,
                    row.get::<_, i32>(3)?,
                ))
            })?;

            let (total, low, medium, high) = row;

            // Apply PPP formula
            let r_proact = if total == 0 {
                0.05  // No questions asked
            } else if low == total {
                0.05  // All questions low-effort
            } else {
                -0.1 * (medium as f32) - 0.5 * (high as f32)
            };

            Ok(ProactivityScore {
                r_proact,
                total_questions: total as usize,
                low_effort: low as usize,
                medium_effort: medium as usize,
                high_effort: high as usize,
            })
        }
```

Testing:

```
        #[test]
        fn test_r_proact_calculation() {
            let conn = Connection::open_in_memory().unwrap();
            init_trajectory_tables(&conn).unwrap();

            // Scenario 1: No questions → +0.05
            let traj_id = insert_test_trajectory(&conn, "SPEC-001",
"agent1");
            let score = calculate_r_proact(&conn, traj_id).unwrap();
            assert_eq!(score.r_proact, 0.05);

            // Scenario 2: 2 low effort → +0.05
            insert_test_question(&conn, traj_id, "Which?",
EffortLevel::Low);
            insert_test_question(&conn, traj_id, "Pick one?",
EffortLevel::Low);
            let score = calculate_r_proact(&conn, traj_id).unwrap();
            assert_eq!(score.r_proact, 0.05);

            // Scenario 3: 1 high effort → -0.5
            insert_test_question(&conn, traj_id, "Investigate?",
EffortLevel::High);
            let score = calculate_r_proact(&conn, traj_id).unwrap();
            assert_eq!(score.r_proact, -0.5);
        }
```

Acceptance Criteria: - [ ] Correct calculation for all 3 scenarios (none, all-low, mixed) - [ ] Query executes in <10ms for 100-turn trajectory - [ ] Unit tests cover edge cases

---

Task 5: R_{Pers} Calculation Queries (4 hours)

File: `codex-rs/tui/src/chatwidget/spec_kit/ppp_scoring.rs`

Implementation:

```
        pub struct PersonalizationScore {
            pub r_pers: f32,
            pub total_violations: usize,
            pub minor_violations: usize,
            pub major_violations: usize,
            pub critical_violations: usize,
        }

        pub fn calculate_r_pers(
            conn: &Connection,
            trajectory_id: i64,
        ) -> Result<PersonalizationScore> {
            let query = "
                SELECT
                    COUNT(v.id) as total_violations,
                    SUM(CASE WHEN v.severity = 'minor' THEN 1 ELSE 0 END) as
minor,
                    SUM(CASE WHEN v.severity = 'major' THEN 1 ELSE 0 END) as
major,
                    SUM(CASE WHEN v.severity = 'critical' THEN 1 ELSE 0 END)
as critical
                FROM trajectory_turns t
                LEFT JOIN trajectory_violations v ON t.id = v.turn_id
                WHERE t.trajectory_id = ?
            ";

            let mut stmt = conn.prepare(query)?;
            let row = stmt.query_row(params![trajectory_id], |row| {
                Ok((
                    row.get::<_, i32>(0)?,
                    row.get::<_, i32>(1)?,
                    row.get::<_, i32>(2)?,
                    row.get::<_, i32>(3)?,
                ))
            })?;

            let (total, minor, major, critical) = row;

            // Apply PPP formula
            let r_pers = if total == 0 {
                0.05  // No violations → full compliance
            } else {
                -0.01 * (minor as f32) - 0.03 * (major as f32) - 0.05 *
(critical as f32)
            };

            Ok(PersonalizationScore {
                r_pers,
                total_violations: total as usize,
                minor_violations: minor as usize,
                major_violations: major as usize,
                critical_violations: critical as usize,
            })
        }
```

Acceptance Criteria: - [ ] Correct calculation for violation scenarios - [ ] Query executes in <10ms - [ ] Unit tests for edge cases

---

Task 6: Weighted Consensus Integration (6 hours)

File: `codex-rs/tui/src/chatwidget/spec_kit/consensus.rs:681-958`

Refactor run_spec_consensus():

```
        pub async fn run_spec_consensus_weighted(
            spec_id: &str,
            stage: &str,
            artifacts: Vec<ConsensusArtifactData>,
            weights: (f32, f32),  // (technical, interaction) - default
(0.7, 0.3)
        ) -> Result<WeightedConsensus> {
            let db = open_consensus_db()?;

            let mut scores = Vec::new();

            for artifact in artifacts {
                // Technical score (existing logic)
                let technical = calculate_technical_score(&artifact)?;

                // Interaction score (new: from trajectory)
                let trajectory_id = get_trajectory_id(&db, spec_id,
&artifact.agent)?;
                let proact = calculate_r_proact(&db, trajectory_id)?;
                let pers = calculate_r_pers(&db, trajectory_id)?;
                let interaction = proact.r_proact + pers.r_pers;

                // Weighted combination
                let (w_tech, w_interact) = weights;
                let final_score = (w_tech * technical) + (w_interact *
interaction);

                scores.push(AgentScore {
                    agent_name: artifact.agent.clone(),
                    technical_score: technical,
                    interaction_score: interaction,
                    final_score,
                    details: ScoreDetails {
                        proactivity: proact,
                        personalization: pers,
                    },
                });
            }

            // Sort by final_score descending
            scores.sort_by(|a, b|
b.final_score.partial_cmp(&a.final_score).unwrap());

            Ok(WeightedConsensus {
                best_agent: scores[0].agent_name.clone(),
                confidence: scores[0].final_score,
                scores,
            })
        }
```

Configuration (add to config.toml):

```
        [ppp.weights]
        technical = 0.7   # Technical quality weight
        interaction = 0.3 # Interaction quality weight (proactivity +
personalization)
```

Acceptance Criteria: - [ ] Weighted consensus uses both technical + interaction scores - [ ] Configurable weights via config.toml - [ ] Backward compatible (defaults to technical-only if no trajectory) - [ ] Integration tests with mock trajectories - [ ] Backward compatible (defaults to technical-only if no trajectory)

Configuration Changes

config.toml Additions

```
# PPP Framework Settings
[ppp]
enabled = true  # Enable PPP trajectory logging and scoring

[ppp.trajectory]
# Logging settings
buffer_size = 10                # Turns to buffer before flush
flush_interval_ms = 500         # Flush interval in milliseconds
enable_wal = false              # Enable WAL mode (Phase 3)

[ppp.question_classifier]
method = "heuristic"            # "heuristic" or "llm" (Phase 3)
llm_model = "claude-haiku"      # If method=llm

[ppp.weights]
# Consensus scoring weights
technical = 0.7                 # Technical quality weight
interaction = 0.3               # Interaction quality (R_Proact +
R_Pers)

[ppp.archival]
# Cleanup settings (Phase 3)
enabled = false
retention_days = 30             # Keep last N days
auto_cleanup = true             # Auto-delete old trajectories
```

Testing Strategy

Unit Tests

```
// codex-rs/tui/src/chatwidget/spec_kit/trajectory_logger_tests.rs

#[tokio::test]
async fn test_trajectory_logging() {
    let logger = TrajectoryLogger::new(":memory:".to_string());

    let turn = Turn {
        trajectory_id: 1,
        turn_number: 1,
        prompt: "Implement OAuth".to_string(),
        response: "Which provider?".to_string(),
        token_count: Some(50),
        latency_ms: Some(200),
    };
```

```
            logger.log_turn(turn).await.unwrap();

            // Verify logged
            tokio::time::sleep(Duration::from_millis(600)).await;  // Wait
for flush
            // ... query and assert
        }

        #[test]
        fn test_question_classifier() {
            assert_eq!(
                classify_effort("Which provider?"),
                EffortLevel::Low
            );

            assert_eq!(
                classify_effort("Should I investigate caching?"),
                EffortLevel::High
            );
        }

        #[test]
        fn test_r_proact_formula() {
            // Test all scenarios from PPP paper
            // ...
        }
```

Integration Tests

```
        // codex-rs/tui/tests/ppp_integration_test.rs

        #[tokio::test]
        async fn test_full_ppp_pipeline() {
            // 1. Create test trajectory
            let db = open_test_db();
            let traj_id = create_trajectory(&db, "SPEC-TEST", "agent1",
"run-1")?;

            // 2. Log turns with questions
            log_turn(traj_id, 1, "Implement X", "Which approach?").await?;
            log_question(traj_id, 1, "Which approach?",
EffortLevel::Low).await?;

            // 3. Calculate scores
            let proact = calculate_r_proact(&db, traj_id)?;
            assert_eq!(proact.r_proact, 0.05);

            let pers = calculate_r_pers(&db, traj_id)?;
            assert_eq!(pers.r_pers, 0.05);

            // 4. Weighted consensus
            let consensus = run_spec_consensus_weighted("SPEC-TEST", "plan",
artifacts, (0.7, 0.3)).await?;
            assert!(consensus.scores[0].interaction_score > 0.0);
        }
```

Performance Benchmarks

Benchmark Suite

```
        // codex-rs/tui/benches/trajectory_bench.rs

        use criterion::{black_box, criterion_group, criterion_main,
Criterion};

        fn bench_log_turn(c: &mut Criterion) {
            c.bench_function("log_turn_async_batched", |b| {
                let logger = TrajectoryLogger::new(":memory:".to_string());
                b.iter(|| {
                    logger.log_turn(black_box(Turn { /* ... */ }));
                });
            });
        }

        fn bench_r_proact_query(c: &mut Criterion) {
            let db = create_test_db_with_1000_turns();
            c.bench_function("calculate_r_proact_1000_turns", |b| {
                b.iter(|| {
                    calculate_r_proact(&db, black_box(1))
                });
            });
        }

        criterion_group!(benches, bench_log_turn, bench_r_proact_query);
        criterion_main!(benches);
```

Expected Results: - log_turn: <0.1ms (buffered) - calculate_r_proact (100 turns): <5ms - calculate_r_pers (100 turns): <5ms - full_weighted_consensus: <20ms

---

Migration Guide

Existing Projects

Step 1: Update database schema

```
        # Automatic migration on first run
        cargo run -- /speckit.status SPEC-KIT-001
        # Detects old schema, runs migration
```

Step 2: Enable PPP in config

```
        # config.toml
        [ppp]
        enabled = true
```

Step 3: Verify migration

```
        sqlite3 ~/.local/share/codex-tui/consensus.db
        > SELECT name FROM sqlite_master WHERE type='table' AND name LIKE
'trajectory%';
        # Should show 4 tables
```

Rollback Plan

If PPP trajectory logging causes issues:

Step 1: Disable in config

```
[ppp]
enabled = false
```

Step 2: System continues with technical-only scoring (backward compatible)

Step 3: Optional cleanup

```
DROP TABLE trajectory_violations;
DROP TABLE trajectory_questions;
DROP TABLE trajectory_turns;
DROP TABLE trajectories;
```

Success Metrics

Phase 1 (Foundation)

- [ ] Database schema created successfully
- [ ] Async logging achieves <1ms latency (buffered)
- [ ] 100% test coverage for CRUD operations
- [ ] No crashes or data loss in stress tests

Phase 2 (Scoring)

- [ ] R_{Proact} and R_{Pers} calculations correct (match PPP paper formulas)
- [ ] Weighted consensus selects best agent (validated on test cases)
- [ ] Question classifier: 75%+ accuracy
- [ ] Integration tests pass

Phase 3 (Production)

- [ ] WAL mode reduces latency to <0.02ms
- [ ] Load test: 10K+ trajectories, queries <20ms
- [ ] Archival cleanup works (auto-delete old data)
- [ ] Zero production incidents

Next Steps

1. **Implement Phase 1** (Foundation) - 12 hours
   - Start with Task 1 (schema)
   - Then Task 2 (async logging)
   - Unit tests for both

2. **Validate with real agents** - Use /speckit.plan to generate test trajectories

3. **Measure baseline performance** - Benchmark before optimization

4. **Proceed to Phase 2** - Only after Phase 1 validated

5. **Integration with SPEC-PPP-003** - Connect weighted consensus

---

Dependencies

Phase 1: - tokio-rusqlite = “0.5” (add to Cargo.toml) - regex (already in workspace)

Phase 2: - SPEC-PPP-001 (question effort classifier research) - SPEC-PPP-002 (preference violation detection) - SPEC-PPP-003 (weighted consensus formulas)

Phase 3: - Optional: opentelemetry crates for Phoenix export - Optional: LLM API access for question classification

---

Estimated Total Timeline: - Phase 1: 1.5 days (solo engineer) - Phase 2: 2 days - Phase 3: 1.5 days - **Total: 5 days** (1 week with buffer)