SKILL-utf8

1 CC2.0 COLLABORATE Function Skill

Function: COLLABORATE - Multi-Agent Coordination & Team Synchronization **Category Theory**: Applicative + Parallel Composition **Purpose**: Orchestrate multiple agents with token optimization and conflict resolution **Status**: =6 Planned for Phase 2 (Weeks 4-6)

1.1 When to Use This Skill

Use COLLABORATE when you need to: - > **Multi-agent coordination** - Orchestrate multiple agents working on related tasks - = **Token optimization** - Share context efficiently across agents to reduce redundancy - i **Parallel execution** - Execute independent tasks concurrently for speed - = **Conflict resolution** - Merge outputs from different agents coherently - =Ê **Load balancing** - Distribute work based on agent capabilities and availability

---

1.2 Core Capabilities (Planned)

1.2.1 1. Agent Orchestration (Applicative Functor)

```
// Coordinate multiple agents
const result = await collaborate({
  agents: [
    { type: 'frontend-architect', task: 'UI design' },
    { type: 'backend-expert', task: 'API implementation' },
    { type: 'test-engineer', task: 'Test suite' }
  ],
  coordination: 'parallel',
  tokenBudget: 100000
});
```

1.2.2 2. Context Sharing

```
// Share extracted context across agents
const sharedContext = extractContext(task);
const results = await Promise.all(
  agents.map(agent => agent.execute(task, sharedContext))
);
// Token savings: ~40% via shared context
```

1.2.3 3. Conflict Resolution

```
// Merge agent outputs coherently
const merged = await resolveConflicts({
  outputs: [frontend, backend, tests],
  strategy: 'consensus',
  validator: ensureConsistency
});
```

1.3 Planned Features

1.3.1 Token Optimization Strategies

Context Extraction: - Extract common context once, share across agents - Typical savings: 40-60% of total token usage - Example: 680KB specs 10KB context (98.5% reduction)

Parallel Execution: - Independent agents run concurrently - Reduces total time by 50-70% - Token efficiency through batching

Incremental Updates: - Only send deltas to agents, not full state - Reduces token usage for iterative refinement

1.3.2 Coordination Patterns

1. Pipeline Pattern (Sequential)

```
const result = await collaborate({
  agents: [
    { agent: observe, output: 'observation' },
    { agent: reason, input: 'observation', output: 'plan' },
    { agent: create, input: 'plan', output: 'artifact' }
  ],
  pattern: 'pipeline'
});
```

2. Parallel Pattern (Concurrent)

```
const results = await collaborate({
  agents: [
    { agent: implementFeature, task: 'feature' },
    { agent: writeTests, task: 'tests' },
    { agent: updateDocs, task: 'docs' }
  ],
  pattern: 'parallel'
});
```

3. MapReduce Pattern

```
const result = await collaborate({
  map: agents.map(agent => agent.execute(subtask)),
  reduce: (outputs) => merge(outputs)
});
```

1.4 Integration with Meta-Infrastructure

1.4.1 Entity-Based Coordination

```
// Track collaboration patterns per entity
await storage.teams.update(teamId, {
  collaborationMetrics: {
    avgAgentsPerTask: 3.2,
    parallelExecutionRate: 0.68,
    tokenSavings: 0.42
  }
});
```

1.4.2 Quality Across Agents

```
// Ensure consistent quality across agent outputs
const quality = await qualityAnalyzer.analyzeMulti(
  agentOutputs,
  { consistencyCheck: true }
);
```

1.5 Categorical Structure

1.5.1 Applicative Functor Laws

1. Identity: `pure(id) <*> v a v` **2. Composition**: `pure( ) <*> u <*> v <*> w a u <*> (v <*> w)` **3. Homomorphism**: `pure(f) <*> pure(x) a pure(f(x))` **4. Interchange**: `u <*> pure(y) a pure(»f.f(y)) <*> u`

Status: ó Will be verified in Phase 2 implementation

1.6 Example Workflows

1.6.1 Workflow 1: Feature Development Team

```javascript
// Parallel feature implementation
const feature = await collaborate({
  agents: [
    { type: 'practical-programmer', task: 'Backend logic', priority: 1 },
    { type: 'frontend-architect', task: 'UI components', priority: 1 },
    { type: 'test-engineer', task: 'Test suite', priority: 2 }
  ],
  tokenBudget: 120000,
  pattern: 'parallel-with-sync'
});

// Token savings: ~45% via shared context
// Time savings: ~60% via parallelization
```

1.6.2 Workflow 2: Code Review

```javascript
// Multiple perspectives on code quality
const review = await collaborate({
  agents: [
    { type: 'code-trimmer', focus: 'Maintainability' },
    { type: 'debug-detective', focus: 'Correctness' },
    { type: 'frontend-architect', focus: 'Best practices' }
  ],
  pattern: 'consensus',
  merge: 'weighted'
});
```

1.7 Performance Targets

- **Token Savings**: 40-60% via context sharing
- **Time Reduction**: 50-70% via parallel execution
- **Quality Consistency**: >90% agreement across agents
- **Coordination Overhead**: <5% of total execution time

1.8 Implementation Timeline

Phase 2 (Weeks 4-6): - Week 4: Basic orchestration framework - Week 5: Token optimization strategies - Week 6: Conflict resolution and testing

Dependencies: - OBSERVE, REASON, CREATE functions (implemented) - Entity storage system (implemented) - Agent SDK integration (implemented) - ó VERIFY function (needed for quality validation)

1.9 Related Skills

- **cc2-observe**: Provides system state for coordination
- **cc2-reason**: Plans collaboration strategies
- **cc2-create**: Generates implementations via agents
- **cc2-verify**: Validates multi-agent outputs
- **cc2-meta-orchestrator**: Manages COLLABORATE in workflows

---

1.10 References

- Specification: `~/cc2.0/functions/collaborate/FUNCTION.md` (planned)
- Integration: `~/cc2.0/functions/collaborate/cc_integration.md` (planned)
- Phase 2 Roadmap: `~/cc2.0/PHASE-1-FOUNDATION-SPEC.md`

1.11 Notes

This skill represents planned functionality for Phase 2 of CC2.0 development. The categorical structure (Applicative Functor) and coordination patterns are designed, but implementation is pending completion of foundation functions (OBSERVE, REASON, CREATE, VERIFY).

Expected Delivery: Week 6 of Phase 1 implementation plan

1.12 Meta-Function: COLLABORATE_SELF

COLLABORATE will eventually observe and improve its own coordination patterns:

```
// Meta-collaboration (planned)
const metaCollaborate = await collaborateSelf(coordinationHistory);

// Returns insights about collaboration efficiency
{
  optimalAgentCount: 3.5,
  bestPattern: "parallel-with-sync",
  tokenEfficiency: 0.68,
  suggestions: [
    "Use pipeline for sequential dependencies",
    "Increase parallelization for independent tasks"
  ]
}
```

This enables continuous improvement of multi-agent coordination strategies.