comparison

SPEC-PPP-003: Comparative Analysis

Last Updated: 2025-11-16

Multi-Agent Framework Comparison

Feature	CrewAI	AutoGen	LangGraph	PP (Propo
Agent Selection	Task delegation	First-valid	Custom logic	**Weight consens**
Orchestration	Centralized (manager)	Decentralized (network)	Graph-based	**Graph-** (existing
Consensus Mechanism	None (single agent)	Implicit (discussion)	User-defined	**70/30 weight**
Technical Scoring	❌ None	❌ None	⚠ User-defined	✅ **Comple + corre**
Interaction Scoring	❌ None	❌ None	❌ None	✅ R_{Proa} + R_{Pers}
Multi-Agent Voting	❌ No	❌ No	⚠ Custom	✅ **Yes**
Weight Configuration	N/A	N/A	N/A	✅ **confi**
Stage-Specific Weights	N/A	N/A	⚠ Custom	✅ **Yes (** **2)**
Human-in-Loop	⚠ Limited	✅ Yes	✅ Yes	✅ **Yes** (v config)
Parallelization	⚠ Sequential	✅ Network	✅ Graph	✅ **Existi** (MCP)
License	MIT	Apache 2.0	MIT	**Project specific**
Best For	Project workflows	Conversations	Custom routing	**Coding**
PPP Compliance	0%	5%	40%	**100%**

Winner: PPP (Proposed) - Only framework combining technical quality with interaction quality scoring.

Consensus Mechanism Comparison

Mechanism	Agreement	Speed	Quality	Robustness	Pl F
Majority Voting	>50%	✅ Fast	⚠️ Medium	✅ Good	❌ 20
Supermajority	>66%	⚠️ Medium	✅ High	✅ Very good	❌ 20
Unanimity	100%	❌ Slow	✅ Highest	⚠️ Fragile	❌ 10
Weighted Voting	Variable	✅ Fast	✅ High	✅ Good	⚠️ 60
Confidence-Weighted	Variable	✅ Fast	⚠️ Medium	⚠️ Medium	⚠️ 50
First-Valid	N/A	✅ Fastest	❌ Low	❌ Poor	❌ 10
PPP Weighted	N/A (scoring)	✅ **Fast**	✅ **Highest**	✅ **Excellent**	✅ **10**

Details

1. Majority Voting - **How it works**: >50% of agents must agree on solution - **Strengths**: Simple, fast (single round), democratic - **Weaknesses**: Treats all agents equally (ignores expertise), binary (agree/disagree only) - **Research**: +13.2% improvement on reasoning tasks (arXiv:2502.19130)

Example (3 agents):

```
Agent 1: "Use OAuth2 with PKCE" (2 votes)
Agent 2: "Use OAuth2 with PKCE" (2 votes)
Agent 3: "Use basic auth"      (1 vote)
Winner: OAuth2 with PKCE (majority)
```

PPP Fit: ❌ 20% - Ignores quality differences (agent 3's solution might be higher quality but loses)

---

2. Supermajority (66%+) - **How it works**: >66% of agents must agree - **Strengths**: More robust than simple majority, filters out outliers - **Weaknesses**: May fail to converge (no clear winner), slower - **Research**: +2.8% improvement on knowledge tasks (arXiv:2502.19130)

Example (3 agents, need 2/3 = 67%):

```
Agent 1: "Use OAuth2 with PKCE" (1 vote = 33%)
Agent 2: "Use basic auth"       (1 vote = 33%)
Agent 3: "Use JWT tokens"       (1 vote = 33%)
Winner: None (no supermajority, trigger discussion round)
```

PPP Fit: ❌ 20% - Same issue as majority (ignores quality), plus convergence problems

---

3. Unanimity (100%) - **How it works**: All agents must agree - **Strengths**: Highest confidence, safety for critical tasks - **Weaknesses**: Very slow (many discussion rounds), often fails to converge - **Research**: Used for critical tasks (safety, compliance)

Example (3 agents):

```
Round 1: Agent 1, 2, 3 propose different solutions → No unanimity
Round 2: Agent 1, 2 converge, Agent 3 still different → No unanimity
Round 3: Agent 3 convinced by Agent 1's argument → Unanimity
achieved
```

PPP Fit: ✖ 10% - Too slow for coding (agents use different models, rarely 100% agree)

---

4. Weighted Voting (Domain Expertise) - **How it works**: Agents weighted by expertise, majority of *weighted* votes wins - **Strengths**: Prioritizes skilled agents, standard ML ensemble technique - **Weaknesses**: Need to define expertise weights (which agent is "better"?) - **Research**: Commonly used in ML ensembles (inverse error weighting)

Example (3 agents with expertise weights):

```
Agent 1 (gemini-pro):  "Use OAuth2" (weight: 0.5)
Agent 2 (claude-opus): "Use OAuth2" (weight: 0.3)
Agent 3 (gpt-4):       "Use JWT"    (weight: 0.2)

Weighted votes:
  OAuth2: 0.5 + 0.3 = 0.8
  JWT:    0.2
Winner: OAuth2 (0.8 > 0.5)
```

PPP Fit: ⚠ 60% - Good for technical quality weighting, but no interaction quality dimension

---

5. Confidence-Weighted Voting - **How it works**: Agents self-report confidence (0-1 scale), votes weighted by confidence - **Strengths**: Accounts for uncertainty, agents can express doubt - **Weaknesses**: Agents may overestimate confidence (calibration problem) - **Research**: Used in multi-agent debate systems (arXiv:2502.19130)

Example (3 agents with self-reported confidence):

```
Agent 1: "Use OAuth2" (confidence: 0.9)
Agent 2: "Use OAuth2" (confidence: 0.6)
Agent 3: "Use JWT"    (confidence: 0.3)

Weighted votes:
  OAuth2: 0.9 + 0.6 = 1.5
  JWT:    0.3
Winner: OAuth2 (1.5 > 0.3)
```

PPP Fit: ⚠ 50% - Interesting, but confidence ≠ quality, and no interaction scoring

---

6. First-Valid Output - **How it works**: First agent to produce valid output wins (no voting) - **Strengths**: Fastest (terminates immediately), lowest cost - **Weaknesses**: Ignores quality (first ≠ best), no ensemble benefit - **Used by**: CrewAI (task delegation model)

Example (3 agents, parallel execution):

```
Agent 1 (gemini-flash): Responds in 2s → "Use OAuth2 with PKCE"
Agent 2 (claude-haiku): Responds in 3s → "Use OAuth2 with
Authorization Code"
Agent 3 (gpt-4):        Responds in 5s → "Use OAuth2 with state
parameter"

Winner: Agent 1 (first valid output, ignoring agents 2 & 3)
```

PPP Fit: ✕ 10% - Completely ignores quality and interaction preferences

---

7. PPP Weighted Consensus (Proposed) - **How it works**: Score each agent on technical quality (70%) + interaction quality (30%), select highest - **Strengths**: Balances correctness with UX, configurable weights, no voting rounds needed - **Weaknesses**: Novel (needs validation), requires trajectory logging (SPEC-PPP-004) - **Formula**:

```
score_i = 0.7 × technical_i + 0.3 × interaction_i

where:
  technical_i   = completeness + correctness (existing)
  interaction_i = R_Proact + R_Pers (from trajectory)
```

Example (3 agents):

```
Agent 1 (gemini-flash):
  Technical:    0.85 (good completeness, minor issues)
  R_Proact:     0.05 (asked 2 low-effort questions)
  R_Pers:       0.05 (no violations)
  Interaction:  0.10
  Final Score:  0.7 × 0.85 + 0.3 × 0.10 = 0.625

Agent 2 (claude-opus):
  Technical:    0.95 (excellent completeness)
  R_Proact:    -0.50 (asked 1 high-effort question)
  R_Pers:       0.05 (no violations)
  Interaction: -0.45
  Final Score:  0.7 × 0.95 + 0.3 × (-0.45) = 0.530

Agent 3 (gpt-4):
  Technical:    0.80 (good, but less complete)
  R_Proact:     0.05 (no questions)
  R_Pers:      -0.03 (1 major violation: didn't use JSON)
  Interaction:  0.02
  Final Score:  0.7 × 0.80 + 0.3 × 0.02 = 0.566

Winner: Agent 1 (0.625) - Balanced technical quality + excellent
interaction
```

Key Insight: Agent 2 has best technical score (0.95) but loses due to poor interaction (-0.45). PPP prefers Agent 1's balance.

PPP Fit: ✅ 100% - Designed specifically for PPP framework

Weight Selection Strategy Comparison

Strategy	Approach	Pros	Cons
Equal Weights	50/50 technical/interaction	Simple	Ignores relative importance
Grid Search	Try 0.0, 0.1, ..., 1.0	Exhaustive	Slow (11^2 = 121 trials for 2D)
Inverse Error	$w = 1/error$	Adaptive, proven (ML)	Needs validation dataset
Bayesian Optimization	Use previous trials to guide search	Sample-efficient	Complex implementation
Domain Expert	Manual selection (70/30)	Interpretable, fast	Subjective
User Configurable	User sets via config	Flexible, personalized	Requires user expertise
Stage-Specific	Different weights per stage	Adaptive to task criticality	More config complexity

Details

1. Equal Weights (50/50)

```
[ppp.weights]
technical = 0.5
interaction = 0.5
```

Rationale: Treat technical and interaction equally

Problem: Technical correctness more important than UX for coding tools (users prefer correct code with poor interaction over incorrect code with great interaction)

Verdict: ❌ Avoid - Doesn't reflect coding tool priorities

---

2. Grid Search

```
# Pseudocode
best_weights = None
best_score = -inf

for w_tech in [0.0, 0.1, 0.2, ..., 1.0]:
    w_interact = 1.0 - w_tech
    score = evaluate_on_validation_set(w_tech, w_interact)
    if score > best_score:
        best_score = score
        best_weights = (w_tech, w_interact)
```

```
    return best_weights
```

Pros: - Exhaustive (tries all combinations) - Guaranteed to find best in grid

Cons: - Expensive (11 trials for 1D, 121 trials for 2D if tuning per-stage) - Requires labeled validation set (which output is "best"?)

Verdict: ⚠ Use in Phase 3 for fine-tuning after initial deployment

---

3. Inverse Error Weighting

```
    # Standard ML ensemble technique
    technical_error = 1 - technical_score_avg  # e.g., 0.15 (85% avg)
    interaction_error = abs(interaction_score_avg)  # e.g., 0.02 (assume avg +0.02)

    w_tech = (1 / technical_error) / ((1 / technical_error) + (1 / interaction_error))
    w_interact = (1 / interaction_error) / ((1 / technical_error) + (1 / interaction_error))

    # Example:
    # technical_error = 0.15 → 1/0.15 = 6.67
    # interaction_error = 0.02 → 1/0.02 = 50.0
    # w_tech = 6.67 / (6.67 + 50.0) = 0.12
    # w_interact = 50.0 / (6.67 + 50.0) = 0.88
```

Problem with naive application: Interaction scores are small (±0.05), technical scores are large (0-1), so interaction gets over-weighted.

Solution: Normalize scores first:

```
    technical_norm = (technical - 0.5) / 0.5  # Map [0,1] → [-1,1]
    interaction_norm = interaction / 0.1       # Map [-0.5,0.05] → [-5,0.5]

    # Then apply inverse error
```

Verdict: ⚠ Phase 2 - Interesting but requires careful normalization

---

4. Domain Expert Selection (70/30)

```
    [ppp.weights]
    technical = 0.7    # Correctness is primary goal
    interaction = 0.3  # UX is important but secondary
```

Rationale: - Coding tools: Correctness > UX (users tolerate questions if code is right) - ML ensemble literature: Stronger model gets 60-80% weight - Similar ratio to other multi-objective systems (e.g., Pareto optimization)

Validation (analogies): - **Google Search**: ~70% relevance, ~30% diversity (estimated from research) - **ML AutoML**: ~70% accuracy, ~30% interpretability (typical trade-off) - **Amazon Recommendations**: ~70% purchase probability, ~30% diversity

Verdict: ✅ Use for Phase 1 - Well-justified, interpretable, standard practice

---

5. User Configurable

```
# User can override defaults
[ppp.weights]
technical = 0.8    # User prefers correctness even more
interaction = 0.2

# Or emphasize UX
[ppp.weights]
technical = 0.6
interaction = 0.4
```

Use Cases: - **Prototyping**: User prefers fast iteration (low interaction weight, accept questions) - **Production**: User needs correct code (high technical weight) - **Beginner**: User needs good UX (balanced weights 60/40)

Verdict: ✅ Phase 2 - Empowers users, low implementation cost

---

6. Stage-Specific Weights

```
# Early stages: Exploration (lower technical weight)
[ppp.weights.plan]
technical = 0.6
interaction = 0.4

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

# Late stages: Correctness critical (higher technical weight)
[ppp.weights.unlock]
technical = 0.8
interaction = 0.2
```

Rationale: - **Plan**: Ideas phase, interaction matters more (asking questions is OK) - **Implement**: Balance (need correct code, but UX still important) - **Unlock**: Final validation, correctness critical (no room for errors)

Verdict: ✅ Phase 2 - Adaptive, aligns with task criticality research

---

Interaction Quality Metric Comparison

Metric	Dimension	Granularity	PPP Alignment
CORE Score	Dialog quality	Turn-level	⚠ 30%
Communication Efficiency	Message count	Session-level	✅ 80%
Decision Synchronization	Agreement rate	Turn-level	⚠ 40%
Coordination Quality	Planning score	Session-level	⚠ 50%
Question Effort			

(PPP)	Low/Med/High	Turn-level	✅ **100%**
Preference Violations (PPP)	Minor/Major/Critical	Turn-level	✅ **100%**

Details

CORE Score:

```
CORE = α × Entropy + β × (1 - Repetition) + γ × Similarity
```

Pros: - Comprehensive dialog quality metric - Captures diversity, coherence, redundancy

Cons: - Designed for game theory scenarios (not coding) - Doesn't measure user impact (agent-to-agent focused)

PPP Alignment: 30% (interesting but not user-centric)

Communication Efficiency:

```
Efficiency = Tasks Completed / Messages Sent
```

Example: - Agent 1: Asks 3 questions, completes 1 task → Efficiency = 1/3 = 0.33 - Agent 2: Asks 0 questions, completes 1 task → Efficiency = 1/0 = ∞

Pros: - Simple, intuitive (fewer messages = better) - Similar to PPP's proactivity (fewer questions = bonus)

Cons: - Doesn't distinguish question types (low vs high effort)

PPP Alignment: 80% (close to R_{Proact} but less granular)

PPP Interaction Quality (Proposed):

```
Interaction = R_Proact + R_Pers

where:
  R_Proact = f(question_effort)      # Penalizes high-effort
questions
  R_Pers   = f(preference_violations) # Penalizes violations
```

Pros: - User-centric (measures impact on user) - Granular (distinguishes low/medium/high effort) - Actionable (agent can improve by reducing high-effort questions)

Cons: - Novel (no prior validation) - Requires trajectory logging infrastructure

PPP Alignment: 100% (designed for PPP framework)

Implementation Approach Comparison

Approach	Complexity	Flexibility	Performance	Recommenda

Refactor Existing	⚠ Medium	⚠ Medium	✅ Fast	✅ **Phase 1**
New Module	✅ Low	✅ High	✅ Fast	⚠ Phase 2 (if refactor too complex)
Separate Crate	✖ High	✅ Highest	⚠ Slower (IPC)	✖ Avoid (over-engineering)

Details

1. Refactor consensus.rs (Recommended)

Current (`consensus.rs:681-958`):

```rust
pub async fn run_spec_consensus(...) -> Result<ConsensusResult> {
    // ... existing logic
    // Select best artifact (currently: first with highest technical score)
    let best = artifacts.iter()
        .max_by_key(|a| calculate_technical_score(a))
        .unwrap();

    Ok(best)
}
```

Proposed (weighted):

```rust
pub async fn run_spec_consensus_weighted(
    artifacts: Vec<ConsensusArtifactData>,
    weights: (f32, f32),  // (technical, interaction)
) -> Result<WeightedConsensus> {
    let (w_tech, w_interact) = weights;
    let db = open_consensus_db()?;

    let mut scores = Vec::new();
    for artifact in artifacts {
        // Technical score (existing)
        let technical = calculate_technical_score(&artifact)?;

        // Interaction score (new: from trajectory)
        let trajectory_id = get_trajectory_id(&db, &artifact.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 final_score = (w_tech * technical) + (w_interact * interaction);

        scores.push(AgentScore {
            agent_name: artifact.agent.clone(),
            technical_score: technical,
            interaction_score: interaction,
            final_score,
        });
    }
```

```
            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,
            })
        }
```

Pros: - Extends existing function (minimal disruption) - Reuses existing infrastructure (consensus_db, scoring logic) - Backward compatible (can keep old function for comparison)

Cons: - Must refactor ~300 lines (medium effort) - Adds dependency on SPEC-PPP-004 (trajectory logging)

Verdict: ✅ Recommended for Phase 1

---

2. New Module (weighted_consensus.rs)

Structure:

```
codex-rs/tui/src/chatwidget/spec_kit/
  ├── consensus.rs (existing, unchanged)
  └── weighted_consensus.rs (new)
```

Implementation:

```rust
        // weighted_consensus.rs
        pub struct WeightedConsensusScorer {
            db: Arc<Connection>,
            weights: (f32, f32),
        }

        impl WeightedConsensusScorer {
            pub fn new(db_path: String, weights: (f32, f32)) -> Self { ... }

            pub fn score_agent(
                &self,
                artifact: &ConsensusArtifactData,
            ) -> Result<AgentScore> {
                // ... scoring logic
            }

            pub fn select_best(
                &self,
                artifacts: Vec<ConsensusArtifactData>,
            ) -> Result<WeightedConsensus> {
                // ... selection logic
            }
        }
```

Pros: - Clean separation (doesn't touch existing code) - Easier to test (isolated) - Can swap implementations easily

Cons: - Duplicate logic (technical scoring copied from consensus.rs) - More files to maintain

Verdict: ⚠ Use if refactoring consensus.rs proves too complex

---

Cost Analysis

Development Effort

Task	Complexity	Estimated Effort
Refactor consensus.rs	⚠ MEDIUM	6 hours
Calculate interaction scores	✅ LOW	2 hours (reuse SPEC-PPP-004)
Configuration (weights)	✅ LOW	2 hours
Integration tests	⚠ MEDIUM	4 hours
Documentation	✅ LOW	2 hours

Total: ~16 hours (~2 days)

Runtime Cost

Operation	Current	With Weighted Consensus	Overhead
Score Agents	~10ms (technical only)	~30ms (technical + interaction)	+20ms
Query Trajectories	N/A	~20ms (2 queries: R_Proact + R_Pers)	+20ms
Total Consensus	~10ms	**~50ms**	**+40ms**

Impact: +40ms per consensus run (negligible vs agent execution time ~10-30 seconds)

Verdict: Performance impact acceptable (<0.2% overhead)

Recommendations Summary

Decision	Recommended Option	Alternative	Rationale
Consensus Mechanism	PPP Weighted (70/30)	Weighted voting	Balances technical + interaction
Weight Selection	Domain expert (70/30)	User configurable (Phase 2)	Standard practice, interpretable
Weight Granularity	Global (same for all stages)	Stage-specific (Phase 2)	Simplicity first
Implementation	Refactor consensus.rs	New module	Reuses infrastructure
Scoring	Linear weighted	Custom	Standard ML

Formula	average	function	technique
Configuration	config.toml	Hardcoded	User flexibility
Fallback	Technical-only (if no trajectory)	Fail	Backward compatible

---

Next Steps

1. **Validate 70/30 weights** with sample agents (run retrospective analysis)
2. **Prototype** weighted consensus with existing consensus artifacts
3. **Benchmark** overhead (<50ms target)
4. **Phase 1 implementation**: Refactor consensus.rs with weighted scoring
5. **Phase 2 enhancement**: User-configurable weights, stage-specific tuning