

Research Topic: Concurrent Session Architecture Limitations in Claude Code Governance

Document Type: Technical Documentation

Generated: October 13, 2025

Tractatus AI Safety Framework

<https://agenticgovernance.digital>

Research Topic: Concurrent Session Architecture Limitations in Claude Code Governance

Status: Discovered Design Constraint **Priority:** Medium **Classification:** Single-Tenant Architecture Limitation **First Identified:** October 2025 (Phase 4) **Related To:** Session state management, framework health metrics, test isolation **Scope:** Concurrent Claude Code sessions

Executive Summary

A significant architectural constraint was discovered during production testing: **the Tractatus framework assumes single-session, single-instance operation**. When multiple Claude Code instances govern the same codebase concurrently, several failure modes emerge:

1. **Contaminated health metrics** (token usage, message counts, pressure scores blend across sessions)
2. **Race conditions in instruction storage** (concurrent writes to `.claude/instruction-history.json`)
3. **Test isolation failures** (concurrent test runs conflict on shared database)
4. **Session state corruption** (last-write-wins on `.claude/session-state.json`)
5. **Inaccurate checkpoint triggers** (blended token counts fire alerts at wrong thresholds)

This is a design constraint, not a bug. The framework was architected for single-developer, single-session workflows—a valid design choice for Phase 1 prototyping. However, this reveals an important limitation for enterprise deployment where multiple developers might use AI governance concurrently on shared codebases.

Discovery method: Dogfooding during production testing when two concurrent sessions were inadvertently run, producing MongoDB duplicate key errors and invalid health metrics.

Good news: This is addressable through multi-tenant architecture patterns (session-specific state files, database-backed state, file locking). However, these capabilities are not yet implemented.

1. The Problem

1.1 Architectural Assumption: Single Session

Framework Design (Phase 1-4):

```
Assumption: ONE Claude Code instance governs codebase at a time
Architecture: Shared state files in .claude/ directory
State persistence: File-based JSON (no locking)
Session identification: Static session ID, manually updated
```

Why This Was Reasonable:

- Phase 1 prototype (research demonstration)
- Solo developer workflow (original use case)
- Simplified implementation (no concurrency complexity)
- Faster development (avoid distributed systems problems)

Where It Breaks:

- Multiple developers using AI governance concurrently
- Production testing while development continues
- Automated CI/CD with AI agents
- Parallel task execution

1.2 Discovered During Production Testing

Scenario: Two Claude Code sessions running concurrently on same codebase

Session A: Production test suite execution (`npm test`) **Session B:** Development work on elevator pitch documentation

Observed Failure: MongoDB duplicate key errors

```
MongoServerError: E11000 duplicate key error collection:
tractatus_prod.documents index: slug_1 dup key:
{ slug: "test-document-integration" }
```

Root Cause: Both sessions running test suites simultaneously, both attempting to create test documents with identical slugs, test cleanup race conditions preventing proper teardown.

Contamination Indicator: Session health metrics became meaningless—token counts, message counts, and pressure scores blended from both conversations, making framework health assessment unreliable.

2. Technical Analysis

2.1 Shared State Files

Files Affected:

```
.claude/instruction-history.json      (18 instructions, ~355 lines)
.claude/session-state.json           (Framework activity tracking)
.claude/token-checkpoints.json       (Milestone monitoring)
```

Problem: No File Locking

```
// Simplified pseudo-code showing vulnerability
function addInstruction(newInstruction) {
  // Session A reads file
  const history = JSON.parse(fs.readFileSync('instruction-history.json'));

  // Session B reads file (same state)
  const history = JSON.parse(fs.readFileSync('instruction-history.json'));

  // Session A adds instruction, writes back
  history.push(instructionA);
  fs.writeFileSync('instruction-history.json', JSON.stringify(history));

  // Session B adds instruction, writes back (overwrites A's change!)
  history.push(instructionB);
  fs.writeFileSync('instruction-history.json', JSON.stringify(history));

  // Result: instructionA is LOST (classic write conflict)
}
```

Impact: Last-write-wins behavior, instruction additions can be silently lost.

Frequency: Low under normal use (instruction additions are infrequent), but probabilistically guaranteed under concurrent operation.

2.2 Session State Contamination

Session State Structure (`.claude/session-state.json`):

```
{
  "session_id": "2025-10-07-001",
  "created_at": "2025-10-07T12:00:00Z",
  "token_budget": 200000,
  "messages": 42,
  "framework_activity": {
    "pressure_checks": 3,
    "instructions_added": 2,
    "validations_run": 15,
    "boundary_enforcements": 1
  }
}
```

Concurrent Session Behavior:

- Session A: 42 messages, 85,000 tokens
- Session B: 18 messages, 32,000 tokens
- **Blended state:** 60 messages, 117,000 tokens (meaningless)

Pressure Score Contamination:

```
Session A calculates: 85,000 / 200,000 = 42.5% (ELEVATED)
Session B reads blended: 117,000 / 200,000 = 58.5% (HIGH)
Session B incorrectly triggers handoff recommendation!
```

Impact: Framework health metrics become unreliable, checkpoint triggers fire at incorrect thresholds, context pressure monitoring fails to serve its purpose.

2.3 Test Isolation Failures

Test Suite Design:

```
// tests/integration/api.documents.test.js
beforeEach(async () => {
  // Create test document
  await db.collection('documents').insertOne({
    slug: 'test-document-integration', // Static slug
    title: 'Test Document',
    // ...
  });
});

afterEach(async () => {
  // Clean up test document
  await db.collection('documents').deleteOne({
    slug: 'test-document-integration'
  });
});
```

Concurrent Session Behavior:

Time	Session A	Session B
T0	Insert test-document-integration	
T1		Insert test-document-integration (FAIL: E11000 duplicate key)
T2	Run tests...	
T3		Delete test-document-integration
T4	Expect document exists (FAIL: document deleted by B!)	

Impact: Test failures not related to actual bugs, unreliable CI/CD, false negatives in quality checks.

Observed: 29 tests failing on production with concurrent sessions vs. 1 failing locally (single session).

2.4 Session Identity Confusion

Current Implementation:

```
// scripts/session-init.js
const SESSION_ID = '2025-10-07-001'; // Static, manually updated
```

Problem: Both concurrent sessions share same session ID

Impact:

- Framework logs ambiguous (can't attribute actions to sessions)
 - Instruction history shows mixed provenance
 - Debugging concurrent issues impossible
 - Audit trail unreliable
-

3. Framework Health Metrics Impact

3.1 Metrics Compromised by Concurrency

Token Usage Tracking:

- **✗ Contaminated:** Sum of both sessions
- **✗ Checkpoint triggers:** Fire at wrong thresholds
- **✗ Budget management:** Neither session knows true usage
- **Reliability:** 0% (completely unreliable)

Message Count Tracking:

- **✗ Contaminated:** Combined message counts
- **✗ Session length assessment:** Meaningless
- **✗ Complexity scoring:** Blended contexts
- **Reliability:** 0% (completely unreliable)

Context Pressure Score:

- **✗ Contaminated:** Weighted average of unrelated contexts

- **✗ Handoff triggers:** May fire prematurely or miss degradation
- **✗ Session health assessment:** Unreliable
- **Reliability:** 0% (completely unreliable)

Error Frequency:

- **⚠ Partially contaminated:** Combined error counts
- **⚠ Error attribution:** Can't determine which session caused errors
- **⚠ Pattern detection:** Mixed signal obscures real patterns
- **Reliability:** 30% (error detection works, attribution doesn't)

Task Complexity:

- **⚠ Partially contaminated:** Sum of concurrent tasks
- **⚠ Complexity scoring:** Appears artificially high
- **Reliability:** 40% (detects high complexity, can't attribute)

3.2 Metrics Unaffected by Concurrency

Test Suite Pass Rate:

- **✅ Database-backed:** Reflects actual system state
- **✅ Objectively measurable:** Independent of session state
- **Reliability:** 100% (fully reliable)
- **Note:** Pass rate itself reliable, but concurrent test execution causes failures

Framework Component Operational Status:

- **✅ Process-local verification:** Each session verifies independently
- **✅ Component availability:** Reflects actual system capabilities
- **Reliability:** 100% (fully reliable)

Instruction Database Content:

- **⚠ Eventually consistent:** Despite write conflicts, instructions persist
- **⚠ Audit trail:** Provenance may be ambiguous
- **Reliability:** 85% (content reliable, provenance uncertain)

3.3 Real-World Impact Example

Observed Scenario (October 2025):

Session A (Production Testing):

- Messages: 8
- Tokens: 29,414
- Pressure: Should be 14.7% (NORMAL)
- Action: Continue testing

Session B (Development):

- Messages: 42
- Tokens: 85,000
- Pressure: Should be 42.5% (ELEVATED)
- Action: Monitor, prepare for potential handoff

Blended State (What Both Sessions See):

- Messages: 50
- Tokens: 114,414
- Pressure: 57.2% (HIGH)
- Action: RECOMMEND HANDOFF (incorrect for both!)

Impact: Session A incorrectly warned about context pressure, Session B unaware of actual elevated pressure. Framework health monitoring counterproductive instead of helpful.

4. Why This Wasn't Caught Earlier

4.1 Development Workflow Patterns

Phase 1-3 Development (Solo workflow):

- Single developer
- Sequential sessions
- One task at a time
- Natural session boundaries

Result: Architectural assumption validated by usage pattern (no concurrent sessions in practice).

4.2 Test Suite Design

Current Testing:

- Unit tests (isolated, no state conflicts)
- Integration tests (assume exclusive database access)
- No concurrency testing
- No multi-session scenarios

Gap: Tests validate framework components work, but don't validate architectural assumptions about deployment model.

4.3 Dogfooding Discovery

How Discovered:

- Production test suite running in one terminal
- Concurrent development session for documentation
- Both sessions accessing shared state files
- MongoDB duplicate key errors surfaced the conflict

Lesson: Real-world usage patterns reveal architectural constraints that design analysis might miss.

Validation: This is exactly what dogfooding is designed to catch—real-world failure modes that theoretical analysis overlooks.

5. Architectural Design Space

5.1 Current Architecture: Single-Tenant

Design:

Codebase

```
└─ .claude/  
    └─ instruction-history.json (shared)  
    └─ session-state.json      (shared)  
    └─ token-checkpoints.json  (shared)
```

Claude Code Instance → Reads/Writes shared files

Assumptions:

- ONE instance active at a time
- Sequential access pattern
- File-based state sufficient
- Manual session ID management

Strengths:

- Simple implementation
- Fast development
- No distributed systems complexity
- Appropriate for Phase 1 prototype

Weaknesses:

- No concurrency support
- Race conditions on writes
- Contaminated metrics
- Test isolation failures

5.2 Alternative: Multi-Tenant Architecture

Design:

```

Codebase
└─ .claude/
  └─ instruction-history.json      (shared, READ-ONLY)
    └─ sessions/
      └─ session-abc123/
        └─ state.json
        └─ checkpoints.json
      └─ session-xyz789/
        └─ state.json
        └─ checkpoints.json

```

```

Claude Code Instance (Session ABC123)
→ Reads shared instruction-history.json
→ Writes session-specific state files

```

Capabilities:

- Multiple concurrent instances
- Session-isolated state
- Accurate per-session metrics
- Instruction history still shared (with locking)

Implementation Requirements:

1. Unique session ID generation (UUID)
2. Session-specific state directory
3. File locking for shared instruction writes
4. Session lifecycle management (cleanup old sessions)
5. Aggregated metrics (if needed)

Complexity: Moderate (2-3 weeks implementation)

5.3 Alternative: Database-Backed State

Design:

MongoDB Collections:

- instructions (shared, indexed)
- sessions (session metadata)
- session_state (session-specific state)
- token_checkpoints (session-specific milestones)

Claude Code Instance

- Reads from MongoDB (supports concurrent reads)
- Writes with transaction support (ACID guarantees)

Capabilities:

- True multi-tenant support
- Transactional consistency
- Query capabilities (aggregate metrics, audit trails)
- Horizontal scaling

Implementation Requirements:

1. Database schema design
2. Migration from file-based to DB-backed state
3. Transaction handling
4. Connection pooling
5. State synchronization

Complexity: High (4-6 weeks implementation)

5.4 Alternative: Distributed Lock Service

Design:

```
Shared State Files (existing)
  + File locking layer (flock, lockfile library)
  OR
  + Redis-based distributed locks

Claude Code Instance
  → Acquires lock before state operations
  → Releases lock after write
  → Handles lock timeouts and contention
```

Capabilities:

- Prevents write conflicts
- Maintains file-based state
- Minimal architectural change

Implementation Requirements:

1. Lock acquisition/release wrapper
2. Deadlock prevention
3. Lock timeout handling
4. Stale lock cleanup

Complexity: Low-Moderate (1-2 weeks implementation)

6. Impact Assessment

6.1 Who Is Affected?

NOT Affected:

- Solo developers using single Claude Code session
- Sequential development workflows
- Current Tractatus development (primary use case)
- Organizations with strict turn-taking on AI usage

Affected:

- Teams with multiple developers using AI governance concurrently
- Production environments with automated testing + development
- CI/CD pipelines with parallel AI-assisted jobs
- Organizations expecting true multi-user AI governance

Severity by Scenario:

Scenario	Impact	Workaround Available?
Solo developer	None	N/A (works as designed)
Team, coordinated usage	Low	Yes (take turns)
Concurrent dev + CI/CD	Medium	Yes (isolate test DB)
True multi-tenant need	High	No (requires architecture change)

6.2 Current Tractatus Deployment

Status: Single-developer, single-session usage **Impact:** None (architectural assumption matches usage pattern) **Risk:** Low for current Phase 1-4 scope

Future Risk:

- Phase 5+: If multi-developer teams adopt framework
- Enterprise deployment: If concurrent AI governance expected
- Scale testing: If parallel sessions needed for research

6.3 Enterprise Deployment Implications

Question: Can Tractatus scale to enterprise teams (10-50 developers)?

Current Answer: Not without architectural changes

Requirements for Enterprise:

1. Multi-session support (multiple developers concurrently)
2. Session isolation (independent health metrics)

3. Shared instruction history (organizational learning)
4. Audit trails (who added which instruction, when)
5. Concurrent test execution (CI/CD pipelines)

Gap: Current architecture supports #3 partially, not #1, #2, #4, #5

7. Mitigation Strategies

7.1 Current Workarounds (No Code Changes)

Workaround 1: Coordinated Usage

- **Approach:** Only one developer uses Claude Code at a time
- **Implementation:** Team agreement, Slack status, mutex file
- **Pros:** Zero code changes, works immediately
- **Cons:** Doesn't scale, manual coordination overhead, limits parallel work

Workaround 2: Isolated Test Databases

- **Approach:** Development and testing use separate databases
- **Implementation:** Environment-specific DB names
- **Pros:** Prevents test collision, easy to implement
- **Cons:** Doesn't solve state contamination, partial solution only

Workaround 3: Session Serialization

- **Approach:** Stop all Claude Code sessions before starting new one
- **Implementation:** `pkill` Claude Code processes, verify before starting
- **Pros:** Guarantees single session, no conflicts
- **Cons:** Disruptive, prevents parallelism, manual process

7.2 Short-Term Solutions (Minimal Code)

Solution 1: Session-Specific State Directories

- **Approach:** Implement multi-tenant architecture (Section 5.2)

- **Effort:** 2-3 weeks development
- **Benefits:** Concurrent sessions, isolated metrics, no contamination
- **Risks:** State directory cleanup, session lifecycle management

Solution 2: File Locking Layer

- **Approach:** Add distributed locks (Section 5.4)
- **Effort:** 1-2 weeks development
- **Benefits:** Prevents write conflicts, preserves file-based architecture
- **Risks:** Lock contention, timeout handling, debugging complexity

7.3 Long-Term Solutions (Architectural)

Solution 3: Database-Backed State

- **Approach:** Migrate to MongoDB-backed state (Section 5.3)
- **Effort:** 4-6 weeks development
- **Benefits:** True multi-tenant, transactional, scalable, queryable
- **Risks:** Migration complexity, backward compatibility, DB dependency

Solution 4: Hybrid Approach

- **Approach:** Shared instruction history (DB), session state (files)
- **Effort:** 3-4 weeks development
- **Benefits:** Balances consistency needs with simplicity
- **Risks:** Two state management systems to maintain

8. Research Questions

8.1 Fundamental Questions

1. **What is the expected concurrency level for AI governance frameworks?**
 - Hypothesis: 2-5 concurrent sessions for small teams, 10-20 for enterprise
 - Method: User studies, enterprise deployment analysis
 - Timeframe: 6-9 months

2. Does multi-session governance create new failure modes beyond state contamination?

- Hypothesis: Yes—instruction conflicts, inconsistent enforcement, coordination overhead
- Method: Controlled experiments with concurrent sessions
- Timeframe: 3-6 months

3. What metrics need to be session-specific vs. aggregate?

- Hypothesis: Context pressure session-specific, instruction effectiveness aggregate
- Method: Multi-session deployment, metric analysis
- Timeframe: 6 months

8.2 Architectural Questions

4. Is file-based state inherently incompatible with multi-tenant AI governance?

- Hypothesis: No, with proper locking mechanisms
- Method: Implement file locking, test under load
- Timeframe: 3 months

5. What are the performance characteristics of DB-backed state vs. file-based?

- Hypothesis: DB-backed has higher latency but better consistency
- Method: Benchmark tests, load testing
- Timeframe: 2 months

6. Can session isolation preserve organizational learning?

- Hypothesis: Yes, if instruction history shared but session state isolated
- Method: Multi-tenant architecture implementation
- Timeframe: 6 months

8.3 Practical Questions

7. At what team size does single-session coordination become impractical?

- Hypothesis: 3-5 developers
- Method: Team workflow studies
- Timeframe: 6 months

8. Do concurrent sessions require different governance rules?

- Hypothesis: Yes—coordination rules, conflict resolution, priority mechanisms
 - Method: Multi-session governance experiments
 - Timeframe: 9 months
-

9. Comparison to Related Systems

9.1 Git (Distributed Version Control)

Concurrency Model: Optimistic concurrency, merge conflict resolution **State Management:** Distributed (each developer has full repo) **Conflict Resolution:** Manual merge, automated for non-conflicting changes **Lesson:** Even file-based systems can support concurrency with proper design

Tractatus Difference: Git merges are explicit, Tractatus state updates implicit **Takeaway:** Could Tractatus adopt merge-based conflict resolution?

9.2 Database Systems

Concurrency Model: ACID transactions, row-level locking **State Management:** Centralized, transactional **Conflict Resolution:** Locks, isolation levels, optimistic concurrency **Lesson:** Centralized state enables strong consistency guarantees

Tractatus Difference: File-based state lacks transactional guarantees **Takeaway:** Database-backed state natural fit for multi-session needs

9.3 Collaborative Editing (Google Docs, VS Code Live Share)

Concurrency Model: Operational transformation, CRDTs (conflict-free replicated data types) **State Management:** Real-time synchronization **Conflict Resolution:** Automatic, character-level merging **Lesson:** Real-time collaboration requires sophisticated conflict resolution

Tractatus Difference: Session state doesn't require character-level merging **Takeaway:** Simpler conflict models (last-write-wins with versioning) might suffice

9.4 Kubernetes (Distributed System Orchestration)

Concurrency Model: Leader election, etcd for distributed state **State Management:** Distributed consensus (Raft protocol) **Conflict Resolution:** Strong consistency, leader serializes writes

Lesson: Distributed systems require consensus for correctness

Tractatus Difference: Framework doesn't need distributed consensus (codebase is single source of truth) **Takeaway:** File locking or DB transactions sufficient, don't need Raft/Paxos

10. Honest Assessment

10.1 Is This a Fatal Flaw?

No. Single-tenant architecture is:

- A valid design choice for Phase 1 prototype
- Appropriate for solo developer workflows
- Simpler to implement and maintain
- Not unique to Tractatus (many tools assume single user)

But: It's a limitation for enterprise deployment and team usage.

10.2 When Does This Become Critical?

Timeline:

- **Now** (Phase 1-4): Not critical (solo developer workflow)
- **Phase 5-6** (6-12 months): May need multi-session if teams adopt
- **Enterprise deployment:** Critical requirement for organizational use
- **Research experiments:** Needed for scalability testing

Conclusion: We have 6-12 months before this becomes a blocking issue

10.3 Why Be Transparent About This?

Reason 1: User Expectations Organizations evaluating Tractatus should know deployment constraints

Reason 2: Research Contribution Other AI governance frameworks will face concurrency challenges

Reason 3: Tractatus Values Honesty about limitations builds more trust than hiding them

Reason 4: Design Trade-offs Single-tenant architecture enabled faster prototype development—valid trade-off for research phase

11. Recommendations

11.1 For Current Tractatus Users

Immediate (Next session):

- Use workaround: Stop concurrent sessions before production testing
- Isolate test databases (development vs. testing)
- Coordinate AI usage in team settings

Short-term (1-3 months):

- Implement session-specific state directories (Phase 5)
- Add unique session ID generation
- Test suite improvements (randomized slugs, better cleanup)

Medium-term (3-12 months):

- Evaluate need for multi-session support based on user adoption
- Research DB-backed state vs. file locking trade-offs
- Implement chosen multi-tenant architecture if needed

11.2 For Organizations Evaluating Tractatus

Be aware:

- Current architecture assumes single Claude Code session
- Concurrent sessions cause state contamination and test failures
- Workarounds available (coordinated usage, isolated databases)
- Multi-tenant architecture planned but not implemented

Consider:

- Is single-session coordination acceptable for your team size?
- Do you need concurrent AI governance? (most teams: no)
- Can you contribute to multi-session architecture development?

11.3 For AI Governance Researchers

Research Opportunities:

- Multi-session governance coordination protocols
- Session-specific vs. aggregate metrics
- Concurrent instruction addition conflict resolution
- Optimistic vs. pessimistic concurrency for AI state





Collaborate on:

- Multi-tenant architecture design patterns
- Concurrency testing methodologies
- Enterprise deployment case studies

12. Conclusion

The Tractatus framework's **single-tenant architecture** is a **design constraint, not a defect**. It was appropriate for Phase 1-4 prototype development but represents a limitation for enterprise deployment.

Key Findings:

-  **Discovered through dogfooding:** Real-world usage revealed architectural assumption
-  **Well-understood:** Root causes clear, mitigation strategies identified
-  **Addressable:** Multiple architectural solutions available (multi-tenant, DB-backed, file locking)
-  **Not yet implemented:** Current framework doesn't support concurrent sessions

Current Status:

- Works reliably for single-session workflows
- Contamination occurs with concurrent sessions
- Workarounds available (coordination, isolation)

Future Direction:

- Multi-tenant architecture (Phase 5-6, if user adoption requires)
- Research on concurrent AI governance coordination
- Evaluation of DB-backed vs. file-based state trade-offs

Transparent Takeaway: Tractatus is effective for solo developers and coordinated teams, has known concurrency limitations, has planned architectural solutions if enterprise adoption requires them.

This is the value of dogfooding: discovering real constraints through actual use, not theoretical speculation.

13. Appendix: Technical Discovery Details

13.1 Observed Error Sequence

Production Test Execution (October 9, 2025):

```
# Session A: Production testing
npm test
# 29 tests failing (duplicate key errors)

# Session B: Development work
# (concurrent documentation edits)

# Conflict manifestation:
MongoServerError: E11000 duplicate key error collection:
tractatus_prod.documents index: slug_1 dup key:
{ slug: "test-document-integration" }
```

Analysis:

- Both sessions running `npm test` simultaneously

- Test setup: Insert document with static slug
- Race condition: Both sessions attempt insert
- MongoDB constraint: Unique index on slug field
- Result: E11000 duplicate key error

Lesson: Concurrent test execution requires randomized identifiers or session-specific test data.

13.2 Session State Comparison

Expected (Session A only):

```
{
  "session_id": "2025-10-07-001",
  "messages": 8,
  "tokens_used": 29414,
  "pressure_score": 14.7,
  "status": "NORMAL"
}
```

Observed (Concurrent A + B):

```
{
  "session_id": "2025-10-07-001",
  "messages": 50,
  "tokens_used": 114414,
  "pressure_score": 57.2,
  "status": "HIGH"
}
```

Impact: Framework health assessment unreliable, checkpoint triggers fire incorrectly.

13.3 File Write Conflict Timeline

```
T0: Session A reads instruction-history.json (18 instructions)
T1: Session B reads instruction-history.json (18 instructions)
T2: Session A adds inst_019, writes file (19 instructions)
T3: Session B adds inst_020, writes file (19 instructions)
T4: File contains inst_020 only (inst_019 lost!)
```

Probability: Low under normal use, 100% guaranteed under heavy concurrent writes.

Mitigation: File locking or atomic operations required.

Document Version: 1.0 **Research Priority:** Medium **Next Review:** Phase 5 planning (or when multi-session need identified) **Status:** Open research topic, community contributions welcome
Scope: Claude Code concurrent session governance

Related Resources:

- [Rule Proliferation Research](#)
- [Framework in Action Case Study](#)
- `.claude/session-state.json` - Current state structure
- `scripts/session-init.js` - Session initialization

Future Research:

- Multi-tenant architecture design (Phase 5-6)
- Database-backed state migration (Phase 6-7)
- Concurrent session coordination protocols (Phase 7)
- Optimistic concurrency control for instruction history (Phase 6)

Contributions: See CONTRIBUTING.md (to be created in GitHub repository)

Anonymization: All identifying information (server IPs, personal names, organizational details) removed. Technical details preserved for research value.

Document Metadata

- **Version:** 1.0
 - **Created:** 2025-10-09
 - **Last Modified:** 2025-10-13
 - **Author:** Tractatus Framework Research Team
 - **Word Count:** 6,674 words
 - **Reading Time:** ~33 minutes
 - **Document ID:** concurrent-session-architecture-limitations
 - **Status:** Discovered Design Constraint
 - **Document Type:** Research Analysis
-

License

Copyright 2025 John Stroh

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at:

<http://www.apache.org/licenses/LICENSE-2.0>

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

Additional Terms:

1. **Attribution Requirement:** Any use, modification, or distribution of this work must include clear attribution to the original author and the Tractatus Framework project.
2. **Moral Rights:** The author retains moral rights to the work, including the right to be identified as the author and to object to derogatory treatment of the work.
3. **Research and Educational Use:** This work is intended for research, educational, and practical implementation purposes. Commercial use is permitted under the terms of the Apache 2.0 license.

4. **No Warranty:** This work is provided "as is" without warranty of any kind, express or implied. The author assumes no liability for any damages arising from its use.

5. **Community Contributions:** Contributions to this work are welcome and should be submitted under the same Apache 2.0 license terms.

For questions about licensing, please contact the author through the project repository.

© 2025 Tractatus AI Safety Framework

This document is part of the Tractatus Agentic Governance System

<https://agenticgovernance.digital>