Internet Engineering Task Force
Internet-Draft
Intended status: Standards Track
Expires: 19 September 2026

J. van de Meent
R. AI
Humotica

18 March 2026

UPIP: Universal Process Integrity Protocol with Fork Tokens for Multi-Actor Continuation

draft-vandemeent-upip-process-integrity-00

Abstract

This document specifies UPIP (Universal Process Integrity Protocol), a five-layer protocol for capturing, verifying, and reproducing computational processes across machines, actors, and trust domains. UPIP defines a cryptographic hash chain over five layers: STATE, DEPS, PROCESS, RESULT, and VERIFY, enabling any party to prove that a process was executed faithfully and can be reproduced.

This document also specifies Fork Tokens, a continuation protocol enabling multi-actor process handoff with cryptographic chain of custody. Fork tokens freeze the complete UPIP stack state and transfer it to another actor (human, AI, or machine) for continuation, maintaining provenance integrity across the handoff boundary.

Together, UPIP and Fork Tokens address process integrity in multi-agent AI systems, distributed computing, scientific reproducibility, autonomous vehicle coordination, and regulatory compliance scenarios.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 19 September 2026.

Table of Contents

1. Introduction

Distributed computing increasingly involves heterogeneous actors: human operators, AI agents, automated pipelines, edge devices, and cloud services. When a process moves between actors -- from one machine to another, from an AI to a human for review, from a drone to a command station -- the integrity of the process state must be verifiable at every handoff point.

Existing solutions address parts of this problem:

* Version control (git) tracks code state but not execution state

* Container images (OCI) capture environment but not intent

* CI/CD pipelines (GitHub Actions, Jenkins) orchestrate execution but provide no cross-machine reproducibility proof

* Package managers (pip, npm) record dependencies but not the context of their use

None of these provide a unified, self-verifying bundle that captures the complete execution context (state, dependencies, process, result) with cryptographic chain of custody across actor boundaries.

UPIP fills this gap with two complementary protocols:

1. The UPIP Stack: a five-layer bundle capturing everything needed to reproduce a process, with a single stack hash that invalidates if any layer is modified.

2. Fork Tokens: a continuation protocol that freezes the stack state and transfers it to another actor with cryptographic proof of what was handed off, who handed it off, why, and what capabilities are required to continue.

Key design principles:

* EVIDENCE OVER ENFORCEMENT: UPIP proves what happened; it does not prevent bad actors from acting. In adversarial environments, enforcement can be bypassed but evidence cannot be un-recorded.

* HASH CHAIN INTEGRITY: Every layer is independently hashed. The stack hash chains them. Fork hashes chain into the fork chain. Tampering with any component invalidates the chain.

* ACTOR AGNOSTICISM: Actors may be human operators, AI agents, automated scripts, IoT devices, or any computational entity. The protocol makes no assumption about actor type.

* TRANSPORT AGNOSTICISM: UPIP bundles are JSON documents that can be transferred via any mechanism: file copy, HTTP API, message queue, or physical media.

2. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174].

Actor An entity that creates, modifies, or continues a UPIP process. Actors may be human operators, AI agents (IDDs as defined in [JIS]), automated scripts, IoT devices, or any computational entity.

Airlock An isolated execution environment (sandbox) where processes run before their results are applied to production state. The airlock captures all side effects without committing them.

Continuation Point A reference to the specific position in the UPIP stack where the fork occurs, expressed as "L{layer}:{position}". Example: "L4:post_result" indicates the fork occurs after L4 RESULT has been captured.

Fork Token A JSON document that freezes the UPIP stack state at a specific point and authorizes another actor to continue the process.

Fork Chain An ordered list of fork token references maintained in the UPIP stack, providing a complete history of all handoffs.

Fork-Squared (Fork^2) Parallel forking, where a single process is split into N independent sub-tasks distributed to N actors, each receiving a fork token of type "fragment".

IDD (Individual Device Derivative) An AI agent with unique identity. Defined in the companion TIBET specification [TIBET].

Shadow-Run Executing a process in the airlock to capture its effects without applying them. Used for fork validation.

Stack Hash The SHA-256 hash computed over the concatenation of L1 through L4 layer hashes, prefixed with "upip:". This single hash represents the complete integrity of the UPIP bundle.

UPIP Stack (Bundle) A JSON document containing all five UPIP layers plus metadata. Files use the ".upip.json" extension.

3. Protocol Overview

UPIP operates in two modes:

Single-Actor Mode (Capture-Run-Verify):

1. CAPTURE: Record L1 (state) and L2 (deps)
2. RUN: Execute the process (L3) in an airlock
3. RESULT: Capture L4 (output, diff, hash)
4. HASH: Compute stack_hash = SHA-256(L1 || L2 || L3 || L4)
5. VERIFY: On another machine, reproduce and compare (L5)

Multi-Actor Mode (Fork-Resume):

1. Actor A completes steps 1-4 (single-actor mode)
2. Actor A creates a Fork Token from the UPIP stack
3. Actor A delivers the fork token to Actor B

4. Actor B validates the fork token hash

5. Actor B checks capability requirements

6. Actor B executes continuation in an airlock (shadow-run)

7. Actor B creates a new UPIP stack linked to Actor A's via the fork chain

8. Actor B sends ACK with resume_hash to Actor A

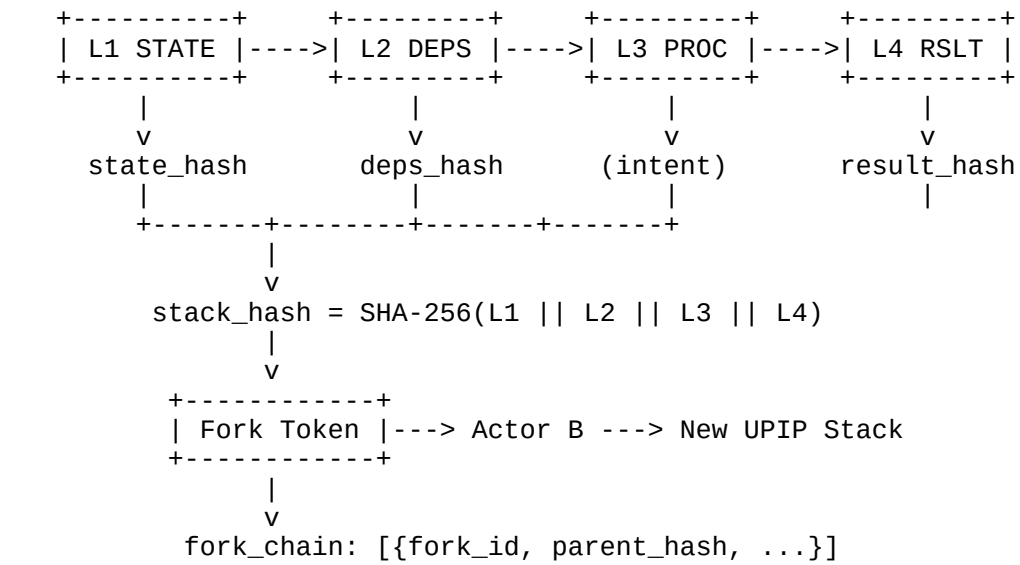

Figure 1: Process Flow Diagram

4. UPIP Stack Structure

A UPIP stack MUST be a [RFC8259] JSON object with the following top-level fields:

```
{
  "protocol": "UPIP",
  "version": "1.0",
  "title": "<human-readable description>",
  "created_by": "<actor identity>",
  "created_at": "<ISO-8601 timestamp>",
  "stack_hash": "upip:sha256:<hex>",
  "state": { "<L1 object>" : "..." },
  "deps": { "<L2 object>" : "..." },
  "process": { "<L3 object>" : "..." },
  "result": { "<L4 object>" : "..." },
  "verify": [ "<L5 array>" ],
  "fork_chain": [ "<fork token references>" ],
  "source_files": { "<optional embedded files>" : "..." }
}
```

4.1. L1 STATE - Input State Capture

L1 captures the complete input state before execution. The state_type field determines the capture method:

```
{
  "state_type": "git | files | image | empty",
  "state_hash": "<type>:<hash>",
  "captured_at": "<ISO-8601 timestamp>"
}
```

State Types:

git Hash is the git commit SHA. MUST include git_remote and git_branch. state_hash prefix: "git:"

files Hash is SHA-256 of the sorted file manifest. state_hash prefix: "files:"

image Hash is the container image digest. state_hash prefix: "image:"

empty No input state. state_hash: "empty:0"

For "git" type, additional fields:

* git_remote: Repository URL

* git_branch: Branch name

* git_dirty: Boolean, true if uncommitted changes exist

For "files" type, additional fields:

* file_count: Number of files captured

* total_size: Total size in bytes

* manifest: Optional array of {path, hash, size} objects

4.2. L2 DEPS - Dependency Snapshot

L2 captures the exact dependency set at execution time.

```
{
  "python_version": "<major.minor.patch>",
  "packages": { "<name>": "<version>" },
  "system_packages": [ "<name>=<version>" ],
  "deps_hash": "deps:sha256:<hex>",
  "captured_at": "<ISO-8601 timestamp>"
}
```

The deps_hash MUST be computed as SHA-256 of the sorted, deterministic serialization of all package name:version pairs.

While this specification uses Python as the reference implementation, L2 is language-agnostic. Other implementations MAY substitute appropriate dependency metadata for their runtime environment (e.g., Cargo.lock for Rust, go.sum for Go, package-lock.json for Node.js).

4.3. L3 PROCESS - Execution Definition

L3 defines what was executed and why.

```
{
  "command": [ "<arg0>", "<arg1>" ],
  "intent": "<human-readable purpose>",
  "actor": "<actor identity>",
  "env_vars": { "<key>": "<value>" },
  "working_dir": "<path>"
}
```

The command field MUST be an array of strings, not a shell command string. This prevents injection attacks and ensures deterministic execution.

The intent field MUST be a human-readable string describing WHY this process is being run. This serves as the ERACHTER (intent) component for TIBET integration.

The actor field MUST identify the entity that initiated the process. This may be a human username, AI agent identifier (IDD), or system service name.

4.4. L4 RESULT - Output Capture

L4 captures the execution result.

```
{
  "success": true,
  "exit_code": 0,
  "stdout": "<captured stdout>",
  "stderr": "<captured stderr>",
  "result_hash": "sha256:<hex>",
  "files_changed": 3,
  "diff": "<unified diff of file changes>",
  "captured_at": "<ISO-8601 timestamp>"
}
```

The result_hash MUST be computed as SHA-256 of the concatenation of: exit_code (as string) + stdout + stderr.

If execution occurs in an airlock, the diff field SHOULD contain the unified diff of all file changes detected.

4.5. L5 VERIFY - Cross-Machine Proof

L5 records verification attempts when the UPIP stack is reproduced on another machine.

```
{
  "machine": "<hostname or identifier>",
  "verified_at": "<ISO-8601 timestamp>",
  "match": true,
  "environment": { "os": "linux", "arch": "x86_64" },
  "original_hash": "upip:sha256:<hex>",
  "reproduced_hash": "upip:sha256:<hex>"
}
```

The match field MUST be true only if reproduced_hash equals original_hash.

L5 is an array, allowing multiple verification records from different machines. Each verification is independent.

4.6. Stack Hash Computation

The stack hash MUST be computed as follows:

1. Serialize each layer hash as a UTF-8 string: L1: state.state_hash, L2: deps.deps_hash, L3: SHA-256(json_serialize(process)), L4: result.result_hash

2. Concatenate with pipe separator: L1 + "|" + L2 + "|" + L3 + "|" + L4

3. Compute SHA-256 of the concatenated string

4. Prefix with "upip:sha256:"

Result: "upip:sha256:4f2e8a..."

This ensures that modifying ANY layer invalidates the stack hash.

5. Fork Tokens

5.1. Fork Token Structure

A fork token MUST be a [RFC8259] JSON object with the following fields. The fork_id SHOULD be a UUID as defined in [RFC4122]:

```
{
  "fork_id": "<unique identifier>",
  "parent_hash": "sha256:<hex>",
  "parent_stack_hash": "upip:sha256:<hex>",
  "continuation_point": "L<n>:<position>",
  "intent_snapshot": "<human-readable purpose>",
  "active_memory_hash": "sha256:<hex>",
  "memory_ref": "<path or URL to memory blob>",
  "fork_type": "script | ai_to_ai | human_to_ai | fragment",
  "actor_from": "<source actor>",
  "actor_to": "<target actor or empty>",
  "actor_handoff": "<from> -> <to>",
  "capability_required": { },
  "forked_at": "<ISO-8601 timestamp>",
  "expires_at": "<ISO-8601 timestamp or empty>",
  "fork_hash": "fork:sha256:<hex>",
  "partial_layers": { },
  "metadata": { }
}
```

The actor_to field MAY be empty, indicating the fork is available to any capable actor. In this case, actor_handoff MUST use "*" as the target: "ActorA -> *".

5.2. Fork Types

script The UPIP bundle IS the complete state. No external memory blob is needed. Used for CLI pipelines, CI/CD, and batch processing. The active_memory_hash is computed from the L1+L2+L3+L4 layer hashes.

ai_to_ai The AI actor's context window is serialized as a binary blob (.blob file). The active_memory_hash is the SHA-256 of this blob. The memory_ref field SHOULD point to the blob's location.

human_to_ai A human creates an intent document (natural language instructions) and delegates to an AI actor. The active_memory_hash is the SHA-256 of the intent document.

fragment A parallel fork (Fork-Squared). The parent process is split into N sub-tasks, each receiving a fork token of type "fragment" with the specific portion assigned.

5.3. Fork Hash Computation

The fork hash MUST be computed as follows:

1. Concatenate with pipe separator: fork_id + "|" + parent_hash + "|" + parent_stack_hash + "|" + continuation_point + "|" + intent_snapshot + "|" + active_memory_hash + "|" + actor_handoff + "|" + fork_type

2. Compute SHA-256 of the concatenated string

3. Prefix with "fork:sha256:"

Result: "fork:sha256:7d3f..."

This ensures that modifying ANY field invalidates the fork.

5.4. Active Memory Hash

The active_memory_hash field captures the cognitive or computational state at the moment of forking.

For fork_type "script": SHA-256(state_hash + "|" + deps_hash + "|" + process_intent + "|" + result_hash)

For fork_type "ai_to_ai": SHA-256(contents of .blob file)

For fork_type "human_to_ai": SHA-256(contents of intent document)

For fork_type "fragment": SHA-256(fragment specification)

This field answers the question: "What was the actor thinking/ processing at the moment of handoff?"

5.5. Capability Requirements

The capability_required field specifies what the resuming actor needs:

```
{
  "capability_required": {
    "deps": ["package>=version"],
    "gpu": true,
    "min_memory_gb": 16,
    "platform": "linux/amd64",
    "custom": { }
  }
}
```

On resume, the receiving actor SHOULD verify these requirements and record the result in the verification record. Missing capabilities MUST NOT prevent execution but MUST be recorded as evidence.

5.6. Fork Chain

The fork_chain field in the UPIP stack is an ordered array of fork token references:

```
{
  "fork_chain": [
    {
      "fork_id": "fork-abc123",
      "fork_hash": "fork:sha256:...",
      "actor_handoff": "A -> B",
      "forked_at": "2026-03-18T14:00:00Z"
    }
  ]
}
```

When a process is resumed, the new UPIP stack MUST include the fork token in its fork_chain. This creates a complete audit trail of all handoffs.

6. Operations

6.1. Capture and Run

Input: command, source_dir, intent, actor

Output: UPIP stack with L1-L4 populated

1. Capture L1 STATE from source_dir
2. Capture L2 DEPS from current environment
3. Define L3 PROCESS from command and intent
4. Execute command in airlock
5. Capture L4 RESULT
6. Compute stack_hash
7. Return UPIP stack

6.2. Reproduce

Input: UPIP stack (.upip.json), target machine

Output: L5 VERIFY record

1. Load UPIP stack from file
2. Restore L1 STATE (checkout git, extract files)
3. Verify L2 DEPS match (warn on mismatches)
4. Execute L3 PROCESS in airlock
5. Capture L4 RESULT on target machine
6. Compare result_hash with original
7. Create L5 VERIFY record
8. Return verification result

6.3. Fork

Input: UPIP stack, actor_from, actor_to, intent

Output: Fork Token

1. Load UPIP stack
2. Compute active_memory_hash from L1-L4
3. Snapshot partial_layers (hash + key fields per layer)
4. Generate fork_id
5. Compute fork_hash
6. Create Fork Token
7. Append to fork_chain in parent stack
8. Return Fork Token

6.4. Resume

Input: Fork Token (.fork.json), command, actor

Output: New UPIP stack, verification record

1. Load Fork Token
2. Recompute fork_hash and compare (tamper check)
3. Check capability_required against local environment
4. Execute command in airlock (shadow-run)
5. Capture new UPIP stack (L1-L4)
6. Copy fork_chain from parent, append this fork
7. Create L5 VERIFY with fork validation results
8. Return new stack + verification

6.5. Fragment (Fork-Squared)

Input: UPIP stack, N fragments, actor list

Output: N Fork Tokens of type "fragment"

1. Load UPIP stack

2. Define fragment specification (how to split)

3. For each fragment i in 1..N: Create Fork Token with fork_type="fragment", set fragment-specific metadata (index, total, range), and deliver to actor[i].

4. Wait for N ACKs

5. Verify all fragment hashes

6. Reconstruct combined result

Fragment tokens MUST include metadata fields:

* fragment_index: Position in sequence (0-based)

* fragment_total: Total number of fragments

* fragment_spec: Description of this fragment's portion

7. Transport: I-Poll Delivery

While UPIP is transport-agnostic, this section defines the I-Poll binding for real-time fork delivery between AI agents.

7.1. Fork Delivery Message Format

Fork tokens are delivered via I-Poll TASK messages:

```
{
  "from_agent": "<source agent>",
  "to_agent": "<target agent>",
  "content": "<human-readable fork summary>",
  "poll_type": "TASK",
  "metadata": {
    "upip_fork": true,
    "fork_id": "<fork_id>",
    "fork_hash": "fork:sha256:<hex>",
    "fork_type": "<type>",
    "continuation_point": "<point>",
    "actor_handoff": "<from> -> <to>",
    "fork_data": { "<complete fork token JSON>" : "..." }
  }
}
```

The "upip_fork" metadata flag MUST be true to identify this message as a fork delivery.

The "fork_data" field MUST contain the complete fork token as defined in Section 5.1. This allows the receiving agent to reconstruct the fork token without needing the .fork.json file.

7.2. Acknowledgment

After processing a fork token, the receiving actor SHOULD send an ACK message:

```
{
  "from_agent": "<resuming agent>",
  "to_agent": "<original agent>",
  "content": "FORK RESUMED_OK -- <fork_id>",
  "poll_type": "ACK",
  "metadata": {
    "upip_fork": true,
    "fork_id": "<fork_id>",
    "fork_status": "RESUMED_OK",
    "resume_hash": "upip:sha256:<hex>",
    "resumed_by": "<agent identity>"
  }
}
```

The resume_hash is the stack_hash of the new UPIP stack created during resume.

The fork_status field MUST be one of "RESUMED_OK" or "RESUMED_FAIL".

7.3. Listener Pattern

Agents MAY implement a poll-based listener that:

1. Periodically polls I-Poll inbox for TASK messages
2. Filters for messages with "upip_fork": true in metadata
3. Extracts the fork token from "fork_data"
4. Validates the fork hash
5. Executes the resume operation
6. Sends ACK back to the original agent

Poll interval SHOULD be configurable. Default: 5 seconds. Implementations SHOULD support exponential backoff when the inbox is empty.

8. Validation Rules

8.1. Stack Validation

A UPIP stack is valid if and only if:

1. All required fields are present
2. state_hash matches SHA-256 of the state data
3. deps_hash matches SHA-256 of the dependency data
4. result_hash matches SHA-256 of exit_code + stdout + stderr
5. stack_hash matches SHA-256(L1 || L2 || L3 || L4)

Validation MUST be performed when loading a .upip.json file and SHOULD be performed before reproduction.

8.2. Fork Validation on Resume

When resuming a fork token, the following checks MUST be performed:

1. FORK HASH: Recompute fork_hash from token fields and compare with stored fork_hash. Records tamper evidence.
2. STORED HASH: Compare fork_hash with the hash stored in the .fork.json file header.
3. CAPABILITIES: Verify each entry in capability_required against the local environment. Record missing capabilities.
4. EXPIRATION: Check expires_at if present. Expired forks SHOULD generate a warning but MUST NOT be blocked.

All four checks MUST be recorded in the L5 VERIFY record. Failed checks MUST NOT prevent execution (evidence over enforcement principle).

8.3. Tamper Evidence

If fork_hash validation fails, the verification record MUST include:

```
{
  "fork_hash_match": false,
  "expected_hash": "fork:sha256:<original>",
  "computed_hash": "fork:sha256:<recomputed>",
  "tamper_evidence": true
}
```

This creates an immutable record that tampering occurred, without preventing the process from continuing. The decision of whether to act on tamper evidence is left to the consuming application.

9. Security Considerations

9.1. Hash Chain Integrity

UPIP uses SHA-256 for all hash computations. Implementations MUST use SHA-256 as defined in [FIPS180-4]. Future versions MAY support SHA-3 or other hash functions via an algorithm identifier prefix.

The hash chain structure ensures that modifying any component at any layer propagates to the stack hash, providing tamper evidence for the entire bundle.

9.2. Evidence vs Enforcement

UPIP is deliberately designed as an evidence protocol, not an enforcement protocol. Fork validation failures do not block execution; they are recorded as evidence. This design choice reflects the reality that:

* In adversarial environments, enforcement can be circumvented
* Evidence creates accountability that enforcement cannot
* Downstream consumers can make their own trust decisions based on the evidence chain

Applications that require enforcement SHOULD implement additional policy layers on top of UPIP evidence.

9.3. Memory Hash for AI Actors

When fork_type is "ai_to_ai", the active_memory_hash represents the SHA-256 of the serialized AI context window. This raises unique considerations:

* Context serialization format is model-dependent

* The blob may contain sensitive information

* Exact reproduction of AI state is generally not possible

Implementations SHOULD encrypt memory blobs at rest. Implementations MUST NOT require exact memory reproduction for fork validation. The memory hash serves as evidence of state at fork time, not as a reproducibility guarantee.

9.4. Capability Verification

Capability requirements in fork tokens are self-reported by the forking actor. The receiving actor SHOULD independently verify capabilities rather than trusting the requirement specification alone.

Package version verification SHOULD use installed package metadata. GPU availability SHOULD be verified via hardware detection, not configuration claims.

9.5. Replay Attacks

Fork tokens include fork_id and forked_at fields to mitigate replay attacks. Implementations SHOULD track consumed fork_ids and reject duplicate fork_ids within a configurable time window.

The expires_at field provides time-based expiration. Agents SHOULD set expires_at for forks that are time-sensitive.

9.6. Regulatory Alignment

UPIP's evidence-based design aligns with requirements from the [EU-AI-ACT], [NIST-AI-RMF], and [ISO42001]. The complete process capture at each layer provides the audit trail required by these frameworks for AI system transparency and accountability.

10. IANA Considerations

This document requests registration of:

10.1. Media Type Registrations

Media Type: application/upip+json

File Extension: .upip.json

Media Type: application/upip-fork+json

File Extension: .fork.json

10.2. HTTP Header Field Registrations

* X-UPIP-Stack-Hash
* X-UPIP-Fork-ID
* X-UPIP-Fork-Hash

11. References

11.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

[RFC8259] Bray, T., "The JavaScript Object Notation (JSON) Data Interchange Format", RFC 8259, December 2017, <https://www.rfc-editor.org/info/rfc8259>.

[RFC4122] Leach, P., Mealling, M., and R. Salz, "A Universally Unique IDentifier (UUID) URN Namespace", RFC 4122, July 2005, <https://www.rfc-editor.org/info/rfc4122>.

[FIPS180-4] National Institute of Standards and Technology (NIST), "Secure Hash Standard (SHS)", FIPS PUB 180-4, August 2015.

11.2. Informative References

[TIBET] van de Meent, J. and R. AI, "TIBET: Transaction/ Interaction-Based Evidence Trail", Work in Progress, Internet-Draft, draft-vandemeent-tibet-provenance-00, January 2026, <https://datatracker.ietf.org/doc/html/ draft-vandemeent-tibet-provenance-00>.

[JIS] van de Meent, J. and R. AI, "JIS: JTel Identity Standard", Work in Progress, Internet-Draft, draft-vandemeent-jis-identity-00, January 2026, <https://datatracker.ietf.org/doc/html/draft-vandemeent-jis-identity-00>.

[EU-AI-ACT]
European Commission, "Regulation laying down harmonised rules on artificial intelligence", 2024.

[NIST-AI-RMF]
National Institute of Standards and Technology (NIST), "Artificial Intelligence Risk Management Framework (AI RMF 1.0)", January 2023.

[ISO42001] International Organization for Standardization (ISO), "Artificial intelligence - Management system", ISO/IEC 42001:2023, 2023.

Appendix A. UPIP Stack JSON Schema

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "type": "object",
  "required": ["protocol", "version", "stack_hash",
               "state", "deps", "process", "result"],
  "properties": {
    "protocol": {"const": "UPIP"},
    "version": {"type": "string"},
    "title": {"type": "string"},
    "created_by": {"type": "string"},
    "created_at": {"type": "string", "format": "date-time"},
    "stack_hash": {
      "type": "string",
      "pattern": "^upip:sha256:[a-f0-9]{64}$"
    },
    "state": {
      "type": "object",
      "required": ["state_type", "state_hash"],
      "properties": {
        "state_type": {
          "enum": ["git", "files", "image", "empty"]
        },
        "state_hash": {"type": "string"}
      }
    },
    "deps": {
      "type": "object",
      "required": ["deps_hash"],
      "properties": {
        "python_version": {"type": "string"},
        "packages": {"type": "object"},
        "deps_hash": {"type": "string"}
      }
```

```
      },
      "process": {
        "type": "object",
        "required": ["command", "intent", "actor"],
        "properties": {
          "command": {"type": "array", "items": {"type": "string"}},
          "intent": {"type": "string"},
          "actor": {"type": "string"}
        }
      },
      "result": {
        "type": "object",
        "required": ["success", "exit_code", "result_hash"],
        "properties": {
          "success": {"type": "boolean"},
          "exit_code": {"type": "integer"},
          "result_hash": {"type": "string"}
        }
      },
      "fork_chain": {
        "type": "array",
        "items": {"type": "object"}
      }
    }
  }
```

Appendix B. Fork Token JSON Schema

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "type": "object",
  "required": ["fork_id", "fork_type", "fork_hash",
               "active_memory_hash", "forked_at"],
  "properties": {
    "fork_id": {"type": "string", "pattern": "^fork-"},
    "parent_hash": {"type": "string"},
    "parent_stack_hash": {
      "type": "string",
      "pattern": "^upip:sha256:"
    },
    "continuation_point": {"type": "string"},
    "intent_snapshot": {"type": "string"},
    "active_memory_hash": {
      "type": "string",
      "pattern": "^sha256:"
    },
    "memory_ref": {"type": "string"},
    "fork_type": {
      "enum": ["script", "ai_to_ai", "human_to_ai", "fragment"]
    },
    "actor_from": {"type": "string"},
    "actor_to": {"type": "string"},
    "actor_handoff": {"type": "string"},
    "capability_required": {"type": "object"},
    "forked_at": {"type": "string", "format": "date-time"},
    "expires_at": {"type": "string"},
    "fork_hash": {
      "type": "string",
      "pattern": "^fork:sha256:[a-f0-9]{64}$"
    },
    "partial_layers": {"type": "object"},
    "metadata": {"type": "object"}
  }
}
```

Appendix C. Use Case Examples

C.1. Multi-Agent AI Task Delegation

An AI orchestrator (Agent A) analyzes a dataset, creates a UPIP bundle, forks it to a specialist AI (Agent B) for deep analysis, and receives the result with cryptographic proof.

```
Agent A:
  capture_and_run(["python", "scan.py"], intent="Initial scan")
  fork_upip(actor_from="A", actor_to="B", intent="Deep analysis")
  deliver_fork(fork, to_agent="B")

Agent B:
  pull_forks()
  resume_upip(fork, command=["python", "deep_analyze.py"])
  ack_fork(fork, resume_hash=stack.hash, success=True)
```

Result: Both agents have UPIP stacks linked by fork_chain. Any auditor can verify the complete chain.

C.2. Drone Swarm Coordination

A command station dispatches N reconnaissance tasks to N drones. Each drone receives a fragment fork token, executes its assigned sector scan, and returns the result.

```
Command Station:
  base_stack = capture_and_run(["mission_plan.py"])
  for i in range(N):
    fork = fork_upip(base_stack,
                     actor_from="command",
                     actor_to=f"drone-{i}",
                     fork_type="fragment",
                     metadata={"sector": sectors[i]})
    deliver_fork(fork, to_agent=f"drone-{i}")

Each Drone:
  fork_msg = pull_forks()
  stack = resume_upip(fork, command=["scan_sector.py"])
  ack_fork(fork, resume_hash=stack.hash)

Command Station:
  # Verify all N results, reconstruct combined map
  for ack in collect_acks():
    verify(ack.resume_hash)
```

C.3. Scientific Experiment Reproduction

Lab A publishes an experiment as a UPIP bundle. Lab B reproduces it independently and gets cryptographic proof that results match (or don’t).

```
Lab A:
  stack = capture_and_run(
    ["python", "train_model.py"],
    source_dir="./experiment",
    intent="Train model v3 on dataset-2026Q1"
  )
  save_upip(stack, "experiment-2026Q1.upip.json")
  # Publish to journal / data repository

Lab B:
  stack = load_upip("experiment-2026Q1.upip.json")
  verify = reproduce_upip(stack)
  # verify.match == True: exact reproduction
  # verify.match == False: divergence (investigate)
```

Acknowledgements

The UPIP protocol was developed as part of HumoticaOS, an AI governance framework built on human-AI symbiosis. UPIP builds on concepts from the TIBET evidence trail protocol and extends them into the domain of process integrity and multi-actor continuation.

The Fork Token mechanism was inspired by the need for cryptographic chain of custody in multi-agent AI systems, where processes move between heterogeneous actors across trust boundaries.

Authors' Addresses

Jasper van de Meent
Humotica
Den Dolder
Netherlands
Email: jasper@humotica.com
URI: https://humotica.com

Root AI
Humotica
Email: root_ai@humotica.nl
URI: https://humotica.com