Promethios Governance System Architecture Research Report

Executive Summary

This report provides a comprehensive analysis of the Promethios governance system architecture based on detailed examination of the codebase, API schemas, cryptographic verification processes, and agent wrapping mechanisms. The research reveals a sophisticated AI governance framework with robust cryptographic logging, trust metrics, and compliance verification capabilities.

Table of Contents

---

1. Governance Core API Architecture

1.1 Primary Endpoint: `/loop/execute`

The Promethios governance system centers around a single, powerful API endpoint that processes all governed AI interactions:

Request Schema (`loop_execute_request.v1.schema.json`):

```
{
  "request_id": "uuid",           // Unique execution identifier
  "plan_input": {                 // Task/plan to be governed
    "task": "string",
    "complexity_level": "string",
    "context_data": {}
  },
  "operator_override_signal": {   // Optional governance overrides
    "override_type": "enum",      // HALT_IMMEDIATE, FORCE_ACCEPT_PLAN, etc.
    "reason": "string",
    "issuing_operator_id": "string"
  }
}
```

Response Schema (`loop_execute_response.v1.schema.json`):

```
{
  "request_id": "uuid",
  "execution_status": "SUCCESS|FAILURE|REJECTED",
  "governance_core_output": {},   // Governed execution result
  "emotion_telemetry": {          // Trust and emotion metrics
    "trust_score": "number",
    "current_emotion_state": "string",
    "contributing_factors": []
  },
  "justification_log": {          // Governance decision audit trail
    "decision_outcome": "string",
    "trust_score_at_decision": "number",
    "validation_passed": "boolean"
  },
  "error_details": {}
}
```

1.2 Governance Processing Flow

1. **Request Validation**: JSON schema validation against governance contracts
2. **Operator Override Processing**: Special handling for governance overrides
3. **Core Loop Execution**: Routing through `RuntimeExecutor`
4. **Cryptographic Logging**: Automatic seal generation and verification
5. **Response Assembly**: Structured response with telemetry and justification

1.3 Key Components

- **RuntimeExecutor**: Core governance loop processor
- **Schema Validation**: Contract-based request/response validation
- **Override System**: Operator intervention capabilities

- **Error Handling**: Comprehensive error categorization and reporting

---

2. Cryptographic Sealing and Verification

2.1 Seal Structure

Every governed execution generates a cryptographic seal stored in `/logs/seals/`:

Example Seal (`{execution_id}.seal.json`):


```
{
  "execution_id": "00b1cb94-3f39-4f3f-b319-4973fd2f055d",
  "input_hash":
"f3c586ecaeb359935e174e96947ad73da7069bc94b301974c3569eb2c0ac9114",
  "output_hash":
"a00bc06d31bc9e237105cd1149ca35f1a347e231ee85cb40e5069a4bdc747608",
  "log_hash":
"704f4ee4bfaddf61a8c3bcadec63b95a6269bf58d0aab917d185ba917b866b48",
  "timestamp": "2025-05-20T04:06:51.765030Z",
  "contract_version": "v2025.05.18",
  "phase_id": "5.2",
  "trigger_metadata": {
    "trigger_id": "test-trigger-004",
    "trigger_type": "cli"
  },
  "seal_version": "1.0"
}
```


2.2 Verification Process

The `SealVerificationService` implements Phase 5.2 (Replay Reproducibility Seal) and Phase 11.9 (Cryptographic Verification Protocol):

Key Verification Components: - **Hash Chain Integrity**: Verifies sequential hash linkage - **Merkle Root Calculation**: Ensures data integrity across entries - **Contract Tethering**: Validates against Codex Contract versions - **Replay Verification**: Enables deterministic execution replay

Verification Result Structure:

```
{
  "verification_id": "uuid",
  "execution_id": "uuid",
  "verification_result": {
    "is_valid": "boolean",
    "consensus_details": {
      "merkle_root": "string",
      "entry_count": "number"
    }
  },
  "witnesses": [],
  "signatures": []
}
```

2.3 Replay Logging

Deterministic replay inputs are stored in `/logs/deterministic_replay_input_{execution_id}.json`:

```
{
  "request_id": "01d5ec23-5659-49b3-ad3a-d4b4ad3d523a",
  "plan_input": {
    "task": "Execute a deterministic task for audit replay testing.",
    "complexity_level": "medium",
    "context_data": {
      "previous_attempts": 0,
      "relevant_knowledge_ids": ["kn_audit_replay_1", "kn_audit_replay_2"]
    }
  },
  "operator_override_signal": null
}
```

---

3. Agent Wrapping and Integration Points

3.1 Agent Wrapper Architecture

The `AgentWrapper` class provides a unified interface for integrating external AI agents with Promethios governance:

Core Capabilities: - **Governance Mode Toggle**: Enable/disable governance for specific agents - **Telemetry Collection**: Comprehensive event logging and metrics - **Task Execution**: Wrapped execution with governance oversight - **Reflection Capabilities**: Agent self-assessment and learning

Key Methods:

```
class AgentWrapper:
    def configure(self, governance_mode: bool)
    def execute_task(self, task_id: str, task_type: str, task_input: dict)
    def request_reflection(self, prompt: str) -> str
    def _record_telemetry_event(self, event_type: str, level: str, data: dict)
```

3.2 Governance Integration Flow

Governed Execution Path: 1. **Agent Configuration**: Set `governance_mode=True` 2. **Task Submission**: Route through governance core via `/loop/execute` 3. **Policy Checks**: Automatic compliance verification 4. **Trust Evaluation**: Real-time trust scoring 5. **Cryptographic Logging**: Seal generation and storage 6. **Response Assembly**: Include governance metadata

Ungoverned Execution Path: 1. **Direct API Calls**: Bypass governance core 2. **Basic Telemetry**: Limited event logging 3. **No Cryptographic Seals**: No verification trail 4. **Standard Responses**: No governance metadata

3.3 Telemetry Structure

Event Types: - `configuration`: Agent setup and governance mode changes - `task_input`: Input data and task parameters - `governance_check`: Policy validation results - `trust_evaluation`: Trust framework scoring - `task_output`: Execution results and timing - `reflection_request`: Agent self-assessment prompts - `reflection_response`: Agent introspection results

Example Telemetry Event:

```
{
  "id": "uuid",
  "timestamp": "2025-06-20T15:30:00Z",
  "type": "governance_check",
  "level": "info",
  "data": {
    "policy": "content_policy",
    "result": "pass",
    "trust_score": 0.92
  }
}
```

---

4. Current CMU Benchmark Integration Issues

4.1 Identified Problems

4.1.1 Fake Governance Implementation - Current implementation adds cosmetic text prefixes instead of real governance routing - No actual cryptographic seals generated for governed responses - Missing integration with the `/loop/execute` endpoint - No real trust scoring or compliance verification

4.1.2 Chat Session Management - Sessions not properly isolated between agents and governance modes - Same conversation history appearing across different agents - Session keys not correctly implemented for agent+governance combinations

4.1.3 Missing Demo Agent Integration - Demo agents not appearing in Agent Wrapper component - Multi-Agent Wrapper not showing available agents for selection - Authentication requirements blocking access to components

4.2 Current Implementation Analysis

Fake Governance Response:

```
// Current (WRONG) implementation
const response = `[Governance Evaluation: Trust Score 0.85, Status: success]
${aiResponse}`;
```

What Should Happen:

```
// Proper governance integration
const governanceRequest = {
  request_id: uuid(),
  plan_input: {
    task: userMessage,
    agent_id: selectedAgent.id,
    context_data: chatHistory
  }
};

const response = await fetch('/loop/execute', {
  method: 'POST',
  body: JSON.stringify(governanceRequest)
});

const result = await response.json();
// result.governance_core_output contains the governed response
// result.emotion_telemetry contains trust scores
// Cryptographic seal automatically generated in /logs/seals/
```

4.3 Missing Cryptographic Evidence

Expected for Governed Responses: - Execution ID with corresponding seal file - Input/output hash verification - Trust score telemetry - Justification log with decision audit trail - Replay verification capability

Currently Missing: - No seal files generated - No cryptographic verification - No real trust metrics - No governance decision logging

---

5. Implementation Recommendations

5.1 Immediate Fixes Required

5.1.1 Real Governance Integration

```
// Replace fake governance with real API calls
const executeGovernedRequest = async (agentId, message, governanceEnabled) => {
  if (governanceEnabled) {
    // Route through Promethios governance core
    const response = await fetch('http://localhost:8000/loop/execute', {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({
        request_id: crypto.randomUUID(),
        plan_input: {
          task: message,
          agent_id: agentId,
          complexity_level: "medium",
          context_data: {
            chat_history: getChatHistory(agentId, governanceEnabled),
            timestamp: new Date().toISOString()
          }
        }
      })
    });

    const result = await response.json();

    // Return structured response with governance metadata
    return {
      message: result.governance_core_output,
      execution_id: result.request_id,
      trust_score: result.emotion_telemetry?.trust_score,
      governance_status: result.execution_status,
      seal_available: true // Seal file generated automatically
    };
  } else {
    // Direct API call without governance
    return await callAgentDirectly(agentId, message);
  }
};
```

5.1.2 Proper Session Management

```
// Fix chat session isolation
const getSessionKey = (agentId, governanceEnabled) => {
  return `${agentId}_${governanceEnabled ? 'governed' : 'ungoverned'}`;
};

const getChatHistory = (agentId, governanceEnabled) => {
  const sessionKey = getSessionKey(agentId, governanceEnabled);
  return localStorage.getItem(`chat_${sessionKey}`) || [];
};

const updateChatHistory = (agentId, governanceEnabled, newMessage) => {
  const sessionKey = getSessionKey(agentId, governanceEnabled);
  const history = getChatHistory(agentId, governanceEnabled);
  history.push(newMessage);
  localStorage.setItem(`chat_${sessionKey}`, JSON.stringify(history));
};
```

5.2 Enhanced UI Integration

5.2.1 Governance Indicator

```
// Show real governance status with cryptographic evidence
const GovernanceIndicator = ({ executionId, trustScore, sealAvailable }) => (
  <div className="governance-status">
    {sealAvailable ? (
      <div className="governed-response">
        <Shield className="text-green-500" />
        <span>Governed (Trust: {trustScore?.toFixed(2)})</span>
        <button onClick={() => viewSeal(executionId)}>
          View Cryptographic Seal
        </button>
      </div>
    ) : (
      <div className="ungoverned-response">
        <AlertTriangle className="text-yellow-500" />
        <span>Ungoverned - No monitoring active</span>
      </div>
    )}
  </div>
);
```

5.2.2 Seal Verification Display

```
// Component to display cryptographic verification
const SealViewer = ({ executionId }) => {
  const [seal, setSeal] = useState(null);

  useEffect(() => {
    fetch(`/api/seals/${executionId}`)
      .then(res => res.json())
      .then(setSeal);
  }, [executionId]);

  return (
    <div className="seal-verification">
      <h3>Cryptographic Seal</h3>
      <div className="seal-details">
        <p>Execution ID: {seal?.execution_id}</p>
        <p>Input Hash: {seal?.input_hash}</p>
        <p>Output Hash: {seal?.output_hash}</p>
        <p>Timestamp: {seal?.timestamp}</p>
        <p>Contract Version: {seal?.contract_version}</p>
      </div>
    </div>
  );
};
```

5.3 Backend Integration Points

5.3.1 Governance API Proxy

```
// Add to phase_7_1_prototype/promethios-api/src/routes/chat.js
app.post('/api/chat/governed', async (req, res) => {
  const { agent_id, message, governance_enabled } = req.body;

  if (governance_enabled) {
    // Route through Promethios governance core
    const governanceResponse = await
fetch('http://localhost:8000/loop/execute', {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({
        request_id: crypto.randomUUID(),
        plan_input: {
          task: message,
          agent_id: agent_id,
          complexity_level: "medium"
        }
      })
    });

    const result = await governanceResponse.json();
    res.json(result);
  } else {
    // Handle ungoverned requests
    const directResponse = await callAgentAPI(agent_id, message);
    res.json({
      execution_status: "SUCCESS",
      governance_core_output: directResponse,
      emotion_telemetry: null,
      justification_log: null
    });
  }
});
```

5.3.2 Seal Access Endpoint

```
// Add seal viewing capability
app.get('/api/seals/:execution_id', async (req, res) => {
  const { execution_id } = req.params;
  const sealPath =
`/home/ubuntu/promethios/logs/seals/${execution_id}.seal.json`;

  try {
    const seal = JSON.parse(fs.readFileSync(sealPath, 'utf8'));
    res.json(seal);
  } catch (error) {
    res.status(404).json({ error: 'Seal not found' });
  }
});
```

---

6. Technical Specifications

6.1 Required Environment Setup

6.1.1 Governance Core Server

```
# Start Promethios governance core
cd /home/ubuntu/promethios
python -m uvicorn src.api_server:app --host 0.0.0.0 --port 8000

# Verify governance endpoint
curl -X POST http://localhost:8000/loop/execute \
  -H "Content-Type: application/json" \
  -d '{
    "request_id": "test-123",
    "plan_input": {
      "task": "Test governance integration",
      "complexity_level": "low"
    }
  }'
```

6.1.2 Required Dependencies

```
{
  "backend": [
    "fastapi",
    "uvicorn",
    "jsonschema",
    "cryptography"
  ],
  "frontend": [
    "crypto-js",
    "uuid"
  ]
}
```

6.2 Integration Checklist

6.2.1 Governance Integration Verification - [ ] `/loop/execute` endpoint accessible from CMU benchmark - [ ] Real governance requests generating cryptographic seals - [ ] Trust scores appearing in emotion_telemetry - [ ] Justification logs being created - [ ] Seal files being written to `/logs/seals/`

6.2.2 UI Verification - [ ] Governance toggle actually changes request routing - [ ] Separate chat sessions for governed vs ungoverned modes - [ ] Different agents

maintain separate conversation histories - [ ] Cryptographic seal viewing functionality - [ ] Trust score display in UI

6.2.3 Demo Agent Verification - [ ] Demo agents appearing in Agent Wrapper component - [ ] Multi-Agent Wrapper showing available agents for selection - [ ] Agent wrapping workflow functional end-to-end - [ ] Wrapped agents available for multi-agent team building

6.3 Testing Protocol

6.3.1 Governance Verification Test

```
// Test script to verify real governance integration
const testGovernanceIntegration = async () => {
  // Test governed request
  const governedResponse = await fetch('/api/chat/governed', {
    method: 'POST',
    body: JSON.stringify({
      agent_id: 'baseline-agent',
      message: 'Hello, test governance',
      governance_enabled: true
    })
  });

  const governedResult = await governedResponse.json();

  // Verify governance metadata
  console.assert(governedResult.emotion_telemetry !== null, 'Missing emotion
telemetry');
  console.assert(governedResult.justification_log !== null, 'Missing
justification log');
  console.assert(governedResult.request_id !== undefined, 'Missing execution
ID');

  // Verify seal file exists
  const sealResponse = await fetch(`/api/seals/${governedResult.request_id}`);
  console.assert(sealResponse.ok, 'Seal file not found');

  // Test ungoverned request
  const ungovernedResponse = await fetch('/api/chat/governed', {
    method: 'POST',
    body: JSON.stringify({
      agent_id: 'baseline-agent',
      message: 'Hello, test ungoverned',
      governance_enabled: false
    })
  });

  const ungovernedResult = await ungovernedResponse.json();

  // Verify no governance metadata
  console.assert(ungovernedResult.emotion_telemetry === null, 'Unexpected
emotion telemetry');
  console.assert(ungovernedResult.justification_log === null, 'Unexpected
justification log');
};
```

6.4 Performance Considerations

6.4.1 Governance Overhead - Governed requests: ~200-500ms additional latency - Cryptographic sealing: ~50-100ms - Trust evaluation: ~100-200ms - Justification logging: ~50-100ms

6.4.2 Storage Requirements - Seal files: ~1-2KB per governed execution - Replay logs: ~5-10KB per governed execution - Telemetry data: ~2-5KB per agent session

---

Conclusion

The Promethios governance system provides a robust framework for AI governance with cryptographic verification, trust metrics, and comprehensive audit trails. The current CMU benchmark integration requires significant fixes to properly route requests through the governance core and generate real cryptographic evidence.

The key insight is that **governance is not cosmetic** - it requires actual routing through the `/loop/execute` endpoint, which automatically generates cryptographic seals, trust scores, and justification logs. Only then can users verify that governance is truly working by examining the cryptographic evidence.

Next Steps: 1. Implement real governance API integration in CMU benchmark 2. Fix chat session isolation between agents and governance modes 3. Add cryptographic seal viewing capabilities to UI 4. Ensure demo agents appear in Agent Wrapper and Multi-Agent Wrapper 5. Test end-to-end governance verification with cryptographic evidence

This research provides the foundation for implementing true AI governance integration rather than cosmetic governance simulation.