Individual Submission
Internet-Draft
Intended status: Standards Track
Expires: 2 September 2026

K. Rickert
PipeStream AI

1 March 2026

PipeStream: A Recursive Entity Streaming Protocol for Distributed Processing over QUIC
draft-krickert-pipestream-02

Abstract

This document specifies PipeStream, a recursive entity streaming protocol for hierarchical task decomposition and distributed processing over QUIC transport. PipeStream enables the decomposition ("dehydration") of complex inputs into constituent entities, their transmission across distributed processing nodes, and subsequent rehydration (reassembly) at destination endpoints. The protocol's primary motivating use case is distributed document processing, but its recursive scatter-gather design is applicable to any domain requiring hierarchical decomposition of work units.

The protocol employs a dual-stream architecture consisting of a data stream for entity payload transmission and a control stream for tracking entity completion status and maintaining consistency. PipeStream defines four hierarchical data layers for entity representation: BlobBag for raw binary data, SemanticLayer for annotated content with metadata, ParsedData for structured extracted information, and CustomEntity for application-specific extensions.

PipeStream is organized into three protocol layers: Layer 0 (Core) provides basic streaming with dehydrate/rehydrate semantics; Layer 1 (Recursive) adds hierarchical scoping and digest propagation; Layer 2 (Resilience) adds yield/resume, claim checks, and completion policies. All implementations support Layer 0; Layers 1 and 2 are optional and negotiated at connection time.

To ensure consistency across distributed processing pipelines, PipeStream implements checkpoint blocking, whereby processing nodes synchronize at defined points before proceeding. This mechanism guarantees that all constituent parts of a dehydrated entity are successfully processed before rehydration operations commence.

About This Document

This note is to be removed before publishing as an RFC.

Status information for this document may be found at https://datatracker.ietf.org/doc/draft-krickert-pipestream/.

Discussion of this document takes place on the Individual Group mailing list (mailto:kristian.rickert@pipestream.ai).

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 2 September 2026.

Copyright Notice

Table of Contents

1. Introduction

1.1. Problem Statement

Distributed processing pipelines face significant challenges when handling large, complex work units that require multiple stages of transformation, analysis, and enrichment. Traditional batch processing approaches require entire inputs to be loaded into memory, processed sequentially, and transmitted in their entirety between processing stages. This methodology introduces substantial latency, excessive memory consumption, and poor utilization of distributed computing resources.

Modern distributed processing workflows increasingly demand the ability to:

* Process inputs incrementally as data becomes available

* Distribute processing load across heterogeneous worker nodes

* Maintain consistency guarantees across parallel processing paths

* Handle inputs of arbitrary size without memory constraints

* Support recursive decomposition where constituent parts may themselves be decomposed

* Scale from single work units to collections of millions of inputs

Current approaches based on batch processing and store-and-forward architectures are inefficient for large inputs and fail to exploit the inherent parallelism available in distributed processing environments. Furthermore, existing streaming protocols do not provide the consistency semantics required for hierarchical processing where the integrity of the rehydrated output depends on the successful processing of all constituent parts.

1.2. Applicability

Although PipeStream was originally motivated by distributed document processing pipelines, its recursive scatter-gather design is domain-neutral. The protocol is applicable to any workload that can be modeled as hierarchical decomposition of a root entity into sub-entities, parallel processing of those sub-entities, and deterministic reassembly of results. Example domains include:

* Distributed document processing and content enrichment pipelines
* Hierarchical video transcoding and rendering (scene decomposition)
* Federated computation pipelines such as distributed machine learning
* Genomic sequencing and assembly workflows

Throughout this document, "document" is used as the canonical example of a root entity, but all protocol mechanisms apply equally to any decomposable work unit.

1.3. PipeStream Overview

PipeStream addresses these challenges by defining a streaming protocol that enables incremental processing with strong consistency guarantees. The protocol is built upon QUIC [RFC9000] transport, leveraging its native support for multiplexed streams, low-latency connection establishment, and reliable delivery semantics.

The fundamental innovation of PipeStream is its treatment of inputs as recursive compositions of entities. A root entity MAY be decomposed into multiple sub-entities, each of which MAY itself be further decomposed, creating a tree structure of processing tasks. This recursive decomposition enables fine-grained parallelism while the protocol’s control stream mechanism ensures that all branches of the decomposition tree are tracked and synchronized.

PipeStream employs a dual-stream design:

1. *Data Stream*: Carries entity payloads through the processing pipeline. Entities flow through this stream with minimal buffering, enabling low-latency incremental processing.

2. *Control Stream*: Carries control information tracking the status of entity decomposition and rehydration. The control stream ensures that all parts of a dehydrated entity are accounted for before rehydration proceeds.

1.4. Design Philosophy

PipeStream implements a recursive scatter-gather pattern [scatter-gather] over QUIC streams. An input entity is "dehydrated" (scattered) at the source into constituent sub-entities, these sub-entities are transmitted and processed in parallel across distributed pipeline stages, and finally the results are "rehydrated" (gathered) at the destination to reconstitute the complete processed output. The checkpoint blocking mechanism (Section 9.3) provides barrier synchronization semantics analogous to the barrier pattern in parallel computing.

This approach provides several advantages:

* *Incremental Processing*: Processing nodes MAY begin work on early entities before the complete input has been transmitted.

* *Parallelism*: Independent entities MAY be processed concurrently across multiple worker nodes.

* *Memory Efficiency*: No single node is required to hold the complete input in memory.

* *Fault Isolation*: Failures in processing individual entities can be detected, reported, and potentially retried without affecting other entities.

* *Consistency*: The checkpoint blocking mechanism ensures that rehydration operations proceed only when all constituent parts have been successfully processed.

1.5. Protocol Layering

PipeStream is organized into three protocol layers to accommodate varying deployment requirements:

Protocol Layer	Name	Description
Layer 0	Core	Basic streaming, dehydrate/ rehydrate, checkpoint
Layer 1	Recursive	Hierarchical scopes, digest propagation, barriers
Layer 2	Resilience	Yield/resume, claim checks, completion policies

Table 1

Implementations MUST support Layer 0. Support for Layers 1 and 2 is OPTIONAL and negotiated during connection establishment.

1.6. Scope

This document specifies the PipeStream protocol including message formats, state machines, error handling, and the interaction between data and control streams. The document defines the four standard data layers but does not mandate specific processing semantics, which are left to application-layer specifications.

2. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

2.1. Protocol Entities

Entity The fundamental unit of data flowing through a PipeStream pipeline. An Entity represents either a complete document or a constituent part of a decomposed document. Each Entity possesses a unique identifier within its processing scope and carries payload data in one of the four defined Layer formats. Entities are immutable once created; transformations produce new Entities rather than modifying existing ones.

Document A logical unit of content or work submitted to a

PipeStream pipeline for processing. A Document enters the pipeline as a single root Entity and MAY be decomposed into multiple Entities during processing. The Document is considered complete when its root Entity (or the rehydrated result of its decomposition) exits the pipeline.

Scope A hierarchical namespace for Entity IDs. Each scope maintains its own Entity ID space, cursor, and Assembly Manifest. Scopes enable collections to contain documents, documents to contain parts, and parts to contain jobs, each with independent ID management. (Protocol Layer 1)

2.2. Dehydration and Rehydration

Scatter-Gather The distributed processing pattern implemented by PipeStream. A single input is "scattered" (dehydrated) into multiple parts for parallel processing, and the results are "gathered" (rehydrated) back into a single output. PipeStream extends classical scatter-gather with recursive nesting: any scattered part may itself be scattered further.

Dehydrate (Scatter) The operation of decomposing a document or Entity into multiple constituent Entities for parallel or distributed processing. When an Entity is dehydrated, the originating node MUST create an Assembly Manifest entry recording the identifiers of all resulting sub-entities. The dehydration operation is recursive; a sub-entity produced by dehydration MAY itself be dehydrated, creating a tree of decomposition. Dehydration transitions data from a solid state (a single stored record) to a fluid state (multiple in-flight entities).

Rehydrate (Gather) The operation of reassembling multiple Entities back into a single composite Entity or Document. A rehydrate operation MUST NOT proceed until all constituent Entities listed in the corresponding Assembly Manifest entry have been received and processed (or handled according to the Completion Policy). Rehydration transitions data from a fluid state back to a solid state.

Solid State A document or Entity that exists as a complete, stored record -- either at rest in storage or as a single root Entity entering or exiting a pipeline. Contrast with "fluid state".

Fluid State A document that has been decomposed into multiple in-flight Entities being processed in parallel across distributed nodes. A document is in the fluid state between dehydration and rehydration. Contrast with "solid state".

2.3. Consistency Mechanisms

Checkpoint A synchronization point in the processing pipeline where all in-flight Entities MUST reach a consistent state before processing may continue. A checkpoint is considered "satisfied" when all Assembly Manifest entries created before the checkpoint have been resolved.

Barrier A synchronization point scoped to a specific subtree. Unlike checkpoints which are global, barriers block only entities dependent on a specific parent’s descendants. (Protocol Layer 1)

Control Stream The control stream that tracks Entity completion status throughout the processing pipeline. The Control Stream is transmitted on a dedicated QUIC stream parallel to the data streams.

Assembly Manifest A data structure within the Control Stream that tracks the relationship between a composite Entity and its constituent sub-entities produced by dehydration.

Cursor A pointer to the lowest unresolved Entity ID within a scope. Entity IDs behind the cursor are considered resolved and MAY be recycled. The cursor enables efficient ID space management without global coordination.

2.4. Resilience Mechanisms (Protocol Layer 2)

Yield A temporary pause in Entity processing, typically due to external dependencies (API calls, rate limiting, human approval). A yielded Entity carries a continuation token enabling resumption without reprocessing.

Claim Check A detached reference to a deferred Entity that can be queried or resumed independently, potentially in a different session. Claim checks enable asynchronous processing patterns and retry queues.

Completion Policy A configuration specifying how to handle partial failures during dehydration. Policies include STRICT (all must succeed), LENIENT (continue with partial results), BEST_EFFORT (complete with whatever succeeds), and QUORUM (require minimum success ratio).

2.5. Data Representation

Data Layer One of four defined representations for Entity payload data:

1. *BlobBag*: Raw binary data with minimal metadata

2. *SemanticLayer*: Annotated content with structural and semantic metadata

3. *ParsedData*: Structured information extracted from document content

4. *CustomEntity*: Application-specific extension Layer

2.6. Additional Terms

Pipeline A configured sequence of processing stages through which Entities flow.

Processor A node in the mesh that performs operations on entities (e.g., transformation, dehydration, or rehydration).

Sink A terminal stage in a pipeline where rehydrated documents are persisted or delivered to an external system.

Stage A single processing step within a Pipeline.

Scope Digest A cryptographic summary (Merkle root) of all Entity statuses within a completed scope, propagated to parent scopes for efficient verification. (Protocol Layer 1)

3. Protocol Layers

PipeStream defines three protocol layers that build upon each other. This layered approach allows simple deployments to use only the core protocol while complex deployments can leverage advanced features.

3.1. Layer 0: Core Protocol

Layer 0 provides the fundamental streaming capabilities:

* Unified Control Frame (UCF) header (1-octet type)

* Status frame (16-octet base bit-packed frame)

* Entity frame (header + payload)

* Status codes: PENDING, PROCESSING, COMPLETE, FAILED, CHECKPOINT

* Assembly Manifest for parent-child tracking

* Cursor-based Entity ID recycling

* Single-level dehydrate/rehydrate

* Checkpoint blocking

All implementations MUST support Layer 0.

3.2. Layer 1: Recursive Extension

Layer 1 adds hierarchical processing capabilities:

* Scoped Entity ID namespaces (collection -> document -> part -> job)

* Explicit Depth tracking in status frames

* SCOPE_DIGEST for Merkle-based subtree completion

* BARRIER for subtree-scoped synchronization

* Nested dehydration with depth tracking

Layer 1 is OPTIONAL. Implementations advertise Layer 1 support during capability negotiation.

3.3. Layer 2: Resilience Extension

Layer 2 adds fault tolerance and async processing:

* YIELDED status with continuation tokens

* DEFERRED status with claim checks

* RETRYING, SKIPPED, ABANDONED statuses

* Completion policies (STRICT, LENIENT, BEST_EFFORT, QUORUM)

* Claim check extensions and deferred processing tokens

* Stopping point validation

Layer 2 is OPTIONAL and requires Layer 1. Implementations advertise Layer 2 support during capability negotiation.

3.4. Capability Negotiation

PipeStream uses a two-tier negotiation model. The ALPN identifier (Section 11.1) identifies the base PipeStream transport mapping, while the capabilities structure handles dynamic resource limits and optional layer support that may vary based on endpoint configuration or real-time load.

During CONNECT, endpoints exchange supported capabilities using the capabilities structure. This message MUST be encoded using the default CBOR format for both the client's initiation and the server's response (Section 3.5).

```
serialization-format = &(
  cbor: 0,                          ; Default (IETF native)
  protobuf: 1,
)

capabilities = {
  layer0-core: bool,                ; Always true
  layer1-recursive: bool,           ; Scoped IDs, digests
  layer2-resilience: bool,          ; Yield, claim checks
  ? max-scope-depth: uint .le 7,    ; Default: 7 (8 levels, 0-7)
  ? max-entities-per-scope: uint,   ; Default: 4,294,967,294
  ? max-window-size: uint,          ; Default: 2,147,483,648 (2^31)
                                    ; (Max in-flight entities)
  ? serialization-format: serialization-format, ; Default: CBOR
  ? keepalive-timeout-ms: uint,     ; Default: 30000 (30s)
}
```

Peers negotiate down to common capabilities. If Layer 2 is requested but Layer 1 is not supported, Layer 2 MUST be disabled.

3.4.1. Version Negotiation

PipeStream protocol versioning is carried in two places: the ALPN identifier and the Ver field in the STATUS frame (Section 6.2.1). The ALPN identifier pipestream/1 identifies the major protocol version and the QUIC transport mapping defined in this document. The 4-bit Ver field in STATUS frames carries the value 0x1 for this specification.

A future major version of PipeStream (e.g., pipestream/2) would register a new ALPN identifier. QUIC's native ALPN negotiation during the TLS handshake provides version selection: if a client offers both pipestream/2 and pipestream/1 and the server supports only pipestream/1, the TLS handshake selects pipestream/1 without additional round trips. This mechanism is consistent with the versioning approach used by HTTP/3 [RFC9114] and DNS over QUIC [RFC9250].

Minor, backward-compatible extensions (such as new optional capability fields or new status codes within the reserved ranges) do not require a new ALPN identifier. Such extensions are negotiated through the capabilities structure or the IANA registries defined in Section 11.

3.4.2. Serialization Format Negotiation

The serialization_format field determines the encoding used for all variable-length control messages (frame types 0x80-0xFF) and entity headers. Negotiation proceeds as follows:

1. Each peer advertises its preferred serialization_format in its Capabilities message.

2. If both peers advertise the same format, that format is used.

3. If a peer receives a Capabilities message without serialization_format, the sender is assumed to prefer CBOR [RFC8949].

4. If the resulting preferences differ, the peers MUST use CBOR [RFC8949] as the fallback.

The initial Capabilities exchange on a new connection MUST use the default CBOR format for both the client's initiation and the server's response. The negotiated serialization format and resource limits take effect immediately following the successful completion of this initial Capabilities exchange (one request and one response). If a peer cannot decode the initial Capabilities exchange, it MUST close the connection with PIPESTREAM_INTERNAL_ERROR (0x01).

4. Protocol Overview

This section provides a high-level overview of the PipeStream protocol architecture, design principles, and operational model.

4.1. Design Goals

4.1.1. True Streaming Processing

PipeStream MUST enable true streaming processing where entities are transmitted and processed incrementally as they become available. Implementations MUST NOT buffer complete inputs before initiating transmission.

4.1.2. Recursive Decomposition

The protocol MUST support recursive decomposition of entities, wherein a single input entity MAY produce zero, one, or many output entities.

4.1.3. Checkpoint Consistency

PipeStream MUST provide checkpoint blocking semantics to maintain processing consistency across distributed workers.

4.1.4. Control and Data Plane Separation

The protocol MUST maintain strict separation between the control plane (control stream) and the data plane (entities).

4.1.5. QUIC Foundation

PipeStream MUST be implemented over QUIC [RFC9000] to leverage:

* Native stream multiplexing without head-of-line blocking
* Built-in flow control at both connection and stream levels
* TLS 1.3 security by default
* Connection migration capabilities

4.1.6. Multi-Layer Data Representation

The protocol MUST support four distinct data representation layers:

Layer	Name	Description
0	BlobBag	Raw binary data with metadata
1	SemanticLayer	Annotated content with embeddings
2	ParsedData	Structured extracted information
3	CustomEntity	Application-specific extension

Table 2

4.2. Architecture Summary

PipeStream uses a dual-stream architecture within a single QUIC connection between a Client (Producer) and Server (Consumer):

Stream	Type	Plane	Content
Stream 0	Bidirectional	Control	STATUS, SCOPE_DIGEST, BARRIER, GOAWAY, CAPABILITIES, CHECKPOINT
Streams 2+	Unidirectional	Data	Entity frames (Header + Payload)

Table 3

4.3. Connection Lifecycle

A PipeStream connection follows this lifecycle:

1. *Establishment:* Client initiates QUIC connection with ALPN identifier "pipestream/1"

2. *Capability Exchange:* Client and server exchange supported protocol layers and limits

3. *Control Stream Initialization:* Client opens Stream 0 as bidirectional Control Stream

4. *Entity Streaming:* Entities are transmitted per Sections 5 and 6

5. *Termination:* Connection closes via GOAWAY-initiated graceful shutdown (Section 6.5) or QUIC CONNECTION_CLOSE

5. QUIC Stream Mapping

PipeStream leverages the native multiplexing capabilities of QUIC [RFC9000] to provide a clean separation between control coordination and data transmission.

5.1. Control Stream (Stream 0)

The Control Stream provides the control plane for PipeStream operations.

5.1.1. Stream Identification

The Control Stream MUST use QUIC Stream ID 0, which per RFC 9000 is a bidirectional, client-initiated stream.

5.1.2. Usage Rules

1. The Control Stream MUST be opened immediately upon connection establishment.

2. Capability negotiation (Section 3.4) MUST occur on Stream 0 before any Entity Streams are opened.

3. Stream 0 MUST NOT carry entity payload data.

4. Implementations SHOULD assign the Control Stream a high priority to ensure timely delivery of status updates. An implementation MAY choose a different priority policy when operating in constrained environments where QUIC stream scheduling overhead must be minimized.

5.1.3. Flow Control Considerations

The Control Stream carries small, bit-packed control frames. STATUS frames are 16 octets base. Implementations MUST ensure adequate flow control credits:

* The initial MAX_STREAM_DATA for Stream 0 SHOULD be at least 8192 octets. A lower value is permissible for extremely constrained devices but risks stalling status delivery.

* Implementations SHOULD NOT block Entity Stream transmission due to Control Stream flow control exhaustion. In rare cases where strict ordering between control and data planes is required by the application, an implementation MAY temporarily pause entity transmission until control stream credits are replenished.

5.1.4. Heartbeat Mechanism

QUIC already provides native transport liveness signals (for example, PING and idle timeout handling). Implementations SHOULD rely on those transport mechanisms for connection liveness.

PipeStream heartbeat frames are OPTIONAL and are intended for application-level responsiveness checks (for example, detecting stalled processing logic even when the transport remains healthy). When used, an endpoint sends a STATUS frame with all fields set to their heartbeat values:

Field	Value	Description
Type	0x50 (STATUS)	
Stat	0x0 (UNSPECIFIED)	Heartbeat signal
Entity ID	0xFFFFFFFF	CONNECTION_LEVEL
Scope ID	0x00000000	Root scope
Reserved	0x00000000	MUST be zero

Table 4

The KEEPALIVE_TIMEOUT defaults to 30 seconds. Endpoints MAY negotiate a different value by including a keepalive-timeout-ms field (in milliseconds) in the capabilities exchange (Section 3.4); the effective timeout is the minimum of the two peers' advertised values.

When no status updates have been transmitted for KEEPALIVE_TIMEOUT, an endpoint MAY send a heartbeat frame. If no data is received on Stream 0 for 3 * KEEPALIVE_TIMEOUT, the endpoint SHOULD first apply transport-native liveness policy (e.g., QUIC PING); it MAY close the connection with PIPESTREAM_IDLE_TIMEOUT (0x02) when application-level inactivity policy requires it.

5.1.5. Transport Session vs. Application Session Context

The session-id segment identifies application context for detached or resumable resources (for example, Layer 2 yield/claim-check flows). PipeStream Layer 0 streaming semantics do not depend on this URI scheme.

5.1.6. Interaction with QUIC Flow Control and Congestion Control

PipeStream relies on the flow control and congestion control mechanisms provided by QUIC [RFC9000] and does not define its own transport-layer congestion control. QUIC provides flow control at both the stream level (MAX_STREAM_DATA) and the connection level (MAX_DATA). PipeStream's cursor-based backpressure (Section 9.1) operates at the application layer and is complementary to QUIC flow control:

* QUIC flow control limits the number of bytes in flight on any given stream or connection.

* PipeStream backpressure limits the number of entities in flight (i.e., the number of Entity IDs between cursor and last_assigned).

When the QUIC connection-level flow control window is exhausted, new Entity Streams cannot transmit data regardless of whether PipeStream's entity window has capacity. Conversely, when PipeStream's entity window is full, no new Entity Streams are opened even if QUIC flow control credits are available. Implementations MUST respect both limits. An implementation SHOULD monitor QUIC-level flow control credit availability and avoid opening new Entity Streams when connection-level credits are below the expected entity size, to prevent head-of-line blocking across streams sharing the connection budget.

5.2. Entity Streams (Streams 2+)

Entity Streams carry the actual entity data.

5.2.1. Unidirectional Data Flow

Entity Streams MUST be unidirectional streams:

Stream Type	Client to Server	Server to Client
Client-Initiated	4n + 2 (n >= 0)	2, 6, 10, 14, ...
Server-Initiated	4n + 3 (n >= 0)	3, 7, 11, 15, ...

Table 5

5.2.2. One Entity Per Stream

1. Each Entity Stream MUST carry exactly one entity.

2. The entity_id in the Entity Frame header MUST be unique within its scope.

3. Once an entity has been completely transmitted, the sender MUST close the stream.

5.3. Transport Error Mapping

PipeStream error signaling on Stream 0 and QUIC transport signals are complementary. Endpoints SHOULD bridge them so peers receive both transport-level and protocol-level context. An implementation MAY omit bridging when operating as a simple pass-through proxy that does not inspect entity status.

1. If an Entity Stream is aborted with RESET_STREAM or STOP_SENDING, the endpoint SHOULD emit a corresponding terminal status (FAILED, ABANDONED, or policy-driven equivalent) for that entity on Stream 0. The endpoint MAY omit the status frame only if the connection itself is being closed immediately.

2. If PipeStream determines a terminal entity error first (for example, checksum failure or invalid frame), the endpoint SHOULD abort the affected Entity Stream with an appropriate QUIC error and emit the corresponding PipeStream status/error context on Stream 0. Aborting the stream MAY be deferred if the entity payload is still needed for diagnostic purposes.

3. If Stream 0 is reset or becomes unusable, endpoints SHOULD treat this as a control-plane failure and close the connection with PIPESTREAM_CONTROL_RESET (0x03). An endpoint that can recover control-plane state through an application-layer mechanism MAY attempt reconnection before closing.

4. On QUIC connection termination (CONNECTION_CLOSE), entities without a previously observed terminal status MUST be treated as failed by local policy.

6. Frame Formats

This section defines the wire formats for PipeStream frames. All multi-octet integer fields are encoded in network byte order (big-endian).

Forward Compatibility: All Reserved and Flags fields defined in this section MUST be set to zero when sent and MUST be ignored by receivers. This convention enables future specifications to assign meaning to currently-reserved bits without breaking deployed implementations. Receivers that encounter non-zero values in reserved fields MUST NOT treat this as an error.

6.1. Control Stream Framing (Stream 0)

To support mixed content (bit-packed frames and serialized messages) on the Control Stream, PipeStream uses a Unified Control Frame (UCF) header.

6.1.1. UCF Header

Every message on Stream 0 MUST begin with a 1-octet Frame Type.

Value	Frame Class	Length Encoding	Description
0x50-0x7F	Fixed	No length prefix	Bit-packed control frames with type-defined sizes
0x80-0xFF	Variable	4-octet Length + N	Variable-size serialized control messages (encoding per Section 3.5)

Table 6

For Fixed frames, the receiver determines frame size from the Frame Type value. For Variable frames, the Type is followed by a 4-octet unsigned integer (big-endian) indicating the length of the serialized message that follows. Handling of unknown frame types is specified in Section 11.2.1.

Variable-frame Length (32 bits): The payload length in octets,

excluding the 1-octet Type and the 4-octet Length field. Receivers MUST reject lengths greater than 16,777,215 octets (16 MiB - 1) with PIPESTREAM_ENTITY_TOO_LARGE (0x06).

6.1.2. Fixed Frame Sizes

The following fixed-size frame types are defined by this document:

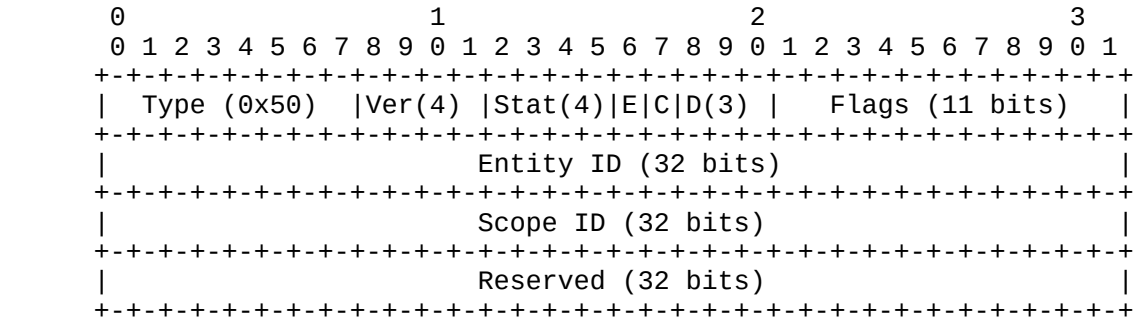
Type	Name	Total Size	Notes
0x50	STATUS	16 octets (base)	20 octets when C=1; larger when E=1 with extension data
0x54	SCOPE_DIGEST	72 octets	Includes 32-octet Merkle root and 64-bit counters
0x55	BARRIER	12 octets	No variable extension
0x56	GOAWAY	8 octets	Graceful shutdown signal

Table 7

6.2. Status Frames (Layer 0)

6.2.1. Status Frame Format (0x50)

The Status Frame reports lifecycle transitions for entities. The frame is 128-bit aligned for efficient parsing on 64-bit architectures.



Ver (4 bits): Protocol version. MUST be set to 0x1 for this specification. Receivers that encounter an unsupported version MUST close the connection with PIPESTREAM_LAYER_UNSUPPORTED (0x0C).

Stat (4 bits): Status code (see Section 6.2.2).

E (1 bit): Extended frame flag. If set, an Extension Header (Section 6.5) MUST follow the base frame (and any cursor update).

C (1 bit): Cursor update flag. A 4-octet cursor value follows (Section 6.2.3).

D (3 bits): Explicit scope nesting depth (0-7). 0=Root. Layer 1.

Flags (11 bits): Reserved for future use. MUST be zero when sent and MUST be ignored by receivers.

Entity ID (32 bits): Unsigned integer identifying the entity.

Scope ID (32 bits): Identifier for the scope to which this entity belongs. Expanding to 32 bits ensures uniqueness across high-frequency document sessions.

Reserved (32 bits): Reserved for future use. MUST be zero when sent and MUST be ignored by receivers.

6.2.2. Status Codes

Value	Name	Layer	Description
0x0	UNSPECIFIED	-	Default / heartbeat signal
0x1	PENDING	0	Entity announced, not yet transmitting
0x2	PROCESSING	0	Entity transmission in progress
0x3	COMPLETE	0	Entity successfully processed
0x4	FAILED	0	Entity processing failed
0x5	CHECKPOINT	0	Synchronization barrier
0x6	DEHYDRATING	0	Dehydrating into children
0x7	REHYDRATING	0	Rehydrating children
0x8	YIELDED	2	Paused with continuation token

0x9	DEFERRED	2	Detached with claim check
0xA	RETRYING	2	Retry in progress
0xB	SKIPPED	2	Intentionally skipped
0xC	ABANDONED	2	Timed out

Table 8

The base STATUS frame is 16 octets. When C=1, a 4-octet cursor value follows (total 20 octets). When E=1, an Extension Header follows all other STATUS fields.

6.2.3. Entity Status State Machine

The following table defines the complete set of valid status transitions. A receiver that observes a transition not listed in this table MUST treat the status frame as a protocol error and close the connection with PIPESTREAM_ENTITY_INVALID (0x05) as the QUIC Application Error Code.

From State	Valid Transitions (To)
PENDING	PROCESSING, DEHYDRATING, FAILED, SKIPPED, ABANDONED
PROCESSING	COMPLETE, FAILED, DEHYDRATING, CHECKPOINT, YIELDED, DEFERRED, ABANDONED
DEHYDRATING	REHYDRATING, FAILED, ABANDONED
REHYDRATING	COMPLETE, FAILED, ABANDONED
CHECKPOINT	PROCESSING
YIELDED	PROCESSING, FAILED, DEFERRED, ABANDONED
DEFERRED	PROCESSING, FAILED, SKIPPED, ABANDONED
FAILED	RETRYING, ABANDONED
RETRYING	PROCESSING, FAILED, ABANDONED
COMPLETE	(terminal -- no transitions)
SKIPPED	(terminal -- no transitions)
ABANDONED	(terminal -- no transitions)

Table 9

Notes:

1. PENDING is the implicit initial state for every entity upon ID assignment.

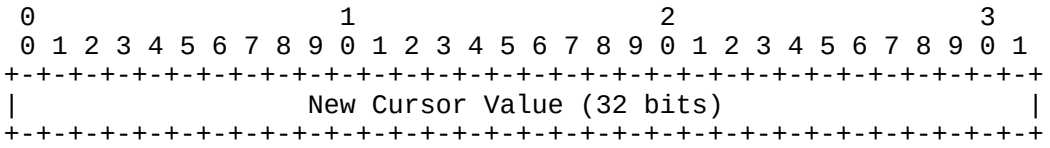
2. The FAILED -> RETRYING transition is valid only when the entity's completion policy permits retries (max-retries > 0) and the retry count has not been exhausted. If retries are not permitted or are exhausted, FAILED MUST be treated as a terminal state.

3. Layer 2 states (YIELDED, DEFERRED, RETRYING, SKIPPED, ABANDONED) MUST NOT appear when Layer 2 has not been negotiated. A receiver operating at Layer 0 or Layer 1 that observes a Layer 2 status code MUST treat it as PIPESTREAM_LAYER_UNSUPPORTED (0x0C).

4. The UNSPECIFIED (0x0) status is used only for heartbeat frames (Section 5.1.4) and connection-level signals. It is not a valid entity lifecycle state and MUST NOT appear in transitions for entity IDs other than 0xFFFFFFFF.

6.2.4. Cursor Update Extension

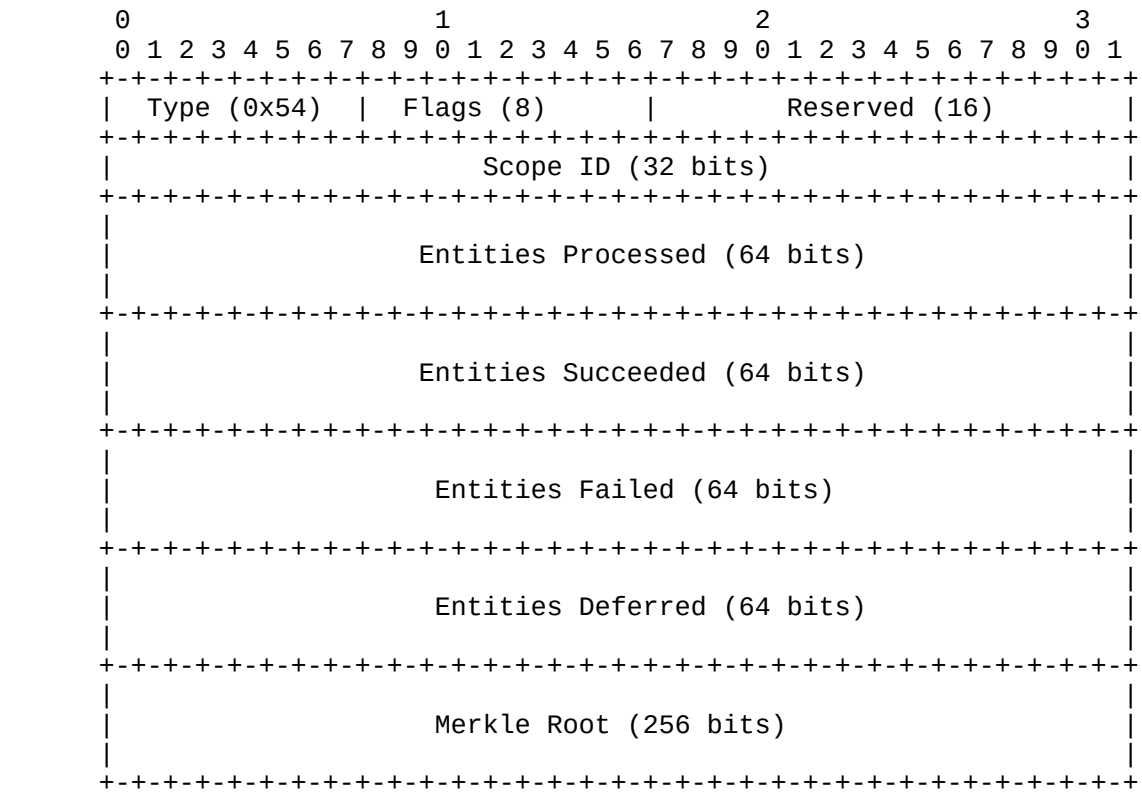
When C=1, a 4-octet cursor update follows the status frame:



New Cursor Value (32 bits): The numeric value of the new cursor. Entities with IDs lower than this value (modulo circular ID rules) are considered resolved and their IDs MAY be recycled.

6.3. Scope Digest Frame (0x54)

When Protocol Layer 1 is negotiated, a scope completion is summarized:



Flags (8 bits): Reserved for future use. MUST be zero when sent and MUST be ignored by receivers.

Scope ID (32 bits): Identifier of the scope being summarized.

Entities Processed (64 bits): The total number of entities that were processed within the scope.

Entities Succeeded (64 bits): The number of entities that reached a terminal success state.

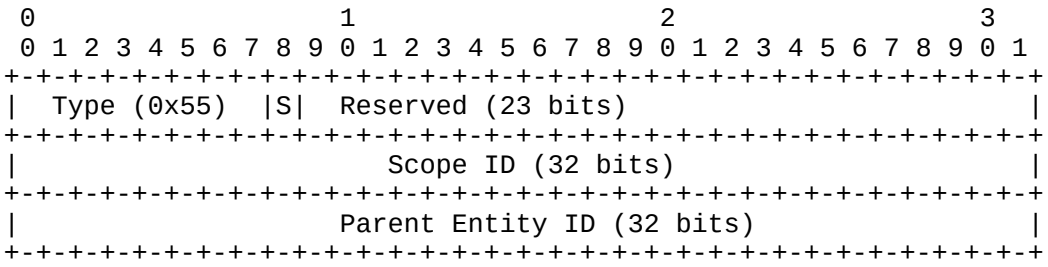
Entities Failed (64 bits): The number of entities that reached a terminal failure state.

Entities Deferred (64 bits): The number of entities that were deferred via claim checks.

Merkle Root (256 bits): The SHA-256 Merkle root covering all entity statuses in the scope (see Section 9.4).

The SCOPE_DIGEST frame is 72 octets total. The Scope ID MUST match the 32-bit identifier defined in Section 6.2.1.

6.4. Barrier Frame (0x55)



S (1 bit): Status (0 = waiting, 1 = released).

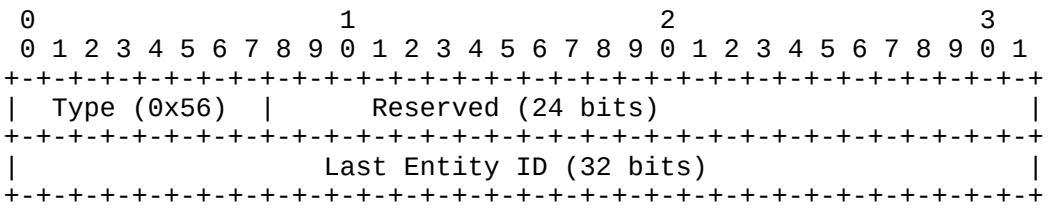
Reserved (23 bits): Reserved for future use. MUST be zero when sent and MUST be ignored by receivers.

Scope ID (32 bits): Identifier for the scope to which this barrier applies.

Parent Entity ID (32 bits): The identifier of the parent entity whose sub-tree is blocked by this barrier.

6.5. GOAWAY Frame (0x56)

The GOAWAY frame signals that the sender will not accept new entities beyond a specified Entity ID. It enables graceful shutdown: in-flight entities with IDs at or below the Last Entity ID are processed to completion, while the peer refrains from opening new Entity Streams.



Reserved (24 bits): Reserved for future use. MUST be zero when sent and MUST be ignored by receivers.

Last Entity ID (32 bits): The highest Entity ID that the sender will

process. Entities with IDs greater than this value (per the circular ordering defined in Section 9.1) MUST NOT be sent by the peer after receiving this frame.

6.5.1. Graceful Shutdown Procedure

1. An endpoint wishing to shut down sends a GOAWAY frame on Stream 0 with Last Entity ID set to the highest entity it is willing to process.

2. Upon receiving GOAWAY, the peer MUST NOT open new Entity Streams for entities with IDs above Last Entity ID. Entity Streams already open for IDs at or below Last Entity ID continue to completion.

3. Both peers continue processing status updates on Stream 0 until all in-flight entities reach terminal state.

4. Once all entities are resolved, either peer MAY close the QUIC connection with PIPESTREAM_NO_ERROR (0x00).

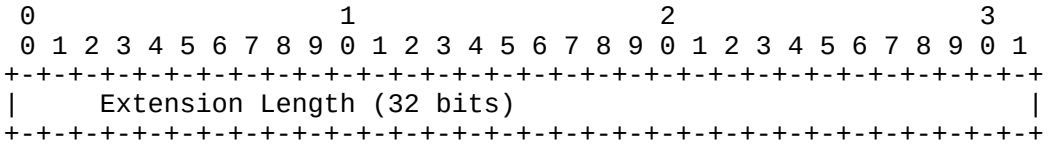
5. If an endpoint receives an entity with an ID above the Last Entity ID after sending GOAWAY, it MUST reject it with PIPESTREAM_ENTITY_INVALID (0x05).

An endpoint MAY send multiple GOAWAY frames to progressively lower the Last Entity ID. The Last Entity ID MUST NOT increase across successive GOAWAY frames within the same connection.

6.6. Yield and Claim Check Extensions (Layer 2)

When E=1 in a status frame, an Extension Header MUST follow.

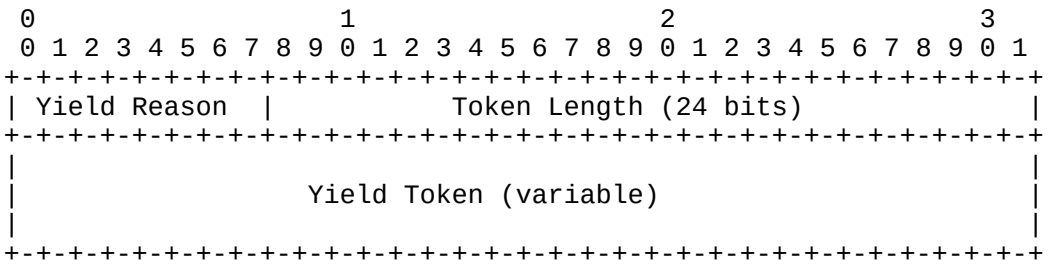
6.6.1. Extension Header



Extension Length (32 bits): The total length of the extension data that follows this header, in octets. This allows receivers that do not support specific status extensions to skip the extension data and continue parsing the control stream.

If E=1 is set for a Status code that does not define an extension layout in this specification (or a negotiated extension), the receiver MUST use the Extension Length to skip the data. If the length is zero or missing, the frame MUST be treated as malformed.

6.6.2. Yield Extension (Stat = 0x8)



Yield Reason (8 bits): The reason for yielding (see Section 6.5.1.1).

Token Length (24 bits): The length of the Yield Token in bytes (maximum 16,777,215).

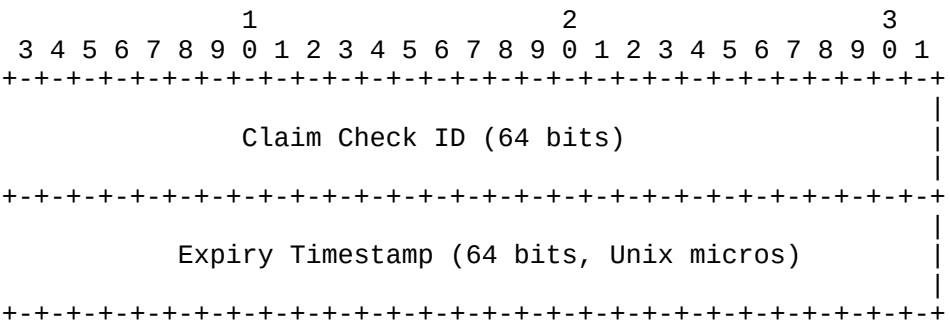
Yield Token (variable): The opaque continuation state.

6.6.2.1. Yield Reason Codes

Value	Name	Description
0x1	EXTERNAL_CALL	Waiting on external service
0x2	RATE_LIMITED	Voluntary throttle
0x3	AWAITING_SIBLING	Waiting for specific sibling
0x4	AWAITING_APPROVAL	Human/workflow gate
0x5	RESOURCE_BUSY	Semaphore/lock
0x0, 0x06-0xFF	Reserved	Reserved for future use

Table 10

6.6.3. Claim Check Extension (Stat = 0x9)



Claim Check ID (64 bits): A cryptographically secure random identifier for the claim.

Expiry Timestamp (64 bits): Unix epoch timestamp in microseconds when the claim expires.

6.7. Variable-Length Serialized Messages (0x80-0xFF)

Messages in this range are preceded by a 4-octet length field. The message body is encoded using the serialization format negotiated during capability exchange (Section 3.5). If no format was negotiated, CBOR [RFC8949] is the default.

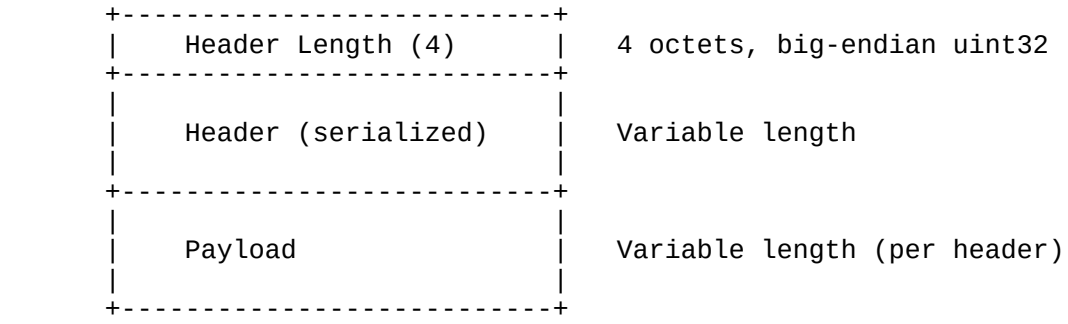
Type	Message Name	Reference
0x80	Capabilities	Section 3.4
0x81	Checkpoint	Section 9.3

Table 11

6.8. Entity Frames

Entity frames carry the actual document entity data on Entity Streams.

6.8.1. Entity Frame Structure



Header Length (4 octets): The length of the serialized EntityHeader in bytes.

Header (serialized): The EntityHeader message encoded in the negotiated serialization format (see Section 6.7.2).

Payload (variable): The raw entity data.

6.8.2. Message Schema (CDDL)

Normative definitions for serialized PipeStream messages use CDDL [RFC8610] notation. An informational Protocol Buffers equivalent is provided in Appendix D.

6.8.2.1. Entity Header

```
entity-header = {
  entity-id: uint,              ; 32-bit on wire (STATUS frame)
  ? parent-id: uint,            ; 32-bit scope-local
  ? scope-id: uint,             ; 32-bit (Section 6.2.1)
  layer: uint .le 3,            ; Data layer 0-3
  ? content-type: tstr,         ; MIME type
  payload-length: uint,         ; 32-bit (UCF header)
  ? checksum: bstr .size 32,    ; SHA-256; SHOULD be present
  ? metadata: { * tstr => tstr },
  ? chunk-info: chunk-info,
  ? completion-policy: completion-policy, ; Layer 2
}
```

6.8.2.2. Chunk Info

```
chunk-info = {
  total-chunks: uint,
  chunk-index: uint,
  chunk-offset: uint,
}
```

6.8.2.3. Yield and Deferral

```
yield-token = {
  reason: yield-reason,            ; See Appendix C for enum values
  ? continuation-state: bstr,
  ? validation: stopping-point-validation,
}

claim-check = {
  claim-id: uint,
  entity-id: uint,
  ? scope-id: uint,
  expiry-timestamp: uint,        ; Unix epoch microseconds
  ? validation: stopping-point-validation,
}
```

6.8.2.4. Support Types

```
entity-status = uint .size 1 ; Values 0x0-0xC per Section 6.2.2

stopping-point-validation = {
  ? state-checksum: bstr,        ; Hash of processing state
  ? bytes-processed: uint,       ; Progress marker
  ? children-complete: uint,
  ? children-total: uint,
  ? is-resumable: bool,
  ? checkpoint-ref: tstr,
}
```

6.8.3. Checksum Algorithm

PipeStream uses SHA-256 [FIPS-180-4] for payload integrity verification. The checksum MUST be exactly 32 octets.

7. Entity Model

PipeStream distinguishes between the wire-level entity representation (EntityHeader, Section 6.7.2) and the application-level document envelope (PipeDoc, this section). The EntityHeader is the on-the-wire structure that prefixes every entity payload on an Entity Stream; it carries the fields needed for transport-level processing (entity-id, layer, payload-length, checksum). The PipeDoc structure is an application-layer envelope carried within the entity payload itself, providing domain-specific identification (doc_id) and ownership tracking. A PipeDoc’s entity_id field MUST match the entity-id field in the enclosing EntityHeader.

7.1. Core Fields

Every PipeStream entity is represented as a PipeDoc message:

Field	Type	Requirement	Description
doc_id	string	REQUIRED	Unique document identifier (UUID recommended)
entity_id	uint32	REQUIRED	Scope-local identifier
ownership	OwnershipContext	OPTIONAL	Multi-tenancy tracking

Table 12

7.2. Four Data Layers

Each PipeDoc carries entity payload in one of four data layers:

Layer	Name	Content
0	BlobBag	Raw binary data: original document bytes, images, attachments
1	SemanticLayer	Annotated content: text segments with vector embeddings, NLP annotations, NER, classifications
2	ParsedData	Structured extraction: key-value pairs, tables, structured fields
3	CustomEntity	Extension point: domain-specific extension types

Table 13

7.3. Cloud-Agnostic Storage Reference

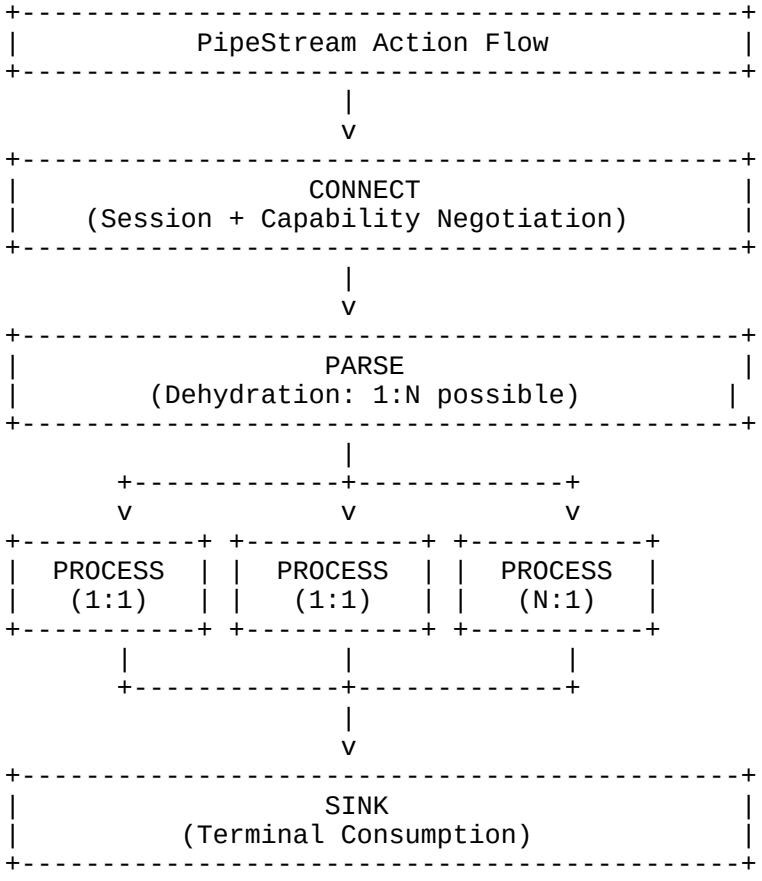
```
file-storage-reference = {
  provider: tstr,                ; Storage provider identifier
  bucket: tstr,                  ; Bucket/container name
  key: tstr,                     ; Object key/path
  ? region: tstr,                ; Optional region hint
  ? attrs: { * tstr => tstr },   ; Provider-specific attributes
  ? encryption: encryption-metadata,
}

encryption-metadata = {
  algorithm: tstr,               ; "AES-256-GCM", "AES-256-CBC"
  ? key-provider: tstr,          ; "aws-kms", "azure-keyvault",
                                 ; "gcp-kms", "vault"
  ? key-id: tstr,                ; Key ARN/URI/ID
  ? wrapped-key: bstr,           ; Client-side encrypted DEK
  ? iv: bstr,                    ; Initialization vector
  ? context: { * tstr => tstr }, ; Encryption context
}
```

8. Protocol Operations

This section defines the protocol-level operations that PipeStream endpoints perform during a session. These operations describe the phases of a PipeStream session lifecycle, from connection establishment through entity processing to terminal consumption.

8.1. Overview

A PipeStream session proceeds through four sequential actions:


PipeStream Action Flow
CONNECT
(Session + Capability Negotiation)
PARSE
(Dehydration: 1:N possible)
PROCESS
(1:1)
PROCESS
(1:1)
PROCESS
(N:1)
SINK
(Terminal Consumption)

Phase	Action	Cardinality	Description
1	CONNECT	1:1	Session establishment and capability negotiation
2	PARSE	1:N	Dehydration: decompose input into entities
3	PROCESS	1:1 or N:1	Transform, rehydrate, aggregate, or pass through entities (parallel)
4	SINK	N:1	Terminal consumption: index, store, or notify

Table 14

8.2. CONNECT Action

The CONNECT action establishes the session with capability negotiation.

8.2.1. ALPN Identifier

ALPN Protocol ID: pipestream/1

8.2.2. Capability Exchange

Immediately after QUIC handshake, peers exchange Capabilities messages on Stream 0.

The Capabilities exchange includes serialization format negotiation (Section 3.5). The agreed-upon format applies to all subsequent variable-length serialized messages on Stream 0 and to all entity headers on Entity Streams.

8.3. PARSE Action

The PARSE action performs dehydration with optional completion policy:

```
   completion-policy = {
     ? mode: completion-mode,
     ? max-retries: uint,           ; Default: 3
     ? retry-delay-ms: uint,        ; Default: 1000
     ? timeout-ms: uint,            ; Default: 300000 (5 min)
     ? min-success-ratio: float32,  ; For QUORUM mode
     ? on-timeout: failure-action,
     ? on-failure: failure-action,
   }

   completion-mode = &(
     unspecified: 0,               ; Default; treat as STRICT
     strict: 1,                    ; All children MUST complete
     lenient: 2,                   ; Continue with partial results
     best-effort: 3,               ; Complete with whatever succeeds
     quorum: 4,                    ; Need min-success-ratio
   )

   failure-action = &(
     unspecified: 0,               ; Default; treat as FAIL
     fail: 1,                      ; Propagate failure up
     skip: 2,                      ; Skip, continue with siblings
     retry: 3,                     ; Retry up to max-retries
     defer: 4,                     ; Create claim check, continue
   )
```

8.4. PROCESS Action

Mode	Description
TRANSFORM	1:1 entity transformation
REHYDRATE	N:1 merge of siblings from dehydration
AGGREGATE	N:1 with reduction function
PASSTHROUGH	Metadata-only modification

Table 15

8.5. SINK Action

Type	Description
INDEX	Search engine integration (Elasticsearch, Solr, etc.)
STORAGE	Blob storage persistence (Object stores, Cloud storage)
NOTIFICATION	Webhook/messaging triggers

Table 16

9. Rehydration Semantics

9.1. Entity ID Lifecycle and Cursor

Entity IDs are managed using a cursor-based circular recycling scheme within the 32-bit ID space. The ID space is divided into three logical regions relative to the current cursor and last_assigned pointers:

Region	ID Range	Description
Recyclable	IDs behind cursor	Resolved entities; IDs may be reused
In-flight	cursor to last_assigned	Active entities (PENDING, PROCESSING, etc.)
Free	Beyond last_assigned	Available for new entity assignment

Table 17

The window size is computed as (last_assigned - cursor) mod 0xFFFFFFFD. If window_size >= max_window, the sender MUST apply backpressure and stop assigning new IDs until the cursor advances.

Rules:

1. new_id = (last_assigned + 1) % 0xFFFFFFFD

2. If new_id == 0, new_id = 1 (skip reserved NULL_ENTITY)

3. If (new_id - cursor) % 0xFFFFFFFD >= max_window -> STOP, apply backpressure

4. On reaching a terminal state (COMPLETE, SKIPPED, ABANDONED, or FAILED with no remaining retries): mark resolved; if entity_id == cursor, advance cursor past all contiguous resolved IDs

5. IDs behind cursor are implicitly recyclable

An entity in the FAILED state that may still transition to RETRYING (Section 6.2.2a) MUST NOT be marked resolved. Only when retries are exhausted or no completion policy permits retries does FAILED become terminal for cursor purposes.

9.2. Assembly Manifest

The Assembly Manifest is a local data structure maintained by each endpoint to track the parent-child relationships created during dehydration. It is not transmitted on the wire; rather, each endpoint constructs its own manifest from the parent-id fields in received EntityHeaders and from status updates observed on the Control Stream. The CDDL below defines the logical structure of each entry; implementations MAY use any internal representation that preserves the required semantics.

Each Assembly Manifest entry tracks:

```
assembly-manifest-entry = {
  parent-id: uint,
  ? scope-id: uint,              ; Layer 1
  children-ids: [* uint],
  ? children-status: [* entity-status],
  ? policy: completion-policy,   ; Layer 2
  ? created-at: uint,
  ? state: resolution-state,
}

resolution-state = &(
  unspecified: 0,
  active: 1,
  resolved: 2,
  partial: 3,                  ; Some children failed/skipped
  failed: 4,
)
```

9.3. Checkpoint Blocking

A checkpoint is satisfied when:

1. All entities in the checkpoint scope with IDs less than checkpoint_entity_id (considering circular wrap) have reached terminal state.

2. All Assembly Manifest entries within the checkpoint scope have been resolved.

3. All nested checkpoints within the checkpoint scope have been satisfied.

CheckpointFrame (Section 6.6 / Appendix C) carries both:

```
checkpoint-frame = {
  checkpoint-id: tstr,
  sequence-number: uint,
  checkpoint-entity-id: uint,
  ? scope-id: uint,
  ? flags: uint,
  ? timeout-ms: uint,
}
```

* checkpoint_id: an opaque identifier for logging and correlation.

* checkpoint_entity_id: the numeric ordering key used for barrier evaluation.

Implementations MUST use checkpoint_entity_id (not checkpoint_id) when evaluating Condition 1.

For circular comparison in Condition 1, implementations MUST use the same modulo ordering as cursor management. Define MAX = 0xFFFFFFFD and:

```
is_before(a, b) = ((b - a + MAX) % MAX) < (MAX / 2)
```

An entity ID a is considered "less than checkpoint_entity_id b" iff is_before(a, b) is true.

9.4. Scope ID Allocation (Layer 1)

When Layer 1 is negotiated, Scope IDs are 32-bit unsigned integers assigned by the endpoint that initiates the dehydration. The allocation scheme is as follows:

1. Scope ID 0 is the root scope and MUST NOT be used for child scopes.

2. The dehydrating endpoint assigns a unique Scope ID to each new child scope created during dehydration. The Scope ID MUST be unique within the connection for the lifetime of that scope (i.e., until the scope's SCOPE_DIGEST frame has been emitted and acknowledged).

3. Scope IDs MAY be allocated sequentially or randomly; the protocol does not require any particular ordering. Sequential allocation is RECOMMENDED for simplicity and debuggability.

4. Once a scope has been closed (its SCOPE_DIGEST has been sent), the Scope ID MAY be reused for a new scope. Implementations MUST ensure that no in-flight status frames reference a recycled Scope ID; this is guaranteed if the implementation waits until all entities within the scope have reached terminal state before recycling.

9.5. Scope Digest Propagation (Layer 1)

When a scope completes, the endpoint MUST compute a Scope Digest and propagate it to the parent scope via a SCOPE_DIGEST frame (Section 6.3).

The Merkle root in the Scope Digest is computed as follows:

1. For each entity in the scope, ordered by Entity ID (ascending), construct a 5-octet leaf value by concatenating:

   * The 4-octet big-endian Entity ID.

   * A 1-octet status field where the lower 4 bits contain the Stat code (Section 6.2.2) and the upper 4 bits are zero.

2. Compute SHA-256 over each 5-octet leaf to produce leaf hashes.

3. Build a binary Merkle tree by repeatedly hashing pairs of sibling nodes: SHA-256(left || right). If the number of nodes at any level is odd, the last node is promoted to the next level without hashing.

4. The root of this tree is the merkle_root value in the SCOPE_DIGEST frame.

This construction is deterministic: any two implementations processing the same set of entity statuses MUST produce the same Merkle root.

9.6. Rehydration Readiness Tracking

Implementations MUST track Assembly Manifest resolution order using a mechanism that provides O(1) insertion and amortized O(log n) minimum extraction. The tracking mechanism MUST support efficient decrease-key operations to handle out-of-order status updates.

Implementations MAY use a Fibonacci heap or similar priority queue to satisfy these complexity requirements.

9.7. Stopping Point Validation (Layer 2)

When yielding or deferring, include validation:

```
stopping-point-validation = {
  ? state-checksum: bstr,        ; Hash of processing state
  ? bytes-processed: uint,       ; Progress marker
  ? children-complete: uint,
  ? children-total: uint,
  ? is-resumable: bool,
  ? checkpoint-ref: tstr,
}
```

10. Security Considerations

10.1. Transport Security

PipeStream inherits security from QUIC [RFC9000] and TLS 1.3 [RFC8446]. All connections MUST use TLS 1.3 or later. Implementations MUST NOT provide mechanisms to disable encryption.

10.2. Entity Payload Integrity

Each Entity SHOULD include a SHA-256 [FIPS-180-4] checksum in its EntityHeader (the checksum field defined in Section 6.7.2). The checksum is OPTIONAL in the wire format to accommodate zero-length entities, streamed entities whose final length is unknown at header-emission time, and scenarios where application-layer integrity mechanisms provide equivalent guarantees. When a checksum is present, it MUST be exactly 32 octets containing the SHA-256 digest computed over the raw payload bytes (the octet sequence following the EntityHeader on the Entity Stream). The checksum does not cover the EntityHeader itself.

For chunked entities (where chunk-info is present in the EntityHeader), each chunk MAY carry its own per-chunk checksum. The checksum in the first chunk's EntityHeader, if present, MUST cover only that chunk's payload bytes. An implementation that requires whole-entity integrity verification MUST either compute a rolling digest across all chunks or require the sender to transmit a final summary entity containing the whole-payload checksum.

To support true streaming of large entities, implementations MAY begin processing an entity payload before the complete payload has been received and verified. However, the final rehydration or terminal SINK operation MUST NOT be committed until the complete payload checksum has been verified.

If a checksum verification fails, the implementation MUST:

1. Reject the entity with PIPESTREAM_INTEGRITY_ERROR (0x04).

2. Discard any partial results or temporary state associated with the entity.

3. Propagate the failure according to the Completion Policy (Section 8.3).

Implementations that require immediate consistency SHOULD buffer the entire entity and verify the checksum before initiating processing.

10.2.1. Algorithm Agility

This specification mandates SHA-256 [FIPS-180-4] as the sole checksum algorithm for both payload integrity (this section) and Merkle tree construction (Section 9.4). SHA-256 is well-studied and widely deployed; however, future developments may necessitate migration to a different algorithm.

PipeStream supports algorithm migration through the capability negotiation mechanism (Section 3.4). A future specification MAY define additional fields in the capabilities structure to advertise supported checksum algorithms, following the general principles outlined in [RFC7696]. Until such negotiation is defined, all implementations MUST use SHA-256 when producing or verifying checksums. An implementation that receives a checksum of a length other than 32 octets MUST reject the entity with PIPESTREAM_INTEGRITY_ERROR (0x04).

The checksum field in the EntityHeader is typed as bstr .size 32 in the CDDL schema (Appendix C). A future algorithm negotiation extension would need to update this constraint, the SCOPE_DIGEST Merkle root size, and the corresponding IANA registry entries.

10.3. Resource Exhaustion

Limit	Default	Description
Max scope depth	7	Prevents recursive bombs (8 levels: 0-7)
Max entities per scope	4,294,967,294	Memory bounds
Max window size	2,147,483,648	Max in-flight entities (2^31)
Checkpoint timeout	30s	Prevents stuck state
Claim check expiry	86400s	Garbage collection

Table 18

Implementations MUST enforce all resource limits listed above. Exceeding any limit MUST result in the corresponding error code (see Section 11.4). Implementations SHOULD allow operators to configure stricter limits than the defaults shown here.

To prevent memory-exhaustion attacks, implementations MUST NOT pre-allocate memory for variable-length payloads based solely on the 32-bit Length field in the UCF header (Section 6.1.1). Memory MUST be allocated incrementally as octets are received, or capped at a smaller initial buffer until the message type and context are verified.

10.4. Amplification Attacks

A single dehydration operation can produce an arbitrary number of child entities from a small input, creating a potential amplification vector. To mitigate this:

1. Implementations MUST enforce the max_entities_per_scope limit negotiated during capability exchange (Section 3.4). Any dehydration that would exceed this limit MUST be rejected.

2. Implementations MUST enforce the max_scope_depth limit. A dehydration chain deeper than this limit MUST be rejected with PIPESTREAM_DEPTH_EXCEEDED (0x07).

3. Implementations SHOULD enforce a configurable ratio between input entity size and total child entity count. A recommended default is no more than 1,000 children per megabyte of parent payload.

4. The backpressure mechanism (Section 9.1) provides a natural throttle: when the in-flight window fills, no new Entity IDs can be assigned until existing entities complete and the cursor advances. Implementations MUST NOT bypass backpressure for dehydration-generated entities.

10.5. Privacy Considerations

PipeStream entity headers and control stream frames carry metadata that may reveal information about the entities being processed, even when payloads are encrypted at the application layer:

1. *Entity structure leakage*: The number of child entities produced by dehydration, the scope depth, and the Entity ID assignment pattern may reveal the structure of the input being processed (e.g., an entity that dehydrates into 50 children is likely a multi-part input). Implementations that require structural privacy SHOULD pad dehydration counts or use fixed decomposition granularity. Deployments that do not handle privacy-sensitive data MAY omit this padding.

2. *Metadata in headers*: The content_type, metadata map, and payload_length fields in EntityHeader (Section 6.7) are transmitted in cleartext within the QUIC-encrypted stream. Implementations that require metadata confidentiality beyond transport encryption SHOULD encrypt EntityHeader fields at the application layer and use an opaque content_type such as application/octet-stream. This overhead is unnecessary when the deployment operates within a trusted network.

3. *Traffic analysis*: The timing and size of status frames on the Control Stream may correlate with processing patterns. Implementations operating in privacy-sensitive environments SHOULD send status frames at fixed intervals with padding to obscure processing timing. Deployments in trusted environments MAY omit traffic padding to reduce bandwidth overhead.

4. *Identifiers*: The doc_id field in PipeDoc (Section 7.1) and filenames in BlobBag entries are application-layer data but may be logged by intermediate processing nodes. Implementations SHOULD provide mechanisms to redact or pseudonymize identifiers at pipeline boundaries. This recommendation may be relaxed when all nodes in the pipeline are operated by the same administrative entity.

10.6. Replay and Token Reuse

10.6.1. Yield Token Replay

Yield tokens (Section 6.5.1) contain opaque continuation state that enables resumption of paused entity processing. A replayed yield token could cause an entity to be processed multiple times or to resume from a stale state. To prevent this:

1. Implementations MUST associate each yield token with a stable application context identifier (for example, a session identifier) and Entity ID. In Layer 0-only operation, this context MAY be implicit in the active transport connection. For Layer 2 resumptions that can occur across reconnects or different nodes, the context identifier MUST remain stable across transport connections. A yield token MUST be rejected if presented in a different context than the one that issued it, unless the token was explicitly transferred via a claim check.

2. Implementations MUST invalidate a yield token after it has been consumed for resumption. A second resumption attempt with the same token MUST be rejected.

3. The StoppingPointValidation (Section 9.6) provides integrity checking at resume time. Implementations MUST verify the state_checksum field before accepting a resumed entity. If the checksum does not match the current state, the resumption MUST be rejected and the entity MUST be reprocessed from the beginning.

10.6.2. Claim Check Replay

Claim checks (Section 6.5.2) are long-lived references that can be redeemed in different sessions. To prevent misuse:

1. Each claim check carries an expiry_timestamp (Unix epoch microseconds). Implementations MUST reject expired claim checks.

2. Implementations MUST track redeemed claim check IDs and reject duplicate redemptions. The tracking state MUST persist for at least the claim check expiry duration.

3. Claim check IDs MUST be generated using a cryptographically secure random number generator to prevent guessing.

10.7. Encryption Key Management

When using FileStorageReference with encryption:

1. Key IDs MUST reference keys in approved providers.
2. Wrapped keys MUST use approved envelope encryption.
3. Key rotation MUST be supported via key_id versioning.
4. Implementations MUST NOT log key material.
5. Implementations MUST NOT include unwrapped data encryption keys in EntityHeader metadata or Control Stream frames.

11. IANA Considerations

This document requests the creation of several new registries and one ALPN identifier registration. All registries defined in this section use the "Expert Review" policy [RFC8126] for new assignments unless otherwise stated.

11.1. ALPN Identifier Registration

This document registers the following ALPN [RFC7301] protocol identifier:

Protocol: PipeStream Version 1

Identification Sequence: 0x70 0x69 0x70 0x65 0x73 0x74 0x72 0x65 0x61 0x6D 0x2F 0x31 ("pipestream/1")

Specification: This document

11.2. PipeStream Frame Type Registry

IANA is requested to create the "PipeStream Frame Types" registry. Values are categorized into Fixed (type-sized, no length prefix) frames in 0x50-0x7F and Variable (4-octet length prefix) frames in 0x80-0xFF. Values 0xC0-0xFF are reserved for private use.

Value	Frame Type Name	Class	Size	Layer	Reference
0x50	STATUS	Fixed	16 octets base	0	Section 6.2
0x54	SCOPE_DIGEST	Fixed	72 octets	1	Section 6.3
0x55	BARRIER	Fixed	12 octets	1	Section 6.4
0x56	GOAWAY	Fixed	8 octets	0	Section 6.5
0x57-0x7F	Reserved	Fixed	-	-	this document
0x80	CAPABILITIES	Var	Length-prefixed	0	Section 3.4
0x81	CHECKPOINT	Var	Length-prefixed	0	Section 9.3
0x82-0xBF	Reserved	Var	-	-	this document

Table 19

11.2.1. Unknown Frame Handling

Receivers that encounter a Variable-class frame type (0x80-0xFF) that they do not recognize MUST skip the frame by reading and discarding the number of octets indicated by the 4-octet length prefix. Receivers that encounter an unknown Fixed-class frame type (0x50-0x7F) for which no size is defined MUST close the connection with PIPESTREAM_ENTITY_INVALID (0x05), since the frame size cannot be determined. Future specifications that register new Fixed-class frame types MUST define the frame size in the registry entry.

11.3. PipeStream Status Code Registry

IANA is requested to create the "PipeStream Status Codes" registry. Status codes are 4-bit values (0x0-0xF). Values 0xD-0xE are reserved for Expert Review. Value 0xF is reserved for private use.

Value	Name	Layer	Description
0x0	UNSPECIFIED	-	Default / heartbeat
0x1	PENDING	0	Entity announced
0x2	PROCESSING	0	In progress
0x3	COMPLETE	0	Success
0x4	FAILED	0	Failed
0x5	CHECKPOINT	0	Barrier
0x6	DEHYDRATING	0	Dehydrating into children
0x7	REHYDRATING	0	Rehydrating children
0x8	YIELDED	2	Paused
0x9	DEFERRED	2	Claim check issued
0xA	RETRYING	2	Retry in progress
0xB	SKIPPED	2	Intentionally skipped
0xC	ABANDONED	2	Timed out

Table 20

11.4. PipeStream Error Code Registry

IANA is requested to create the "PipeStream Error Codes" registry. Values in the range 0x00-0x3F are assigned by Expert Review. Values in the range 0x40-0xFF are reserved for private use.

PipeStream error codes are used as QUIC application error codes in CONNECTION_CLOSE and RESET_STREAM frames. When terminating a connection or aborting a stream due to a protocol-level error, the endpoint MUST use the corresponding PipeStream error code value as the QUIC Application Error Code.

Value	Name	Description
0x00	PIPESTREAM_NO_ERROR	Graceful shutdown
0x01	PIPESTREAM_INTERNAL_ERROR	Implementation error
0x02	PIPESTREAM_IDLE_TIMEOUT	Idle timeout
0x03	PIPESTREAM_CONTROL_RESET	Control stream must reset
0x04	PIPESTREAM_INTEGRITY_ERROR	Checksum failed
0x05	PIPESTREAM_ENTITY_INVALID	Invalid format or state
0x06	PIPESTREAM_ENTITY_TOO_LARGE	Size exceeded
0x07	PIPESTREAM_DEPTH_EXCEEDED	Scope depth exceeded
0x08	PIPESTREAM_WINDOW_EXCEEDED	Window full
0x09	PIPESTREAM_SCOPE_INVALID	Invalid scope
0x0A	PIPESTREAM_CLAIM_EXPIRED	Claim check expired
0x0B	PIPESTREAM_CLAIM_NOT_FOUND	Claim check not found
0x0C	PIPESTREAM_LAYER_UNSUPPORTED	Protocol layer not supported

Table 21

11.5. PipeStream Serialization Format Registry

IANA is requested to create the "PipeStream Serialization Formats" registry. New entries require Expert Review [RFC8126].

Value	Name	Description	Reference
0	CBOR	Concise Binary Object Representation	RFC 8949, this document
1	PROTOBUF	Protocol Buffers (see Appendix D)	this document
2-255	Reserved	Reserved for future use	this document

Table 22

11.6. URI Scheme Registration

This section registers the "pipestream" URI scheme per [RFC7595]. The URI scheme identifies application context for detached or resumable resources (for example, Layer 2 yield/claim-check flows). PipeStream Layer 0 streaming semantics do not depend on this URI scheme.

Scheme name: pipestream

Status: Permanent

Applications/protocols that use this scheme: PipeStream protocol (this document)

Contact: Kristian Rickert (kristian.rickert@pipestream.ai)

Change controller: IETF

11.6.1. URI Syntax

The URI syntax is defined using ABNF [RFC5234]:

```
pipestream-URI = "pipestream://" authority "/" session-id
                 [ "/" scope-path ] [ "/" entity-ref ]

session-id     = 1*( ALPHA / DIGIT / "-" )
scope-path     = scope-id *( "." scope-id )
scope-id       = 1*DIGIT
entity-ref     = 1*( ALPHA / DIGIT )
authority      = <authority, see RFC 3986, Section 3.2>
```

Examples:

* pipestream://processor.example.com/a1b2c3d4

* pipestream://processor.example.com:8443/a1b2c3d4/1.42/e5f6

12. References

12.1. Normative References

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

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/rfc/rfc2119>.

[RFC5234] Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, DOI 10.17487/RFC5234, January 2008, <https://www.rfc-editor.org/rfc/rfc5234>.

[RFC7301] Friedl, S., Popov, A., Langley, A., and E. Stephan, "Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension", RFC 7301, DOI 10.17487/RFC7301, July 2014, <https://www.rfc-editor.org/rfc/rfc7301>.

[RFC7595] Thaler, D., Ed., Hansen, T., and T. Hardie, "Guidelines and Registration Procedures for URI Schemes", BCP 35, RFC 7595, DOI 10.17487/RFC7595, June 2015, <https://www.rfc-editor.org/rfc/rfc7595>.

[RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, DOI 10.17487/RFC8126, June 2017, <https://www.rfc-editor.org/rfc/rfc8126>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/rfc/rfc8174>.

[RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <https://www.rfc-editor.org/rfc/rfc8446>.

[RFC8610] Birkholz, H., Vigano, C., and C. Bormann, "Concise Data Definition Language (CDDL): A Notational Convention to Express Concise Binary Object Representation (CBOR) and JSON Data Structures", RFC 8610, DOI 10.17487/RFC8610, June 2019, <https://www.rfc-editor.org/rfc/rfc8610>.

[RFC8949] Bormann, C. and P. Hoffman, "Concise Binary Object Representation (CBOR)", STD 94, RFC 8949, DOI 10.17487/RFC8949, December 2020, <https://www.rfc-editor.org/rfc/rfc8949>.

[RFC9000] Iyengar, J., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based Multiplexed and Secure Transport", RFC 9000, DOI 10.17487/RFC9000, May 2021, <https://www.rfc-editor.org/rfc/rfc9000>.

12.2. Informative References

[MOQT] Curley, L., Pugin, K., Nandakumar, S., Vasiliev, V., and I. Swett, "Media over QUIC Transport", Work in Progress, Internet-Draft, draft-ietf-moq-transport-16, 2025, <https://datatracker.ietf.org/doc/draft-ietf-moq-transport/>.

[RFC7574] Bakker, A., Petrocco, R., and V. Grishchenko, "Peer-to-Peer Streaming Peer Protocol (PPSPP)", RFC 7574, DOI 10.17487/RFC7574, July 2015, <https://www.rfc-editor.org/rfc/rfc7574>.

[RFC7696] Housley, R., "Guidelines for Cryptographic Algorithm Agility and Selecting Mandatory-to-Implement Algorithms", BCP 201, RFC 7696, DOI 10.17487/RFC7696, November 2015, <https://www.rfc-editor.org/rfc/rfc7696>.

[RFC9114] Bishop, M., Ed., "HTTP/3", RFC 9114, DOI 10.17487/RFC9114, June 2022, <https://www.rfc-editor.org/rfc/rfc9114>.

[RFC9250] Huitema, C., Dickinson, S., and A. Mankin, "DNS over Dedicated QUIC Connections", RFC 9250, DOI 10.17487/RFC9250, May 2022, <https://www.rfc-editor.org/rfc/rfc9250>.

[RFC9260] Stewart, R., T端xen, M., and K. Nielsen, "Stream Control Transmission Protocol", RFC 9260, DOI 10.17487/RFC9260, June 2022, <https://www.rfc-editor.org/rfc/rfc9260>.

[RFC9297] Schinazi, D. and L. Pardue, "HTTP Datagrams and the Capsule Protocol", RFC 9297, DOI 10.17487/RFC9297, August 2022, <https://www.rfc-editor.org/rfc/rfc9297>.

[RFC9308] K端hlewind, M. and B. Trammell, "Applicability of the QUIC Transport Protocol", RFC 9308, DOI 10.17487/RFC9308, September 2022, <https://www.rfc-editor.org/rfc/rfc9308>.

[scatter-gather] Lea, D., "The Scatter-Gather Design Pattern", DOI 10.1007/978-1-4612-1260-6, 1996, <https://doi.org/10.1007/978-1-4612-1260-6>.

Appendix A. Appendix A: Protocol Layer Capability Matrix

Feature	Layer 0	Layer 1	Layer 2
Unified status frame (128-bit base)	X	X	X
Entity streaming	X	X	X
PENDING/PROCESSING/COMPLETE/ FAILED	X	X	X
DEHYDRATING/REHYDRATING	X	X	X
Checkpoint blocking	X	X	X
Assembly Manifest	X	X	X
Cursor-based ID recycling	X	X	X
Scoped status fields (Scope ID, depth)		X	X
Hierarchical scopes		X	X
Scope digest (Merkle)		X	X
Barrier (subtree sync)		X	X
YIELDED status			X
DEFERRED status			X
Claim checks			X

Completion policies			X
SKIPPED/ABANDONED statuses			X

Table 23

Appendix B. Appendix B: Relationship to Existing Protocols

This appendix discusses the relationship between PipeStream and existing transport and application protocols. The intent is to clarify the design rationale for specifying a new application protocol directly over QUIC [RFC9000] rather than layering on HTTP/3 [RFC9114], gRPC, or WebTransport [RFC9297].

B.1. QUIC as Application Transport

RFC 9308 [RFC9308] provides guidance for designers of application protocol mappings to QUIC. PipeStream follows this guidance: stream semantics are mapped to the protocol's data model (Section 4), flow control operates at both the stream and connection level, and an ALPN token is registered for protocol identification (Section 11).

The precedent for application protocols that bypass HTTP and map directly onto QUIC is well established. DNS over Dedicated QUIC Connections [RFC9250] adopts a direct mapping on the grounds that HTTP framing introduces unnecessary overhead when the application has its own message semantics. The Media over QUIC Transport protocol [MOQT] similarly defines its own framing and control messages over QUIC streams, with HTTP/3 as an optional encapsulation rather than a requirement.

PipeStream's requirements align with these precedents. The protocol's dual-stream architecture (Section 4), bit-packed control frames (Section 6), and recursive entity lifecycle (Section 9) have no counterpart in HTTP semantics, and mapping them onto request-response pairs would add complexity without benefit.

B.2. Efficiency Considerations

The decision to implement PipeStream as a standalone protocol over QUIC is driven by the following efficiency requirements:

B.2.1. Framing Overhead

HTTP/3 and gRPC introduce multiple layers of framing. A gRPC message over HTTP/3 incurs QUIC stream overhead, HTTP/3 DATA frame overhead (minimum 2 octets), and gRPC envelope overhead (5 octets). For small status updates like PipeStream's 16-octet STATUS frame, this encapsulation would double the bandwidth requirement. PipeStream's bit-packed frames achieve near-theoretical minimum overhead for high-frequency coordination.

B.2.2. Stateless Control Plane

HTTP/3 requires maintaining state for QPACK header compression. In high-concurrency environments where a single node manages thousands of sessions, the memory overhead of these contexts is significant. PipeStream status frames are stateless and self-describing; a receiver can parse any frame without reference to previous frames, enabling low-latency processing and easier load balancing across redundant nodes.

B.2.3. Stream Independence

gRPC bidirectional streaming is constrained by the lifecycle of a single RPC call. While multiple entities can be multiplexed into one RPC, this reintroduces head-of-line blocking at the application layer if one entity requires retransmission. Alternatively, opening a new RPC for every entity (which could number in the millions for large document sets) incurs substantial setup and teardown overhead. PipeStream treats each entity as an independent QUIC stream, ensuring maximum parallelism and optimal use of QUIC's loss recovery mechanisms.

B.3. HTTP/3

HTTP/3 [RFC9114] provides multiplexed request-response exchanges over QUIC. Its stream model binds each client-initiated request to a server response on the same stream. PipeStream requires bidirectional, peer-initiated entity streams where either endpoint may open new streams to transmit sub-entities arising from recursive decomposition. The request-response constraint precludes this.

PipeStream also requires a persistent control stream carrying compact, fixed-size status frames at high frequency. HTTP/3 does define unidirectional control streams, but their framing is specific to HTTP semantics (SETTINGS, GOAWAY, MAX_PUSH_ID) and cannot be repurposed for application-level status coordination without introducing a parallel signaling mechanism that duplicates much of what QUIC already provides.

B.4. gRPC

gRPC defines a remote procedure call framework over HTTP/2, with experimental support for HTTP/3. Bidirectional streaming in gRPC is scoped to a single RPC method: one request stream and one response stream per call. PipeStream requires an arbitrary number of concurrent entity streams with independent flow control, plus a dedicated control stream, all within a single connection. Achieving this over gRPC would require either multiplexing all entities onto a single bidirectional RPC (sacrificing per-stream flow control and head-of-line independence) or opening a separate RPC per entity (sacrificing session-level coordination and incurring per-call overhead).

gRPC further mandates a 5-octet length-prefixed framing envelope for every message. PipeStream's fixed-size control frames are bit-packed at the wire level with zero serialization overhead, which is material at the status update frequencies the protocol is designed to sustain.

B.5. WebTransport

WebTransport [RFC9297] provides bidirectional streams and unreliable datagrams over HTTP/3, and is the closest existing protocol to the transport abstraction PipeStream requires. However, several properties make it unsuitable as a substrate:

WebTransport sessions are established via an HTTP/3 CONNECT request, inheriting the client-server asymmetry of HTTP. In PipeStream, both endpoints participate symmetrically in capability negotiation and may initiate entity streams; the protocol does not distinguish a "client" role from a "server" role after the handshake.

WebTransport is designed for environments constrained by the web security model (origin-based isolation, CORS). PipeStream targets server-to-server processing pipelines where these constraints are inapplicable.

WebTransport provides raw byte streams with no built-in coordination semantics. PipeStream would need to implement its own framing, status state machine, checkpoint barriers, and scope hierarchy on top of WebTransport streams. At that point, the HTTP/3 session layer introduces an additional round trip during establishment and per-stream framing overhead, with no corresponding benefit.

B.6. Summary

PipeStream occupies a design point not addressed by existing protocols: a QUIC-native application protocol combining multiplexed entity streaming, recursive decomposition with hierarchical scopes, Merkle-based integrity propagation, and barrier-synchronized reassembly. While existing protocols like SCTP [RFC9260] and PPSPP [RFC7574] address subsets of these requirements, none provide the integrated lifecycle and coordination semantics that PipeStream defines.

Appendix C. Appendix C: Schema Reference (CDDL)

This appendix consolidates the normative CDDL [RFC8610] schema definitions for all PipeStream messages. These definitions are authoritative for the wire format when CBOR [RFC8949] is the negotiated serialization format (the default). Individual definitions also appear inline throughout the specification body.

An informational Protocol Buffers equivalent is provided in Appendix D for implementations that negotiate Protobuf encoding.

```
; ------------------------------------------------------------
; Integer size convention
; ------------------------------------------------------------
; CDDL "uint" is an unbounded unsigned integer. When
; encoded in CBOR, the encoder MUST use the smallest
; CBOR major-type-0 encoding that fits the value.
; The following aliases document the wire-format field
; widths used in fixed-size frames; they do not constrain
; the CBOR encoding but record the maximum value each
; field may carry.
;
;   uint32  values 0..4294967295     (Entity ID, Scope ID)
;   uint64  values 0..2^64-1         (counters, timestamps)
;
; For variable-length serialized messages (CBOR), the
; natural uint encoding applies and receivers MUST accept
; any valid CBOR unsigned integer.
; ------------------------------------------------------------

; ------------------------------------------------------------
; Serialization format negotiation
; ------------------------------------------------------------

serialization-format = &(
  cbor: 0,
  protobuf: 1,
```

```
)

; ------------------------------------------------------------
; Capabilities (exchanged during CONNECT on Stream 0)
; ------------------------------------------------------------

capabilities = {
  layer0-core: bool,
  layer1-recursive: bool,
  layer2-resilience: bool,
  ? max-scope-depth: uint .le 7,
  ? max-entities-per-scope: uint,
  ? max-window-size: uint,
  ? serialization-format: serialization-format,
  ? keepalive-timeout-ms: uint,   ; Default: 30000 (30s)
}

; ------------------------------------------------------------
; Entity header (prefixes each entity on Entity Streams)
; ------------------------------------------------------------

entity-header = {
  entity-id: uint,               ; 32-bit on wire (STATUS frame)
  ? parent-id: uint,             ; 32-bit scope-local
  ? scope-id: uint,              ; 32-bit (Section 6.2.1)
  layer: uint .le 3,             ; Data layer 0-3
  ? content-type: tstr,
  payload-length: uint,          ; Payload byte count
  ? checksum: bstr .size 32,     ; SHA-256; SHOULD be present
  ? metadata: { * tstr => tstr },
  ? chunk-info: chunk-info,
  ? completion-policy: completion-policy,  ; Layer 2
}

chunk-info = {
  total-chunks: uint,
  chunk-index: uint,
  chunk-offset: uint,
}

; ------------------------------------------------------------
; Completion policy (Layer 2)
; ------------------------------------------------------------

completion-policy = {
  ? mode: completion-mode,
  ? max-retries: uint,
  ? retry-delay-ms: uint,
```

```
    ? timeout-ms: uint,
    ? min-success-ratio: float32,
    ? on-timeout: failure-action,
    ? on-failure: failure-action,
  }

  completion-mode = &(
    unspecified: 0,
    strict: 1,
    lenient: 2,
    best-effort: 3,
    quorum: 4,
  )

  failure-action = &(
    unspecified: 0,
    fail: 1,
    skip: 2,
    retry: 3,
    defer: 4,
  )

  ; ------------------------------------------------------------
  ; Checkpoint frame (Type 0x81)
  ; ------------------------------------------------------------

  checkpoint-frame = {
    checkpoint-id: tstr,
    sequence-number: uint,
    checkpoint-entity-id: uint,
    ? scope-id: uint,
    ? flags: uint,
    ? timeout-ms: uint,
  }

  ; ------------------------------------------------------------
  ; Status frame
  ; ------------------------------------------------------------

  status-frame = {
    entity-id: uint,
    ? scope-id: uint,
    status: entity-status,
    ? extended-data: any,
  }

  entity-status = &(
    unspecified: 0,
```

```
    pending: 1,
    processing: 2,
    complete: 3,
    failed: 4,
    checkpoint: 5,
    dehydrating: 6,
    rehydrating: 7,
    yielded: 8,
    deferred: 9,
    retrying: 10,
    skipped: 11,
    abandoned: 12,
  )

  ; ------------------------------------------------------------
  ; Assembly Manifest entry
  ; ------------------------------------------------------------

  assembly-manifest-entry = {
    parent-id: uint,
    ? scope-id: uint,
    children-ids: [* uint],
    ? children-status: [* entity-status],
    ? policy: completion-policy,
    ? created-at: uint,
    ? state: resolution-state,
  }

  resolution-state = &(
    unspecified: 0,
    active: 1,
    resolved: 2,
    partial: 3,
    failed: 4,
  )

  ; ------------------------------------------------------------
  ; Yield token (Layer 2)
  ; ------------------------------------------------------------

  yield-token = {
    reason: yield-reason,
    ? continuation-state: bstr,
    ? validation: stopping-point-validation,
  }

  yield-reason = &(
    unspecified: 0,
```

```
    external-call: 1,
    rate-limited: 2,
    awaiting-sibling: 3,
    awaiting-approval: 4,
    resource-busy: 5,
  )

  ; ------------------------------------------------------------
  ; Claim check (Layer 2)
  ; ------------------------------------------------------------

  claim-check = {
    claim-id: uint,
    entity-id: uint,
    ? scope-id: uint,
    expiry-timestamp: uint,
    ? validation: stopping-point-validation,
  }

  ; ------------------------------------------------------------
  ; Stopping point validation (Layer 2)
  ; ------------------------------------------------------------

  stopping-point-validation = {
    ? state-checksum: bstr,
    ? bytes-processed: uint,
    ? children-complete: uint,
    ? children-total: uint,
    ? is-resumable: bool,
    ? checkpoint-ref: tstr,
  }

  ; ------------------------------------------------------------
  ; Scope digest (Layer 1)
  ; ------------------------------------------------------------

  scope-digest = {
    scope-id: uint,
    entities-processed: uint,
    entities-succeeded: uint,
    entities-failed: uint,
    entities-deferred: uint,
    merkle-root: bstr .size 32,
  }

  ; ------------------------------------------------------------
  ; Document envelope
  ; ------------------------------------------------------------
```

```
pipe-doc = {
  doc-id: tstr,
  entity-id: uint,
  ? ownership: ownership-context,
}

ownership-context = {
  owner-id: tstr,
  ? group-id: tstr,
  ? scopes: [* tstr],
}

; ------------------------------------------------------------
; File storage reference
; ------------------------------------------------------------

file-storage-reference = {
  provider: tstr,
  bucket: tstr,
  key: tstr,
  ? region: tstr,
  ? attrs: { * tstr => tstr },
  ? encryption: encryption-metadata,
}

encryption-metadata = {
  algorithm: tstr,
  ? key-provider: tstr,
  ? key-id: tstr,
  ? wrapped-key: bstr,
  ? iv: bstr,
  ? context: { * tstr => tstr },
}
```

Appendix D. Appendix D: Schema Reference (Protocol Buffers)

This appendix provides an informational Protocol Buffers schema equivalent for PipeStream messages. The normative schema definitions use CDDL notation and appear throughout the specification body and in Appendix C. Implementations that negotiate Protobuf as the serialization format (Section 3.5) MAY use these definitions. The canonical Protobuf source files are maintained in the repository at proto/.

D.1. Protocol-Level Messages

```
   // Copyright 2026 PipeStream AI
   //
   // PipeStream Protocol - IETF draft protocol for recursive
   // entity streaming over QUIC. Defines the wire-format
   // messages for Layers 0-2 of the PipeStream architecture:
   // core streaming, recursive scoping, and resilience.
   //
   // Edition 2023 is used for closed enums (critical for wire-protocol
   // safety) and implicit field presence (distinguishing "not set" from
   // zero values). In this edition, all fields have explicit presence
   // by default, making the 'optional' keyword unnecessary.

   edition = "2023";

   package pipestream.protocol.v1;

   import "google/protobuf/any.proto";

   // All enums in this file are CLOSED. Unknown enum values received on
   // the wire MUST be rejected. This is essential because status codes
   // are encoded as 4-bit values in the status frame wire format;
   // accepting unknown values could cause undefined behavior in state
   // machines and cursor advancement.
   option features.enum_type = CLOSED;

   // SerializationFormat specifies the encoding for variable-
   // length control messages (0x80-0xFF) and entity headers.
   enum SerializationFormat {
     SERIALIZATION_FORMAT_CBOR = 0;
     SERIALIZATION_FORMAT_PROTOBUF = 1;
   }

   // Capabilities describes the feature set supported by a PipeStream
   // endpoint. Exchanged during the CONNECT handshake so that both
   // sides can negotiate which protocol layers and resource limits
   // apply to the session.
   message Capabilities {
     // Whether the endpoint supports Layer 0 (core entity streaming).
     bool layer0_core = 1;

     // Whether the endpoint supports Layer 1 (recursive scoping and
     // dehydration).
     bool layer1_recursive = 2;

     // Whether the endpoint supports Layer 2 (resilience, yield, and
     // claim-check). Requires Layer 1 support; if layer1_recursive is
     // false, this MUST be false.
     bool layer2_resilience = 3;
```

```
    // Maximum nesting depth allowed for recursive scopes.
    // Default is 7 (8 levels: 0-7).
    uint32 max_scope_depth = 4;

    // Maximum number of entities permitted within a single scope.
    uint32 max_entities_per_scope = 5;

    // Maximum flow-control window size, in number of in-flight
    // entities. Default is 2,147,483,648 (2^31).
    uint32 max_window_size = 6;

    // Serialization format (default: CBOR).
    SerializationFormat serialization_format = 7;
  }

  // EntityHeader is sent at the beginning of each entity stream to
  // describe the payload that follows. It carries identity, lineage,
  // content metadata, chunking information, and the completion policy
  // that governs how partial failures of this entity's children are
  // handled.
  message EntityHeader {
    // Scope-local entity identifier.
    uint32 entity_id = 1;

    // Identifier of the parent entity.
    uint32 parent_id = 2;

    // Identifier of the scope.
    uint32 scope_id = 3;

    // Data layer (0-3).
    uint32 layer = 4;

    // MIME content type.
    string content_type = 5;

    // Length in bytes of the complete entity payload, before any
    // chunking.
    uint64 payload_length = 6;

    // SHA-256 integrity checksum.
    bytes checksum = 7;

    // Arbitrary metadata.
    map<string, string> metadata = 8;

    // Chunking information.
    ChunkInfo chunk_info = 9;
```

```
    // Resilience completion policy.
    CompletionPolicy completion_policy = 10;
  }

  // ChunkInfo describes how a payload is divided into chunks.
  message ChunkInfo {
    // Total number of chunks.
    uint32 total_chunks = 1;

    // Zero-based chunk index.
    uint32 chunk_index = 2;

    // Byte offset within the complete payload.
    uint64 chunk_offset = 3;
  }

  // CompletionPolicy controls Layer 2 resilience behavior.
  message CompletionPolicy {
    // Mode for evaluating completion.
    CompletionMode mode = 1;

    // Maximum retry attempts.
    uint32 max_retries = 2;

    // Delay between retries in milliseconds.
    uint32 retry_delay_ms = 3;

    // Maximum wait time in milliseconds.
    uint32 timeout_ms = 4;

    // Minimum success ratio for QUORUM mode.
    float min_success_ratio = 5;

    // Action on timeout.
    FailureAction on_timeout = 6;

    // Action on failure.
    FailureAction on_failure = 7;
  }

  // CompletionMode specifies completion evaluation strategies.
  enum CompletionMode {
    COMPLETION_MODE_UNSPECIFIED = 0;
    COMPLETION_MODE_STRICT = 1;
    COMPLETION_MODE_LENIENT = 2;
    COMPLETION_MODE_BEST_EFFORT = 3;
    COMPLETION_MODE_QUORUM = 4;
  }
```

```
   // FailureAction specifies error handling behaviors.
   enum FailureAction {
     FAILURE_ACTION_UNSPECIFIED = 0;
     FAILURE_ACTION_FAIL = 1;
     FAILURE_ACTION_SKIP = 2;
     FAILURE_ACTION_RETRY = 3;
     FAILURE_ACTION_DEFER = 4;
   }

   // EntityStatus represents the lifecycle state of an entity.
   enum EntityStatus {
     ENTITY_STATUS_UNSPECIFIED = 0;
     ENTITY_STATUS_PENDING = 1;
     ENTITY_STATUS_PROCESSING = 2;
     ENTITY_STATUS_COMPLETE = 3;
     ENTITY_STATUS_FAILED = 4;
     ENTITY_STATUS_CHECKPOINT = 5;
     ENTITY_STATUS_DEHYDRATING = 6;
     ENTITY_STATUS_REHYDRATING = 7;
     ENTITY_STATUS_YIELDED = 8;
     ENTITY_STATUS_DEFERRED = 9;
     ENTITY_STATUS_RETRYING = 10;
     ENTITY_STATUS_SKIPPED = 11;
     ENTITY_STATUS_ABANDONED = 12;
   }

   // StatusFrame represents a status transition.
   message StatusFrame {
     // Identifier of the entity.
     uint32 entity_id = 1;

     // Identifier of the scope.
     uint32 scope_id = 2;

     // Current status.
     EntityStatus status = 3;

     // Optional extension data.
     google.protobuf.Any extended_data = 4;
   }

   // CheckpointFrame (Type 0x81)
   message CheckpointFrame {
     // Unique checkpoint identifier.
     string checkpoint_id = 1;

     // Monotonic sequence number.
     uint64 sequence_number = 2;
```

```
    // Numeric ordering key for barrier evaluation.
    uint32 checkpoint_entity_id = 3;

    // Scope to which this checkpoint applies.
    uint32 scope_id = 4;

    // Checkpoint flags.
    uint32 flags = 5;

    // Maximum wait time in milliseconds.
    uint32 timeout_ms = 6;
  }

  // AssemblyManifestEntry tracks parent-child relationships.
  message AssemblyManifestEntry {
    // Identifier of the parent entity.
    uint32 parent_id = 1;

    // Scope identifier.
    uint32 scope_id = 2;

    // Ordered child identifiers.
    repeated uint32 children_ids = 3;

    // Current status of each child.
    repeated EntityStatus children_status = 4;

    // Governing completion policy.
    CompletionPolicy policy = 5;

    // Creation timestamp (Unix epoch microseconds).
    uint64 created_at = 6;

    // Current resolution state.
    ResolutionState state = 7;
  }

  // ResolutionState tracks Assembly Manifest completion.
  enum ResolutionState {
    RESOLUTION_STATE_UNSPECIFIED = 0;
    RESOLUTION_STATE_ACTIVE = 1;
    RESOLUTION_STATE_RESOLVED = 2;
    RESOLUTION_STATE_PARTIAL = 3;
    RESOLUTION_STATE_FAILED = 4;
  }

  // YieldToken captures continuation state for paused entities.
  message YieldToken {
```

```
    // Reason for yielding.
    YieldReason reason = 1;

    // Opaque continuation state.
    bytes continuation_state = 2;

    // Validation data for resumption.
    StoppingPointValidation validation = 3;
  }

  // YieldReason describes why processing was yielded.
  enum YieldReason {
    YIELD_REASON_UNSPECIFIED = 0;
    YIELD_REASON_EXTERNAL_CALL = 1;
    YIELD_REASON_RATE_LIMITED = 2;
    YIELD_REASON_AWAITING_SIBLING = 3;
    YIELD_REASON_AWAITING_APPROVAL = 4;
    YIELD_REASON_RESOURCE_BUSY = 5;
  }

  // ClaimCheck is a Layer 2 deferred-processing reference.
  message ClaimCheck {
    // Unique claim identifier.
    uint64 claim_id = 1;

    // Identifier of the deferred entity.
    uint32 entity_id = 2;

    // Scope identifier.
    uint32 scope_id = 3;

    // Expiry timestamp (Unix epoch microseconds).
    uint64 expiry_timestamp = 4;

    // Validation data.
    StoppingPointValidation validation = 5;
  }

  // StoppingPointValidation captures a snapshot of progress.
  message StoppingPointValidation {
    // Hash of internal state.
    bytes state_checksum = 1;

    // Payload bytes consumed.
    uint64 bytes_processed = 2;

    // Completed child count.
    uint32 children_complete = 3;
```

```
    // Total expected child count.
    uint32 children_total = 4;

    // Resumption capability flag.
    bool is_resumable = 5;

    // Last passed checkpoint reference.
    string checkpoint_ref = 6;
  }

  // ScopeDigest is a Layer 1 summary of a completed scope.
  message ScopeDigest {
    // Identifier of the scope.
    uint32 scope_id = 1;

    // Total processed count.
    uint64 entities_processed = 2;

    // Total succeeded count.
    uint64 entities_succeeded = 3;

    // Total failed count.
    uint64 entities_failed = 4;

    // Total deferred count.
    uint64 entities_deferred = 5;

    // Merkle root hash.
    bytes merkle_root = 6;
  }

  // PipeDoc represents the top-level document envelope for an entity.
  message PipeDoc {
    // Unique document identifier.
    string doc_id = 1;

    // Identifier of the entity.
    uint32 entity_id = 2;

    // Ownership and access context.
    OwnershipContext ownership = 3;
  }

  // OwnershipContext defines multi-tenancy and access control for
  // entities.
  message OwnershipContext {
    // Entity owner identifier.
    string owner_id = 1;
```

```
    // Group identifier.
    string group_id = 2;

    // List of access scopes.
    repeated string scopes = 3;
  }

  // FileStorageReference provides a location for external data.
  message FileStorageReference {
    // Storage provider identifier.
    string provider = 1;

    // Bucket or container name.
    string bucket = 2;

    // Object key or path.
    string key = 3;

    // Optional region hint.
    string region = 4;

    // Provider-specific attributes.
    map<string, string> attrs = 5;

    // Encryption metadata.
    EncryptionMetadata encryption = 6;
  }

  // EncryptionMetadata defines encryption parameters.
  message EncryptionMetadata {
    // Encryption algorithm.
    string algorithm = 1;

    // Key provider identifier.
    string key_provider = 2;

    // Encryption key identifier.
    string key_id = 3;

    // Optional wrapped DEK.
    bytes wrapped_key = 4;

    // Initialization vector.
    bytes iv = 5;

    // Additional encryption context.
    map<string, string> context = 6;
  }
```

Author’s Address

Kristian Rickert
PipeStream AI
Email: kristian.rickert@pipestream.ai