

# ADR-003-003-refactor-consensus-vs-new-module

## ADR-003-003: Refactor consensus.rs vs New Module for Weighted Consensus

**Status:** Accepted **Date:** 2025-11-16 **Deciders:** Research Team  
**Related:** SPEC-PPP-003 (Interaction Scoring & Weighted Consensus)

---

### Context

We need to implement PPP weighted consensus (technical + interaction scoring) in the codex-tui codebase. The existing `consensus.rs` module (lines 681-958) implements technical-only consensus.

**Question:** Should we refactor the existing `consensus.rs` or create a new module?

Options considered: 1. **Refactor consensus.rs** - Extend existing `run_spec_consensus()` function 2. **New module (weighted\_consensus.rs)** - Separate file for PPP logic 3. **New crate** - Extract consensus to separate library 4. **Trait-based** - Define Consensus trait, multiple implementations

---

### Decision

We will **refactor consensus.rs** to add weighted consensus functionality while maintaining backward compatibility.

**Approach:** - Add new function: `run_spec_consensus_weighted()` - Refactor existing `run_spec_consensus()` to delegate to weighted version - Keep all code in single file (`consensus.rs`)

---

### Rationale

#### 1. Code Reuse & Integration

**Existing consensus.rs** has infrastructure we need: - Database connection management - Artifact validation - Error handling - Logging/telemetry

**Refactoring** (reuses 80% of code):

```

// consensus.rs (refactored)

pub async fn run_spec_consensus(
    spec_id: &str,
    stage: &str,
    artifacts: Vec<ConsensusArtifactData>,
) -> Result<ConsensusResult> {
    // Delegate to weighted version with default weights
    let weighted = run_spec_consensus_weighted(
        spec_id,
        stage,
        artifacts,
        Some((0.7, 0.3)), // Default
    ).await?;

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

pub async fn run_spec_consensus_weighted(
    spec_id: &str,
    stage: &str,
    artifacts: Vec<ConsensusArtifactData>,
    weights: Option<(f32, f32)>,
) -> Result<WeightedConsensus> {
    // NEW: Weighted scoring logic
    // REUSE: DB access, validation, error handling from existing
code
}

```

**New module** (duplicates 80% of code):

```

// consensus.rs (unchanged)
pub async fn run_spec_consensus(...) -> Result<ConsensusResult> {
    // ... existing 300 lines
}

// weighted_consensus.rs (new)
pub async fn run_weighted_consensus(...) ->
Result<WeightedConsensus> {
    // DUPLICATE: DB access, validation, error handling
    // NEW: Weighted scoring logic
}

```

**Verdict:** Refactoring avoids duplication, reuses proven code.

---

## 2. Backward Compatibility

**Requirement:** Existing code using `run_spec_consensus()` must continue working.

**Refactoring approach** (✓ backward compatible):

```

// Old callers (no changes needed)
let result = run_spec_consensus(spec_id, stage, artifacts).await?;

```

```
// Still works! Delegates to weighted version internally
```

**New module approach** ( $\Delta$  requires migration):

```
// Old code breaks unless we keep both implementations  
use crate::consensus::run_spec_consensus; // Old  
use crate::weighted_consensus::run_weighted_consensus; // New  
  
// Users must decide which to call
```

**Verdict:** Refactoring ensures zero breaking changes.

---

### 3. Simplicity (Single Source of Truth)

**Refactoring:** 1 file, 1 consensus implementation - All consensus logic in consensus.rs - Easier to find (git grep "consensus") - Single import: use crate::chatwidget::spec\_kit::consensus

**New module:** 2 files, 2 implementations - consensus.rs (technical-only, legacy) - weighted\_consensus.rs (PPP) - Users confused: which should I use? - Maintenance burden: must keep both updated

**Verdict:** Refactoring reduces cognitive load.

---

### 4. Incremental Migration

**Refactoring strategy:** 1. **Phase 1:** Add

run\_spec\_consensus\_weighted(), keep run\_spec\_consensus() as wrapper

2. **Phase 2:** Gradually migrate callers to weighted version 3. **Phase 3:**

Deprecate old function (if desired)

**Example migration:**

```
// Old caller  
let result = run_spec_consensus(spec_id, stage, artifacts).await?;  
  
// Migrate to weighted (Phase 2)  
let weighted = run_spec_consensus_weighted(  
    spec_id,  
    stage,  
    artifacts,  
    get_weights_from_config(), // From config.toml  
).await?;
```

**Verdict:** Refactoring enables gradual adoption (no big-bang migration).

---

### 5. Performance (Shared Database Connection)

**Refactoring** (single DB connection):

```
pub async fn run_spec_consensus_weighted(...) ->  
Result<WeightedConsensus> {  
    let db = open_consensus_db()?; // Open once  
  
    for artifact in artifacts {  
        let technical = calculate_technical_score(&artifact)?;
```

```

        let trajectory_id = get_trajectory_id(&db, ...)?; // Reuse
connection
        let proact = calculate_r_proact(&db, trajectory_id)?; //
Reuse connection
        // ...
    }
}

```

**New module** (duplicate connections):

```

// consensus.rs
pub async fn run_spec_consensus(...) {
    let db = open_consensus_db()?; // Connection 1
    // ...
}

// weighted_consensus.rs
pub async fn run_weighted_consensus(...) {
    let db = open_consensus_db()?; // Connection 2 (duplicate)
    // ...
}

```

**Impact:** Minimal (SQLite connection is cheap), but refactoring is cleaner.

**Verdict:** Refactoring avoids redundant connections.

## Comparison to Alternatives

Approach	Code Reuse	Backward Compat	Complexity	Migration	Decision
<b>Refactor consensus.rs</b>	✓ High (80%)	✓ 100%	✓ Low	✓ Gradual	✓ AC
New module	✗ Low (20%)	⚠ Requires wrapper	⚠ Medium	⚠ Big-bang	✗ R
New crate	✗ None	✗ Breaking	✗ High	✗ Major	✗ R
Trait-based	⚠ Medium	✓ Good	✗ High	⚠ Big-bang	✗ R

### Alternative 1: New Module (weighted\_consensus.rs)

**Structure:**

```

codex-rs/tui/src/chatwidget/spec_kit/
├── consensus.rs (existing, 1200 lines)
└── weighted_consensus.rs (new, 500 lines)

```

**Pros:** - Clean separation (PPP code isolated) - Easy to delete if PPP fails (just remove file) - Parallel development (no merge conflicts)

**Cons:** - Duplicates DB access, validation, error handling (~300 lines) - Two consensus implementations to maintain - Users confused: which to call? - Backward compatibility requires keeping both forever

**Verdict:** ✗ Reject - Duplication outweighs separation benefits.

---

## Alternative 2: New Crate (codex-consensus)

**Structure:**

```
codex-rs/
├── tui/
└── consensus/ (new crate)
    ├── src/
    │   ├── lib.rs
    │   ├── technical.rs
    │   └── weighted.rs
```

**Pros:** - Maximum reusability (other tools could use) - Strong API boundaries (pub/private) - Independent versioning

**Cons:** - Over-engineering (consensus only used by codex-tui) - Breaking change (move existing code to new crate) - Adds dependency (codex-tui depends on codex-consensus) - Complexity (separate crate to manage)

**Verdict:** ✗ Reject - Premature optimization (no other consumers).

---

## Alternative 3: Trait-Based (Polymorphism)

**Structure:**

```
pub trait Consensus {
    async fn select_best(
        &self,
        artifacts: Vec<ConsensusArtifactData>,
    ) -> Result<String>;
}

pub struct TechnicalConsensus;
impl Consensus for TechnicalConsensus { ... }

pub struct WeightedConsensus;
impl Consensus for WeightedConsensus { ... }

// Usage
let consensus: Box<dyn Consensus> = if config.ppp.enabled {
    Box::new(WeightedConsensus::new())
} else {
    Box::new(TechnicalConsensus::new())
};

let best = consensus.select_best(artifacts).await?;
```

**Pros:** - Flexible (can swap implementations at runtime) - Testable (mock consensus for tests) - Extensible (add new consensus algorithms easily)

**Cons:** - Complex (trait + 2 implementations + dynamic dispatch) - Overhead (Box allocations) - Overkill (only 2 implementations, unlikely to add more)

**Verdict:** ✗ Reject - Unnecessary abstraction for 2 implementations.

---

## Consequences

### Positive

1. ✓ **Code reuse:** Avoids duplicating DB access, validation, error handling
2. ✓ **Backward compatible:** Existing callers work without changes
3. ✓ **Single file:** All consensus logic in one place (easier to find)
4. ✓ **Gradual migration:** Can adopt weighted consensus incrementally
5. ✓ **Minimal diff:** Smaller PR (refactor vs new file)

### Negative

1. ⚠ **Larger file:** consensus.rs grows from ~1200 → ~1500 lines
  - Mitigation: Still manageable (<2000 line threshold)
  - Alternative: Can split later if becomes unwieldy
2. ⚠ **Coupling:** PPP logic coupled to consensus.rs
  - Mitigation: Clear function separation (`_weighted` suffix)
  - Impact: Low (PPP is consensus feature, coupling is natural)
3. ⚠ **Testing:** Must test both old and new functions
  - Mitigation: Old function delegates to new (test new = tests both)

### Neutral

1. ⚠ **Migration path:** Old → wrapper → weighted (3-step process)
    - Phase 1: Add weighted function
    - Phase 2: Change callers to use weighted
    - Phase 3: Remove old function (optional)
- 

## Implementation Plan

### Phase 1: Add Weighted Function

```
// consensus.rs (add at end of file)

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

pub struct WeightedConsensusResult {
    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)>,
) -> Result<WeightedConsensusResult> {
    // ... implementation (see recommendations.md)
}

```

**Lines added:** ~150 lines (structs + function)

---

## Phase 2: Refactor Old Function

```

// consensus.rs (modify existing function)

pub async fn run_spec_consensus(
    spec_id: &str,
    stage: &str,
    artifacts: Vec<ConsensusArtifactData>,
) -> Result<ConsensusResult> {
    // Old implementation (300 lines) → DELETE

    // New implementation (delegate to weighted)
    let weighted = run_spec_consensus_weighted(
        spec_id,
        stage,
        artifacts,
        Some((0.7, 0.3)), // Default weights
    ).await?;

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

```

**Lines removed:** ~280 lines (old technical-only logic) **Lines added:** ~20 lines (delegation + conversion) **Net change:** -260 lines (simplification!)

---

## Phase 3: Migration (Optional)

**Gradually replace** old callers:

```

// Before
let result = run_spec_consensus(spec_id, stage, artifacts).await?;
let best_agent = result.selected_agent;

// After
let weighted = run_spec_consensus_weighted(
    spec_id,
    stage,
    artifacts,
    get_config_weights(), // User-configurable
).await?;
let best_agent = weighted.best_agent;

```

**Timeline:** Phase 2-3 (after initial deployment)

---

## Testing Strategy

### Unit Tests

```
#[cfg(test)]
mod tests {
    #[tokio::test]
    async fn test_backward_compatibility() {
        // Old function should still work
        let result = run_spec_consensus("SPEC-001", "plan",
artifacts).await.unwrap();
        assert!(result.selected_agent.len() > 0);
    }

    #[tokio::test]
    async fn test_weighted_consensus() {
        // New function with custom weights
        let weighted = run_spec_consensus_weighted(
            "SPEC-001",
            "plan",
            artifacts,
            Some((0.6, 0.4)),
        ).await.unwrap();

        assert_eq!(weighted.scores.len(), 3);
        assert!(weighted.confidence > 0.0);
    }

    #[tokio::test]
    async fn test_delegation() {
        // Old and new should select same agent (with default
weights)
        let old_result = run_spec_consensus("SPEC-001", "plan",
artifacts.clone()).await.unwrap();
        let new_result = run_spec_consensus_weighted(
            "SPEC-001",
            "plan",
            artifacts,
            Some((0.7, 0.3)),
        ).await.unwrap();

        assert_eq!(old_result.selected_agent,
new_result.best_agent);
    }
}
```

---

## Rollback Plan

If refactoring causes issues:

**Option 1:** Revert to pre-refactor state

```
git revert <commit-hash>
```

**Option 2:** Disable PPP in config

```
[ppp]
enabled = false # Fall back to technical-only
```

**Option 3:** Keep both implementations

```
// Restore old run_spec_consensus() implementation
// Rename weighted version to run_spec_consensus_ppp()
```

**Trigger:** Production incident or consensus failures >5%.

---

## References

1. Martin Fowler, "Refactoring" - Refactor before adding features
  2. Dependency Inversion Principle - Depend on abstractions (traits) not concrete implementations
  3. Single Responsibility Principle - Each module should have one reason to change
  4. consensus.rs (existing) - 1200 lines, lines 681-958 most relevant
- 

## Decision Matrix

Criterion	Refactor	New Module	New Crate	Trait-Based
<b>Code Reuse</b>	✓ 80%	✗ 20%	△ 50%	△ 60%
<b>Backward Compat</b>	✓ 100%	△ Wrapper	✗ Breaking	✓ Good
<b>Simplicity</b>	✓ <b>1 file</b>	△ 2 files	✗ New crate	✗ Complex
<b>Maintenance</b>	✓ <b>Low</b>	△ Medium	△ Medium	✗ High
<b>Testability</b>	✓ Good	✓ Good	✓ Good	✓ <b>Excellent</b>
<b>Migration</b>	✓ <b>Gradual</b>	△ Big-bang	✗ Breaking	△ Big-bang

**Total Score:** Refactor **95%**, New Module **60%**, New Crate **40%**, Trait-Based **70%**

**Winner:** Refactor consensus.rs - Best overall score.