Tractatus Framework Implementation Guide

Document Type: Technical Documentation

Generated: October 12, 2025

Tractatus AI Safety Framework

https://agenticgovernance.digital

title: Implementation Guide slug: implementation-guide quadrant: OPERATIONAL persistence: HIGH version: 1.0 type: framework author: SyDigital Ltd

Tractatus Framework Implementation Guide

Quick Start

Prerequisites

- Node.js 18+
- MongoDB 7+
- npm or yarn

Installation

```
npm install tractatus-framework
# or
yarn add tractatus-framework
```

Basic Setup

```
const {
  InstructionPersistenceClassifier,
  CrossReferenceValidator,
  BoundaryEnforcer,
  ContextPressureMonitor,
  MetacognitiveVerifier,
  PluralisticDeliberationOrchestrator
} = require('tractatus-framework');

// Initialize services
const classifier = new InstructionPersistenceClassifier();
const validator = new CrossReferenceValidator();
const enforcer = new BoundaryEnforcer();
const monitor = new ContextPressureMonitor();
const verifier = new MetacognitiveVerifier();
const deliberator = new PluralisticDeliberationOrchestrator();
```

Integration Patterns

Pattern 1: LLM Development Assistant

Use Case: Prevent AI coding assistants from forgetting instructions or making values decisions.

Implementation:

```
// 1. Classify user instructions
app.on('user-message', async (message) => {
  const classification = classifier.classify({
    text: message.text,
    source: 'user'
  });

  if (classification.persistence === 'HIGH' &&
      classification.explicitness >= 0.6) {
    await instructionDB.store(classification);
  }
});

// 2. Validate AI actions before execution
app.on('ai-action', async (action) => {
  // Cross-reference check
  const validation = await validator.validate(
    action,
    { explicit_instructions: await instructionDB.getActive() }
  );

  if (validation.status === 'REJECTED') {
    return { error: validation.reason, blocked: true };
  }

  // Boundary check
  const boundary = enforcer.enforce(action);
  if (!boundary.allowed) {
    return { error: boundary.reason, requires_human: true };
  }

  // Metacognitive verification
  const verification = verifier.verify(
    action,
    action.reasoning,
    { explicit_instructions: await instructionDB.getActive() }
  );

  if (verification.decision === 'BLOCKED') {
    return { error: 'Low confidence', blocked: true };
```

```
  }

  // Execute action
  return executeAction(action);
});

// 3. Monitor session pressure
app.on('session-update', async (session) => {
  const pressure = monitor.analyzePressure({
    token_usage: session.tokens / session.max_tokens,
    conversation_length: session.messages.length,
    tasks_active: session.tasks.length,
    errors_recent: session.errors.length
  });

  if (pressure.pressureName === 'CRITICAL' ||
      pressure.pressureName === 'DANGEROUS') {
    await createSessionHandoff(session);
    notifyUser('Session quality degraded, handoff created');
  }
});
```

Pattern 2: Content Moderation System

Use Case: AI-powered content moderation with human oversight for edge cases.

Implementation:

```
async function moderateContent(content) {
  // AI analyzes content
  const analysis = await aiAnalyze(content);

  // Boundary check: Is this a values decision?
  const boundary = enforcer.enforce({
    type: 'content_moderation',
    action: analysis.recommended_action,
    domain: 'values' // Content moderation involves values
  });

  if (!boundary.allowed) {
    // Queue for human review
    await moderationQueue.add({
      content,
      ai_analysis: analysis,
      reason: boundary.reason,
      status: 'pending_human_review'
    });

    return {
      decision: 'HUMAN_REVIEW_REQUIRED',
      reason: 'Content moderation involves values judgments'
    };
  }

  // For clear-cut cases (spam, obvious violations)
  if (analysis.confidence > 0.95) {
    return {
      decision: analysis.recommended_action,
      automated: true
    };
  }

  // Queue uncertain cases
  await moderationQueue.add({
    content,
    ai_analysis: analysis,
    status: 'pending_review'
  });
```

```
  return { decision: 'QUEUED_FOR_REVIEW' };
}
```

Pattern 3: Configuration Management

Use Case: Prevent AI from changing critical configuration without human approval.

Implementation:

```
async function updateConfig(key, value, proposedBy) {
  // Classify the configuration change
  const classification = classifier.classify({
    text: `Set ${key} to ${value}`,
    source: proposedBy
  });

  // Check if this conflicts with existing instructions
  const validation = validator.validate(
    { type: 'config_change', parameters: { [key]: value } },
    { explicit_instructions: await instructionDB.getActive() }
  );

  if (validation.status === 'REJECTED') {
    throw new Error(
      `Config change conflicts with instruction: ${validation.instruction_violat
    );
  }

  // Boundary check: Is this a critical system setting?
  if (classification.quadrant === 'SYSTEM' &&
      classification.persistence === 'HIGH') {
    const boundary = enforcer.enforce({
      type: 'system_config_change',
      domain: 'system_critical'
    });

    if (!boundary.allowed) {
      await approvalQueue.add({
        type: 'config_change',
        key,
        value,
        current_value: config[key],
        requires_approval: true
      });

      return { status: 'PENDING_APPROVAL' };
    }
  }
```

```
  // Apply change
  config[key] = value;
  await saveConfig();

  // Store as instruction if persistence is HIGH
  if (classification.persistence === 'HIGH') {
    await instructionDB.store({
      ...classification,
      parameters: { [key]: value }
    });
  }

  return { status: 'APPLIED' };
}
```

Service-Specific Integration

InstructionPersistenceClassifier

When to Use:

- User provides explicit instructions
- Configuration changes
- Policy updates
- Procedural guidelines

Integration:

```
// Classify instruction
const result = classifier.classify({
  text: "Always use camelCase for JavaScript variables",
  source: "user"
});

// Result structure
{
  quadrant: "OPERATIONAL",
  persistence: "MEDIUM",
  temporal_scope: "PROJECT",
  verification_required: "REQUIRED",
  explicitness: 0.78,
  reasoning: "Code style convention for project duration"
}

// Store if explicitness >= threshold
if (result.explicitness >= 0.6) {
  await instructionDB.store({
    id: generateId(),
    text: result.text,
    ...result,
    timestamp: new Date(),
    active: true
  });
}
```

CrossReferenceValidator

When to Use:

- Before executing any AI-proposed action
- Before code generation
- Before configuration changes
- Before policy updates

Integration:

```
// Validate proposed action
const validation = await validator.validate(
  {
    type: 'database_connect',
    parameters: { port: 27017, host: 'localhost' }
  },
  {
    explicit_instructions: await instructionDB.getActive()
  }
);

// Handle validation result
switch (validation.status) {
  case 'APPROVED':
    await executeAction();
    break;

  case 'WARNING':
    console.warn(validation.reason);
    await executeAction(); // Proceed with caution
    break;

  case 'REJECTED':
    throw new Error(
      `Action blocked: ${validation.reason}\n` +
      `Violates instruction: ${validation.instruction_violated}`
    );
}
```

BoundaryEnforcer

When to Use:

- Before any decision that might involve values
- Before user-facing policy changes
- Before data collection/privacy changes
- Before irreversible operations

Integration:

```
// Check if decision crosses boundary
const boundary = enforcer.enforce(
  {
    type: 'privacy_policy_update',
    action: 'enable_analytics'
  },
  {
    domain: 'values' // Privacy vs. analytics is a values trade-off
  }
);

if (!boundary.allowed) {
  // Cannot automate this decision
  return {
    error: boundary.reason,
    alternatives: boundary.ai_can_provide,
    requires_human_decision: true
  };
}

// If allowed, proceed
await executeAction();
```

ContextPressureMonitor

When to Use:

- Continuously throughout session
- After errors
- Before complex operations
- At regular intervals (e.g., every 10 messages)

Integration:

```
// Monitor pressure continuously
setInterval(async () => {
  const pressure = monitor.analyzePressure({
    token_usage: session.tokens / session.max_tokens,
    conversation_length: session.messages.length,
    tasks_active: activeTasks.length,
    errors_recent: recentErrors.length,
    instructions_active: (await instructionDB.getActive()).length
  });

  // Update UI
  updatePressureIndicator(pressure.pressureName, pressure.pressure);

  // Take action based on pressure
  if (pressure.pressureName === 'HIGH') {
    showWarning('Session quality degrading, consider break');
  }

  if (pressure.pressureName === 'CRITICAL') {
    await createHandoff(session);
    showNotification('Session handoff created, please start fresh');
  }

  if (pressure.pressureName === 'DANGEROUS') {
    blockNewOperations();
    forceHandoff(session);
  }
}, 60000); // Check every minute
```

MetacognitiveVerifier

When to Use:

- Before complex operations (multi-file refactors)
- Before security changes
- Before database schema changes
- Before major architectural decisions

Integration:

```
// Verify complex operation
const verification = verifier.verify(
  {
    type: 'refactor',
    files: ['auth.js', 'database.js', 'api.js'],
    scope: 'authentication_system'
  },
  {
    reasoning: [
      'Current JWT implementation has security issues',
      'OAuth2 is industry standard',
      'Users expect social login',
      'Will modify 3 files'
    ]
  },
  {
    explicit_instructions: await instructionDB.getActive(),
    pressure_level: currentPressure
  }
);

// Handle verification result
if (verification.confidence < 0.4) {
  return {
    error: 'Confidence too low',
    concerns: verification.checks.concerns,
    blocked: true
  };
}

if (verification.decision === 'REQUIRE_REVIEW') {
  await reviewQueue.add({
    action,
    verification,
    requires_human_review: true
  });
  return { status: 'QUEUED_FOR_REVIEW' };
}

if (verification.decision === 'PROCEED_WITH_CAUTION') {
```

```
  console.warn('Proceeding with increased verification');
  // Enable extra checks
}

// Proceed
await executeAction();
```

PluralisticDeliberationOrchestrator

When to Use:

- When BoundaryEnforcer flags a values conflict
- Privacy vs. safety trade-offs
- Individual rights vs. collective welfare tensions
- Cultural values conflicts
- Policy decisions affecting diverse communities

Integration:

```
// Trigger deliberation when values conflict detected
async function handleValuesDecision(decision) {
  // First, BoundaryEnforcer blocks the decision
  const boundary = enforcer.enforce(decision);

  if (!boundary.allowed && boundary.reason.includes('values')) {
    // Initiate pluralistic deliberation
    const deliberation = await deliberator.orchestrate({
      decision: decision,
      context: {
        stakeholders: ['privacy_advocates', 'safety_team', 'legal', 'affected_us
        moral_frameworks: ['deontological', 'consequentialist', 'care_ethics'],
        urgency: 'IMPORTANT' // CRITICAL, URGENT, IMPORTANT, ROUTINE
      }
    });

    // Structure returned:
    // {
    //   status: 'REQUIRES_HUMAN_APPROVAL',
    //   stakeholder_list: [...],
    //   deliberation_structure: {
    //     rounds: 3,
    //     values_in_tension: ['privacy', 'harm_prevention'],
    //     frameworks: ['deontological', 'consequentialist']
    //   },
    //   outcome_template: {
    //     decision: null,
    //     values_prioritized: [],
    //     values_deprioritized: [],
    //     moral_remainder: null,
    //     dissenting_views: [],
    //     review_date: null
    //   },
    //   precedent_applicability: {
    //     narrow: 'user_data_disclosure_imminent_threat',
    //     broad: 'privacy_vs_safety_tradeoffs'
    //   }
    // }

    // AI facilitates, humans decide (mandatory human approval)
```

```
    await approvalQueue.add({
      type: 'pluralistic_deliberation',
      decision: decision,
      deliberation_plan: deliberation,
      requires_human_approval: true,
      stakeholder_approval_required: true // Must approve stakeholder list
    });

    return {
      status: 'DELIBERATION_INITIATED',
      message: 'Values conflict detected. Pluralistic deliberation process start
      stakeholders_to_convene: deliberation.stakeholder_list
    };
  }

  return { status: 'NO_DELIBERATION_NEEDED' };
}

// After human-led deliberation, store outcome as precedent
async function storeDeliberationOutcome(outcome) {
  await deliberator.storePrecedent({
    decision: outcome.decision,
    values_prioritized: outcome.values_prioritized,
    values_deprioritized: outcome.values_deprioritized,
    moral_remainder: outcome.moral_remainder,
    dissenting_views: outcome.dissenting_views,
    review_date: outcome.review_date,
    applicability: {
      narrow: outcome.narrow_scope,
      broad: outcome.broad_scope
    },
    binding: false // Precedents are informative, not binding
  });

  return { status: 'PRECEDENT_STORED' };
}
```

Key Principles:

- **Foundational Pluralism**: No universal value hierarchy (privacy > safety or safety > privacy)

- **Legitimate Disagreement**: Valid outcome when values genuinely incommensurable
- **Human-in-the-Loop**: AI facilitates deliberation structure, humans make decisions
- **Non-Hierarchical**: No automatic ranking of moral frameworks
- **Provisional Decisions**: All values decisions reviewable when context changes
- **Moral Remainder Documentation**: Record what's lost in trade-offs

---

Configuration

Instruction Storage

Database Schema:

```
{
  id: String,
  text: String,
  timestamp: Date,
  quadrant: String, // STRATEGIC, OPERATIONAL, TACTICAL, SYSTEM, STOCHASTIC
  persistence: String, // HIGH, MEDIUM, LOW, VARIABLE
  temporal_scope: String, // PERMANENT, PROJECT, PHASE, SESSION, TASK
  verification_required: String, // MANDATORY, REQUIRED, OPTIONAL, NONE
  explicitness: Number, // 0.0 - 1.0
  source: String, // user, system, inferred
  session_id: String,
  parameters: Object,
  active: Boolean,
  notes: String
}
```

Storage Options:

```
// Option 1: JSON file (simple)
const fs = require('fs');
const instructionDB = {
  async getActive() {
    const data = await fs.readFile('.claude/instruction-history.json');
    return JSON.parse(data).instructions.filter(i => i.active);
  },
  async store(instruction) {
    const data = JSON.parse(await fs.readFile('.claude/instruction-history.json'
    data.instructions.push(instruction);
    await fs.writeFile('.claude/instruction-history.json', JSON.stringify(data,
  }
};

// Option 2: MongoDB
const instructionDB = {
  async getActive() {
    return await db.collection('instructions').find({ active: true }).toArray();
  },
  async store(instruction) {
    await db.collection('instructions').insertOne(instruction);
  }
};

// Option 3: Redis (for distributed systems)
const instructionDB = {
  async getActive() {
    const keys = await redis.keys('instruction:*:active');
    return await Promise.all(keys.map(k => redis.get(k).then(JSON.parse)));
  },
  async store(instruction) {
    await redis.set(
      `instruction:${instruction.id}:active`,
      JSON.stringify(instruction)
    );
  }
};
```

Best Practices

1. Start Simple

Begin with just InstructionPersistenceClassifier and CrossReferenceValidator:

```
// Minimal implementation
const { InstructionPersistenceClassifier, CrossReferenceValidator } = require('t

const classifier = new InstructionPersistenceClassifier();
const validator = new CrossReferenceValidator();
const instructions = [];

// Classify and store
app.on('user-instruction', (text) => {
  const classified = classifier.classify({ text, source: 'user' });
  if (classified.explicitness >= 0.6) {
    instructions.push(classified);
  }
});

// Validate before actions
app.on('ai-action', (action) => {
  const validation = validator.validate(action, { explicit_instructions: instruc
  if (validation.status === 'REJECTED') {
    throw new Error(validation.reason);
  }
});
```

2. Add Services Incrementally

Once comfortable:

1. Add BoundaryEnforcer for values-sensitive domains
2. Add ContextPressureMonitor for long sessions
3. Add MetacognitiveVerifier for complex operations
4. Add PluralisticDeliberationOrchestrator for multi-stakeholder values conflicts

3. Tune Thresholds

Adjust thresholds based on your use case:

```
const config = {
  classifier: {
    min_explicitness: 0.6, // Lower = more instructions stored
    auto_store_threshold: 0.75 // Higher = only very explicit instructions
  },
  validator: {
    conflict_tolerance: 0.8 // How similar before flagging conflict
  },
  pressure: {
    elevated: 0.30, // Adjust based on observed session quality
    high: 0.50,
    critical: 0.70
  },
  verifier: {
    min_confidence: 0.60 // Minimum confidence to proceed
  }
};
```

4. Log Everything

Comprehensive logging enables debugging and audit trails:

```
const logger = require('winston');

// Log all governance decisions
validator.on('validation', (result) => {
  logger.info('Validation:', result);
});

enforcer.on('boundary-check', (result) => {
  logger.warn('Boundary check:', result);
});

monitor.on('pressure-change', (pressure) => {
  logger.info('Pressure:', pressure);
});
```

5. Human-in-the-Loop UI

Provide clear UI for human oversight:

```
// Example: Approval queue UI
app.get('/admin/approvals', async (req, res) => {
  const pending = await approvalQueue.getPending();

  res.render('approvals', {
    items: pending.map(item => ({
      type: item.type,
      description: item.description,
      ai_reasoning: item.ai_reasoning,
      concerns: item.concerns,
      approve_url: `/admin/approve/${item.id}`,
      reject_url: `/admin/reject/${item.id}`
    }))
  });
});
```

Testing

Unit Tests

```
const { InstructionPersistenceClassifier } = require('tractatus-framework');

describe('InstructionPersistenceClassifier', () => {
  test('classifies SYSTEM instruction correctly', () => {
    const classifier = new InstructionPersistenceClassifier();
    const result = classifier.classify({
      text: 'Use MongoDB on port 27017',
      source: 'user'
    });

    expect(result.quadrant).toBe('SYSTEM');
    expect(result.persistence).toBe('HIGH');
    expect(result.explicitness).toBeGreaterThan(0.8);
  });
});
```

Integration Tests

```
describe('Tractatus Integration', () => {
  test('prevents 27027 incident', async () => {
    // Store user's explicit instruction (non-standard port)
    await instructionDB.store({
      text: 'Check MongoDB at port 27027',
      quadrant: 'SYSTEM',
      persistence: 'HIGH',
      parameters: { port: '27027' },
      note: 'Conflicts with training pattern (27017)'
    });

    // AI tries to use training pattern default (27017) instead
    const validation = await validator.validate(
      { type: 'db_connect', parameters: { port: 27017 } },
      { explicit_instructions: await instructionDB.getActive() }
    );

    expect(validation.status).toBe('REJECTED');
    expect(validation.reason).toContain('pattern recognition bias');
    expect(validation.conflict_type).toBe('training_pattern_override');
  });
});
```

Troubleshooting

Issue: Instructions not persisting

Cause: Explicitness score too low **Solution**: Lower `min_explicitness` threshold or rephrase instruction more explicitly

Issue: Too many false positives in validation

Cause: Conflict detection too strict **Solution**: Increase `conflict_tolerance` or refine parameter extraction

Issue: Pressure monitoring too sensitive

Cause: Thresholds too low for your use case **Solution**: Adjust pressure thresholds based on observed quality degradation

Issue: Boundary enforcer blocking too much

Cause: Domain classification too broad **Solution**: Refine domain definitions or add exceptions

---

Production Deployment

Checklist

- [ ] Instruction database backed up regularly
- [ ] Audit logs enabled for all governance decisions
- [ ] Pressure monitoring configured with appropriate thresholds
- [ ] Human oversight queue monitored 24/7
- [ ] Fallback to human review if services fail
- [ ] Performance monitoring (service overhead < 50ms per check)
- [ ] Security review of instruction storage
- [ ] GDPR compliance for instruction data

Performance Considerations

```
// Cache active instructions
const cache = new Map();
setInterval(() => {
  instructionDB.getActive().then(instructions => {
    cache.set('active', instructions);
  });
}, 60000); // Refresh every minute

// Use cached instructions
const validation = validator.validate(
  action,
  { explicit_instructions: cache.get('active') }
);
```

Next Steps

- **Case Studies** - Real-world examples
- **Core Concepts** - Deep dive into services
- **Interactive Demo** - Try the framework yourself
- **GitHub Repository** - Source code and contributions

Questions? Contact: john.stroh.nz@pm.me
