# SKILL

# 1 CC2.0 OBSERVE Function Skill

**Function**: OBSERVE - System State Observation **Category Theory**: Comonad (extract, extend, duplicate) **Purpose**: Observe system state and extract comonadic structures for analysis **Status**: ✅ Production-Ready (100% complete, 18/18 laws passing)

---

## 1.1 When to Use This Skill

Use OBSERVE when you need to: - 🔍 **Analyze system state** - Current code, metrics, environment - 📊 **Extract patterns** - Identify recurring structures and behaviors - 🎯 **Contextualize data** - Wrap data with relevant context (comonad) - 🔄 **Enable composition** - Prepare data for REASON → CREATE chain - 📈 **Track trends** - Time-series analysis, quality metrics

---

## 1.2 Core Capabilities

### 1.2.1 1. State Observation (Comonad Extract)

```
// Extract current value from context
const observation = await observe(systemState);
// Returns: { value: T, context: Context }
```

### 1.2.2 2. Context Extension (Comonad Extend)

```
// Build new observations from existing context
const extended = observation.extend(obs => analyze(obs));
// Returns: New observation with enriched context
```

### 1.2.3 3. Duplication (Comonad Duplicate)

```
// Create nested observations (meta-observation)
const meta = observation.duplicate();
// Returns: Observation of observations
```

---

## 1.3 Input/Output Format

### 1.3.1 Input

```
{
  "systemState": {
    "code": "...",
    "metrics": {...},
    "environment": {...}
  },
  "options": {
    "depth": "shallow" | "deep",
    "includeMetrics": boolean,
    "contextWindow": number
  }
}
```

### 1.3.2 Output

```
{
  "observation": {
    "value": "...",
    "context": {
      "timestamp": "...",
      "quality": 0.85,
      "patterns": [...],
      "trends": [...]
    }
  },
  "confidence": 0.92,
  "nextActions": [...]
}
```

---

## 1.4 Integration with Meta-Infrastructure

### 1.4.1 Entity Storage

```
// Store observations for historical analysis
await storage.developers.update(devId, {
  observations: [...existing, observation]
});
```

### 1.4.2 Quality Analyzer

```
// Use observations for quality analysis
const quality = await qualityAnalyzer.analyze(
  observation.value,
  { context: observation.context }
);
```

---

## 1.5 Practical Examples

### 1.5.1 Example 1: Code Quality Observation

```
// Input
const codeState = {
  code: `function calculate(x, y) { return x + y; }`,
  metrics: { complexity: 1, coverage: 0 }
};

// Execute
const observation = await observe(codeState);

// Output
{
  value: codeState,
  context: {
    quality: 0.60,
    issues: ["no error handling", "no type safety"],
    patterns: ["simple arithmetic"],
    suggestions: ["add TypeScript types", "add validation"]
  },
  confidence: 0.85
}
```

### 1.5.2 Example 2: Performance Observation

```
// Input
const perfState = {
  metrics: {
    responseTime: [120, 150, 180, 200],
    memoryUsage: [450, 480, 500, 520]
  }
};

// Execute
const observation = await observe(perfState);

// Output
{
  value: perfState,
  context: {
    trend: "degrading",
    analysis: "Response time increasing 20% over window",
    forecast: "Will exceed 300ms in 4 iterations",
    recommendations: ["profile hot paths", "check for memory leaks"]
  },
  confidence: 0.78
}
```

---

## 1.6 Categorical Laws Verified

All comonad laws verified with property-based testing:

1. **Left Identity**: `extract(duplicate(w)) ≡ w`
2. **Right Identity**: `fmap(extract, duplicate(w)) ≡ w`
3. **Associativity**: `duplicate(duplicate(w)) ≡ fmap(duplicate, duplicate(w))`

**Status**: ✅ 18/18 tests passing

---

# 1.7 Command-Line Usage

```
# Basic observation
cc2 observe <system-state.json>

# With options
cc2 observe --depth=deep --metrics <state.json>

# Pipe to REASON
cc2 observe <state.json> | cc2 reason

# Full chain
cc2 observe <state.json> | cc2 reason | cc2 create
```

---

# 1.8 Performance

- **Cold Start**: <100ms
- **Warm Execution**: <50ms
- **Memory**: <100MB
- **Scalability**: O(n) for state size

---

# 1.9 Limitations

- Requires structured input (JSON or TypeScript objects)
- Context window limited to 10K tokens
- Temporal analysis limited to provided data
- No external API calls (self-contained)

## 1.10 Related Skills

- **cc2-reason**: Use OBSERVE output for strategic planning
- **cc2-create**: Use observations to guide code generation
- **cc2-meta-orchestrator**: Orchestrate OBSERVE with other functions

## 1.11 References

- Implementation: `~/cc2.0/src/functions/observe/`
- Tests: `~/cc2.0/src/functions/observe/__tests__/`
- Documentation: `~/cc2.0/functions/observe/FUNCTION.md`
- Integration: `~/cc2.0/functions/observe/cc_integration.md`

## 1.12 Meta-Function: OBSERVE_SELF

OBSERVE can observe itself:

```
// Meta-observation
const metaObs = await observeSelf(observationHistory);

// Returns insights about observation patterns
{
  patterns: ["quality focus", "performance monitoring"],
  effectiveness: 0.87,
  suggestions: ["add security observations", "increase context depth"]
}
```

See: `~/cc2.0/functions/observe/modules/OBSERVE_SELF.md`