recommendations

SPEC-PPP-003: Implementation Recommendations

Last Updated: 2025-11-16 **Target**: Weighted consensus combining technical quality + interaction quality **Total Effort**: ~16 hours (2 days, 1 engineer)

Executive Summary

Implement PPP weighted consensus by refactoring existing `consensus.rs` to combine technical quality (70%) with interaction quality (30%) when selecting the best agent output. Use standard ML ensemble weighted averaging technique with configurable weights.

Key Decisions: 1. ✅ Refactor consensus.rs (NOT new module) - Extends existing infrastructure 2. ✅ 70/30 weights (technical/interaction) - Domain expert selection 3. ✅ Linear weighted average - Standard ML ensemble technique 4. ✅ User-configurable weights - Via config.toml (Phase 2) 5. ✅ Fallback to technical-only - If no trajectory available (backward compatible)

Phased Rollout

Phase 1: Core Weighted Consensus (8 hours, 1 day)

Goal: Basic weighted scoring with fixed 70/30 weights

Deliverables: - ✅ Refactor `consensus.rs:681-958` to add weighted scoring - ✅ Integration with SPEC-PPP-004 trajectory logging - ✅ Unit tests (weighted formula, edge cases) - ✅ Backward compatibility (fallback to technical-only)

Capabilities: - Score agents on technical (70%) + interaction (30%) - Select best agent based on weighted score - Handle missing trajectories gracefully

NOT Included (defer to Phase 2): - User-configurable weights - Stage-specific weights - Weight optimization/tuning

---

Phase 2: Configurability & Tuning (6 hours, 0.75 days)

Goal: User control and stage-specific weights

Deliverables: - ✅ Configuration via config.toml - ✅ Stage-specific weights (plan: 60/40, unlock: 80/20) - ✅ Validation (weights must sum to 1.0) - ✅ Integration tests (different weight scenarios)

Capabilities: - Users can override default 70/30 weights - Different weights per spec-kit stage - Config validation with clear error messages

NOT Included (defer to Phase 3): - Dynamic weight adaptation - Weight optimization from historical data

---

Phase 3: Advanced Optimization (Optional, 12 hours, 1.5 days)

Goal: Data-driven weight tuning

Deliverables: - ✅ Historical regret tracking (selected vs best) - ✅ Grid search for optimal weights - ✅ Inverse error weighting (adaptive) - ✅ A/B testing framework

Capabilities: - Measure weight effectiveness - Automatic weight tuning based on outcomes - Per-user weight preferences

---

Detailed Task Breakdown

Task 1: Refactor consensus.rs for Weighted Scoring (4 hours)

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

Current Implementation:

```
pub async fn run_spec_consensus(
    spec_id: &str,
    stage: &str,
    artifacts: Vec<ConsensusArtifactData>,
) -> Result<ConsensusResult> {
    // Current: Select based on technical score only
    let best = artifacts.iter()
        .max_by_key(|a| calculate_technical_score(a))
        .unwrap();

    Ok(ConsensusResult {
        selected_agent: best.agent.clone(),
        selected_content: best.content.clone(),
    })
}
```

Proposed Implementation:

```
use crate::chatwidget::spec_kit::ppp_scoring::{calculate_r_proact,
calculate_r_pers};
use crate::chatwidget::spec_kit::consensus_db::open_consensus_db;

pub struct AgentScore {
    pub agent_name: String,
            pub technical_score: f32,
            pub interaction_score: f32,
            pub final_score: f32,
            pub details: ScoreDetails,
        }

        pub struct ScoreDetails {
            pub proactivity: ProactivityScore,  // From SPEC-PPP-004
            pub personalization: PersonalizationScore,
        }

        pub struct WeightedConsensus {
            pub best_agent: String,
            pub confidence: f32,
            pub scores: Vec<AgentScore>,
        }

        pub async fn run_spec_consensus_weighted(
            spec_id: &str,
            stage: &str,
            artifacts: Vec<ConsensusArtifactData>,
            weights: Option<(f32, f32)>,  // (technical, interaction),
defaults to (0.7, 0.3)
        ) -> Result<WeightedConsensus> {
            let (w_tech, w_interact) = weights.unwrap_or((0.7, 0.3));

            // Validate weights
            if (w_tech + w_interact - 1.0).abs() > 0.001 {
                return Err(anyhow!("Weights must sum to 1.0, got {} + {} =
{}",

                    w_tech, w_interact, w_tech + w_interact));
            }

            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 interaction = if let Ok(traj_id) =
get_trajectory_id(&db, spec_id, &artifact.agent) {
                    let proact = calculate_r_proact(&db, traj_id)?;
                    let pers = calculate_r_pers(&db, traj_id)?;

                    let score = proact.r_proact + pers.r_pers;

                    scores.push(AgentScore {
                        agent_name: artifact.agent.clone(),
                        technical_score: technical,
                        interaction_score: score,
                        final_score: (w_tech * technical) + (w_interact *
score),
                        details: ScoreDetails {
                            proactivity: proact,
                            personalization: pers,
                        },
                    });
                    score
                } else {
                    // Fallback: No trajectory available, use technical only
                    warn!("No trajectory found for agent {}, using technical
score only", artifact.agent);

                    scores.push(AgentScore {
                        agent_name: artifact.agent.clone(),
                        technical_score: technical,
                        interaction_score: 0.0,
                        final_score: technical,  // 100% technical (backward
compatible)
                        details: ScoreDetails {
                            proactivity: ProactivityScore::default(),
                            personalization:
PersonalizationScore::default(),
                        },
                    });

                    0.0
                };
            }

            // 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,
            })
        }
```

Integration Points:

```
        // In consensus.rs, update public API
        pub async fn run_spec_consensus(
            spec_id: &str,
            stage: &str,
            artifacts: Vec<ConsensusArtifactData>,
        ) -> Result<ConsensusResult> {
            // Delegate to weighted consensus
            let weighted = run_spec_consensus_weighted(spec_id, stage,
artifacts, None).await?;

            // Convert to old format for backward compatibility
            Ok(ConsensusResult {
                selected_agent: weighted.best_agent,
                selected_content: get_artifact_content(&weighted.best_agent,
&artifacts)?,
            })
        }
```

Acceptance Criteria: - [ ] Weighted consensus selects best agent (validated with test cases) - [ ] Fallback to technical-only if no trajectory - [ ] Weights validated (must sum to 1.0) - [ ] Backward compatible (existing code still works)

Task 2: Configuration Support (2 hours)

File: `config.toml.example`

Add PPP Weights Section:

```
        # PPP Framework - Weighted Consensus
        [ppp.weights]
        # Global default weights
        technical = 0.7      # Technical quality weight (correctness,
completeness)
        interaction = 0.3    # Interaction quality weight (proactivity +
personalization)

        # Stage-specific weights (Phase 2)
        [ppp.weights.plan]
        technical = 0.6      # Planning: Exploration phase, interaction
matters more
        interaction = 0.4

        [ppp.weights.tasks]
        technical = 0.7      # Tasks: Balanced
        interaction = 0.3

        [ppp.weights.implement]
        technical = 0.7      # Implementation: Balanced
        interaction = 0.3

        [ppp.weights.validate]
        technical = 0.75     # Validation: Correctness important
        interaction = 0.25

        [ppp.weights.audit]
        technical = 0.8      # Audit: Security/compliance critical
        interaction = 0.2

        [ppp.weights.unlock]
        technical = 0.8      # Unlock: Final validation, correctness
critical
        interaction = 0.2
```

Configuration Loading:

```
        // codex-rs/core/src/config_types.rs

        #[derive(Deserialize, Serialize, Debug, Clone)]
        pub struct PppWeights {
            #[serde(default = "default_technical_weight")]
            pub technical: f32,

            #[serde(default = "default_interaction_weight")]
            pub interaction: f32,

            // Stage-specific overrides (Phase 2)
            #[serde(default)]
            pub plan: Option<StageWeights>,

            #[serde(default)]
            pub tasks: Option<StageWeights>,

            #[serde(default)]
```

```rust
    pub implement: Option<StageWeights>,

    #[serde(default)]
    pub validate: Option<StageWeights>,

    #[serde(default)]
    pub audit: Option<StageWeights>,

    #[serde(default)]
    pub unlock: Option<StageWeights>,
}

#[derive(Deserialize, Serialize, Debug, Clone)]
pub struct StageWeights {
    pub technical: f32,
    pub interaction: f32,
}

fn default_technical_weight() -> f32 { 0.7 }
fn default_interaction_weight() -> f32 { 0.3 }

impl PppWeights {
    pub fn validate(&self) -> Result<()> {
        let sum = self.technical + self.interaction;
        if (sum - 1.0).abs() > 0.001 {
            return Err(anyhow!(
                "PPP weights must sum to 1.0, got technical={} +
interaction={} = {}",
                self.technical, self.interaction, sum
            ));
        }

        // Validate stage-specific weights (Phase 2)
        for (stage, weights) in [
            ("plan", &self.plan),
            ("tasks", &self.tasks),
            ("implement", &self.implement),
            ("validate", &self.validate),
            ("audit", &self.audit),
            ("unlock", &self.unlock),
        ] {
            if let Some(w) = weights {
                let sum = w.technical + w.interaction;
                if (sum - 1.0).abs() > 0.001 {
                    return Err(anyhow!(
                        "PPP weights.{} must sum to 1.0, got {}",
                        stage, sum
                    ));
                }
            }
        }

        Ok(())
    }

    pub fn get_weights_for_stage(&self, stage: &str) -> (f32, f32) {
        let stage_weights = match stage {
            "plan" => &self.plan,
            "tasks" => &self.tasks,
            "implement" => &self.implement,
            "validate" => &self.validate,
            "audit" => &self.audit,
            "unlock" => &self.unlock,
            _ => &None,
        };

        stage_weights
            .as_ref()
            .map(|w| (w.technical, w.interaction))
            .unwrap_or((self.technical, self.interaction))
    }
}

// Add to AgentConfig
#[derive(Deserialize, Serialize, Debug, Clone)]
pub struct AgentConfig {
    // ... existing fields

    #[serde(default)]
    pub ppp: Option<PppConfig>,
}

#[derive(Deserialize, Serialize, Debug, Clone)]
pub struct PppConfig {
    #[serde(default)]
    pub enabled: bool,

    #[serde(default)]
    pub weights: PppWeights,

    #[serde(default)]
    pub trajectory: TrajectoryConfig,  // From SPEC-PPP-004
}
```

Usage in consensus.rs:

```
    pub async fn run_spec_consensus_with_config(
        spec_id: &str,
        stage: &str,
        artifacts: Vec<ConsensusArtifactData>,
        config: &AgentConfig,
    ) -> Result<WeightedConsensus> {
        let weights = if let Some(ppp) = &config.ppp {
            ppp.weights.get_weights_for_stage(stage)
        } else {
            (0.7, 0.3)  // Default
        };

        run_spec_consensus_weighted(spec_id, stage, artifacts,
Some(weights)).await
    }
```

Acceptance Criteria: - [ ] Config loads with defaults (70/30) - [ ] User can override via config.toml - [ ] Stage-specific weights work (Phase 2) - [ ] Validation catches invalid weights (sum ≠ 1.0)

Task 3: Integration Tests (4 hours)

File: `codex-rs/tui/tests/ppp_weighted_consensus_test.rs`

Test Scenarios:

```
use codex_tui::chatwidget::spec_kit::consensus::*;

#[tokio::test]
async fn test_weighted_consensus_basic() {
    // Setup: 3 agents with different technical + interaction scores
    let artifacts = vec![
        create_test_artifact("agent1", 0.85, 0.10),   // Balanced
        create_test_artifact("agent2", 0.95, -0.45),  // High tech,
poor interaction
        create_test_artifact("agent3", 0.80, 0.02),   // Medium
tech, ok interaction
    ];

    let result = run_spec_consensus_weighted(
        "SPEC-TEST-001",
        "implement",
        artifacts,
        Some((0.7, 0.3)),
    ).await.unwrap();

    // Agent1 should win: 0.7*0.85 + 0.3*0.10 = 0.625
    assert_eq!(result.best_agent, "agent1");
    assert_eq!(result.scores.len(), 3);
    assert!((result.confidence - 0.625).abs() < 0.01);
}

#[tokio::test]
async fn test_fallback_no_trajectory() {
    // Setup: Agent with no trajectory (no interaction score)
    let artifacts = vec![
        create_test_artifact_no_trajectory("agent1", 0.90),
    ];

    let result = run_spec_consensus_weighted(
        "SPEC-TEST-002",
        "plan",
        artifacts,
        Some((0.7, 0.3)),
    ).await.unwrap();

    // Should use technical score only (100%)
    assert_eq!(result.best_agent, "agent1");
    assert_eq!(result.scores[0].final_score, 0.90);
    assert_eq!(result.scores[0].interaction_score, 0.0);
}

#[tokio::test]
async fn test_stage_specific_weights() {
    let config = load_test_config_with_stage_weights();
    let artifacts = vec![
        create_test_artifact("agent1", 0.80, 0.05),
        create_test_artifact("agent2", 0.70, 0.10),
    ];

    // Plan stage: 60/40 (interaction matters more)
    let result = run_spec_consensus_with_config(
        "SPEC-TEST-003",
        "plan",
                artifacts.clone(),
                &config,
            ).await.unwrap();

            // Agent2 should win in plan: 0.6*0.70 + 0.4*0.10 = 0.46
            // vs Agent1: 0.6*0.80 + 0.4*0.05 = 0.50
            assert_eq!(result.best_agent, "agent1");

            // Unlock stage: 80/20 (technical matters more)
            let result = run_spec_consensus_with_config(
                "SPEC-TEST-003",
                "unlock",
                artifacts.clone(),
                &config,
            ).await.unwrap();

            // Agent1 should still win: 0.8*0.80 + 0.2*0.05 = 0.65
            assert_eq!(result.best_agent, "agent1");
        }

        #[tokio::test]
        async fn test_weight_validation() {
            let artifacts = vec![create_test_artifact("agent1", 0.8, 0.1)];

            // Invalid: weights don't sum to 1.0
            let result = run_spec_consensus_weighted(
                "SPEC-TEST-004",
                "plan",
                artifacts,
                Some((0.5, 0.6)),  // Sum = 1.1
            ).await;

            assert!(result.is_err());
            assert!(result.unwrap_err().to_string().contains("must sum to
1.0"));
        }

        #[tokio::test]
        async fn test_equal_scores_tiebreaker() {
            // Setup: 2 agents with identical final scores
            let artifacts = vec![
                create_test_artifact("agent1", 0.80, 0.05),  // 0.7*0.8 +
0.3*0.05 = 0.575
                create_test_artifact("agent2", 0.75, 0.10),  // 0.7*0.75 +
0.3*0.10 = 0.555
            ];

            let result = run_spec_consensus_weighted(
                "SPEC-TEST-005",
                "implement",
                artifacts,
                Some((0.7, 0.3)),
            ).await.unwrap();

            // Agent1 should win (higher score)
            assert_eq!(result.best_agent, "agent1");
        }
```

Helper Functions:

```
        fn create_test_artifact(agent: &str, technical: f32, interaction:
```

```
f32) -> ConsensusArtifactData {
            // Create artifact with pre-computed scores
            // Insert trajectory with calculated R_Proact + R_Pers =
interaction
            // ...
        }

        fn create_test_artifact_no_trajectory(agent: &str, technical: f32) -
> ConsensusArtifactData {
            // Create artifact without trajectory (test fallback)
            // ...
        }
```

Acceptance Criteria: - [ ] All test scenarios pass - [ ] Weighted formula validated (spot checks) - [ ] Fallback behavior tested - [ ] Stage-specific weights tested

---

Task 4: Documentation & Examples (2 hours)

File: `docs/ppp-weighted-consensus-guide.md`

Contents:

```
        # PPP Weighted Consensus Guide

        ## Overview

        PPP weighted consensus combines **technical quality** (correctness,
completeness) with **interaction quality** (proactivity,
personalization) when selecting the best agent output.

        ## Formula
```

final_score = (w_technical × technical_score) + (w_interaction × interaction_score)

Default weights: - w_technical = 0.7 (70%) - w_interaction = 0.3 (30%)

~~~
## Configuration

### Global Weights

```toml
[ppp.weights]
technical = 0.7
interaction = 0.3
~~~

Stage-Specific Weights

```
        [ppp.weights.plan]
        technical = 0.6   # Exploration phase
        interaction = 0.4

        [ppp.weights.unlock]
        technical = 0.8   # Correctness critical
        interaction = 0.2
```

Examples

Example 1: Balanced Agent Wins

```
Agent 1:
  Technical: 0.85 (good code quality)
  Interaction: 0.10 (asked 2 low-effort questions)
  Final: 0.7 × 0.85 + 0.3 × 0.10 = 0.625 ← Winner

Agent 2:
  Technical: 0.95 (excellent code quality)
  Interaction: -0.45 (asked 1 high-effort blocking question)
  Final: 0.7 × 0.95 + 0.3 × (-0.45) = 0.530
```

Winner: Agent 1 - Better overall balance

Example 2: Technical Quality Dominates

```
Weights: 80/20 (unlock stage)

Agent 1:
  Technical: 0.95
  Interaction: -0.20
  Final: 0.8 × 0.95 + 0.2 × (-0.20) = 0.72 ← Winner

Agent 2:
  Technical: 0.85
  Interaction: 0.05
  Final: 0.8 × 0.85 + 0.2 × 0.05 = 0.69
```

Winner: Agent 1 - Technical quality prioritized

Tuning Weights

When to Increase Technical Weight (>0.7)

- Critical stages (audit, unlock)
- Production deployments
- Security-sensitive code
- Complex algorithms

When to Increase Interaction Weight (>0.3)

- Early stages (plan, tasks)
- Prototyping/exploration
- Beginner users
- Interactive workflows

Finding Your Ideal Weights

1. Start with defaults (70/30)
2. Run 10-20 spec-kit executions
3. Review selected agents vs your preference
4. Adjust weights in 0.05 increments
5. Repeat until satisfied

~~~
**Acceptance Criteria**:
- [ ] Guide explains formula clearly
- [ ] Examples demonstrate use cases
- [ ] Tuning guidance provided
- [ ] Links to full SPEC-PPP-003 research

---

## Performance Validation

### Benchmark Targets

| Operation | Target | Measurement |
|-----------|--------|-------------|
| **Calculate Interaction Score** | <20ms | 2 SQL queries (R_Proact
+ R_Pers) |
| **Calculate Final Score** | <1ms | Weighted average (arithmetic) |
| **Select Best Agent** | <5ms | Sort by score |
| **Total Consensus Overhead** | **<50ms** | All operations combined
|

### Test Queries

```sql
-- Test 1: R_Proact with 100 turns, 50 questions
SELECT ... FROM trajectory_turns t LEFT JOIN trajectory_questions q
...;
-- Expected: <10ms

-- Test 2: R_Pers with 100 turns, 10 violations
SELECT ... FROM trajectory_turns t LEFT JOIN trajectory_violations v
...;
-- Expected: <10ms

-- Test 3: Sort 5 agents by final score
SELECT * FROM agent_scores ORDER BY final_score DESC;
-- Expected: <1ms
~~~

Phase 2 Enhancements

User-Configurable Weights

UI Addition (optional, codex-tui):

```
┌─ PPP Consensus Settings ─────────────────┐
│ Technical Weight:   [0.70] (0.0 - 1.0)  │
│ Interaction Weight: [0.30] (auto-calc)  │
│                                          │
│ Presets:                                 │
│ [ ] Balanced (70/30)                     │
│ [ ] Technical Focus (80/20)              │
│ [ ] Interaction Focus (60/40)            │
│                                          │
│ [Apply]  [Reset to Defaults]             │
└──────────────────────────────────────────┘
```

Stage-Specific Weight Recommendations

Based on task criticality research:

Stage	Technical	Interaction	Rationale
plan	0.6	0.4	Exploration, questions OK
tasks	0.7	0.3	Balanced
implement	0.7	0.3	Balanced
validate	0.75	0.25	Testing correctness important
audit	0.8	0.2	Security/compliance critical
unlock	0.8	0.2	Final validation

Phase 3 Advanced Features

Regret Tracking

Metric: Regret = score(best) - score(selected)

Implementation:

```rust
pub struct ConsensusRegret {
    pub selected_agent: String,
    pub selected_score: f32,
    pub best_agent: String,  // In retrospect (user feedback)
    pub best_score: f32,
    pub regret: f32,
}

pub fn track_regret(
    consensus: &WeightedConsensus,
    user_preferred: &str,
) -> ConsensusRegret {
    let selected_score = consensus.confidence;
    let best_score = consensus.scores.iter()
        .find(|s| s.agent_name == user_preferred)
        .map(|s| s.final_score)
        .unwrap_or(0.0);

    ConsensusRegret {
        selected_agent: consensus.best_agent.clone(),
        selected_score,
        best_agent: user_preferred.to_string(),
        best_score,
        regret: best_score - selected_score,
    }
}
```

Usage: Collect regret over 100+ runs, optimize weights to minimize average regret.

Grid Search Weight Optimization

Pseudocode:

```
        best_weights = (0.7, 0.3)
        min_regret = float('inf')

        for w_tech in [0.5, 0.55, 0.6, ..., 0.9]:
            w_interact = 1.0 - w_tech

            total_regret = 0
            for run in historical_runs:
                consensus = run_weighted_consensus(run.artifacts, (w_tech,
w_interact))
                regret = calculate_regret(consensus, run.user_preferred)
                total_regret += regret

            avg_regret = total_regret / len(historical_runs)

            if avg_regret < min_regret:
                min_regret = avg_regret
                best_weights = (w_tech, w_interact)

        print(f"Optimal weights: {best_weights} (regret: {min_regret})")
```

Expected Result: Optimal weights within ±0.05 of 70/30 for most users.

Success Metrics

Phase 1 (Core)

- [ ] Weighted consensus selects best agent (validated on test cases)
- [ ] <50ms overhead (measured via benchmarks)
- [ ] 100% backward compatible (existing code works)
- [ ] Zero production incidents

Phase 2 (Configuration)

- [ ] Users can configure weights via config.toml
- [ ] Stage-specific weights applied correctly
- [ ] Config validation prevents invalid weights
- [ ] Documentation guides users to optimal weights

Phase 3 (Optimization)

- [ ] Regret tracking measures effectiveness
- [ ] Grid search finds optimal weights per user
- [ ] Average regret <0.05 (near-optimal selection)

Dependencies

Phase 1: - SPEC-PPP-004 (trajectory logging) - Must be implemented first - SPEC-PPP-002 (preference violations) - For R_Pers calculation - consensus.rs (existing) - Refactor target

Phase 2: - config_types.rs (existing) - Extend with PppWeights

Phase 3: - Historical consensus data (100+ runs) - User feedback mechanism (which agent was best?)

Estimated Total Timeline: - Phase 1: 1 day (solo engineer) - Phase 2: 0.75 days - Phase 3: 1.5 days (optional) - **Total: 2-3 days** (core + configuration)