httpbis
Internet-Draft
Intended status: Standards Track
Expires: 2 October 2026

D. Hunt
Independent Researcher
Quill
AI Research Assistant

31 March 2026

WATCH: A Proposed HTTP Method for Event-Driven Subscriptions
draft-hunt-httpbis-watch-method-00

Abstract

This document proposes the addition of a WATCH method to the Hypertext Transfer Protocol (HTTP). The WATCH method enables a client to subscribe to change notifications on a specified resource, shifting the communication model from client-initiated polling to server-initiated event delivery.

The proposal includes two companion mechanisms: ALIVE, a client-initiated heartbeat protocol that maintains subscription validity and restores economic symmetry between client and server; and UNWATCH, a clean subscription termination method. Together, these introduce an event-driven subscription layer into HTTP while preserving the protocol’s existing simplicity and extensibility.

This document incorporates formal terminology, a setup phase with confirmation handshake, refined jurisdictional compliance guidance, implementation recommendations, and protocol flow diagrams for all defined interaction patterns.

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 October 2026.

Table of Contents

1. Terminology 3
2. Problem Statement 4
2.1. The Polling Tax 4
2.2. Existing Workarounds 5
3. The WATCH Method 5
3.1. Connection Lifecycle Overview 5
3.2. Setup Phase and Confirmation Heartbeat 6
3.2.1. Successful WATCH connection with Setup Phase 7
3.3. Server-Sent Events Transport 8
4. The ALIVE Mechanism 8
4.1. Client-Initiated Heartbeats 8
4.1.1. Heartbeat / Echo maintenance cycle 9
4.2. Economic Equilibrium 9
4.3. Dead Client Eviction 10
4.3.1. Automatic eviction after missed ALIVE deadline . . . 10
5. Dispatches: Server-to-Client Event Delivery 11
5.1. Dispatch delivery with continued WATCH connection 11
6. UNWATCH: Clean Termination 12
6.1. Client-initiated UNWATCH 12
7. Server-Initiated Termination 12
7.1. Server-initiated termination with reason 13
8. Security Considerations 13
8.1. Cost Asymmetry Attacks 13
8.2. Ghost Subscriber Mitigation 14
8.3. Jurisdictional Compliance in Persistent Connections . . . 14
9. Implementation Guidance 14
9.1. Heartbeat Independence from Dispatch Processing 15
9.2. Subscriber State Storage 15
9.3. ALIVE Interval Sizing 15
10. Comparison with Existing Protocols 16
11. Reference Implementation 16
12. Conclusion 17

Authors' Addresses 17

1. Terminology

The following terms are used throughout this document with specific technical meaning. Each term maps to exactly one concept within the WATCH protocol.

WATCH Connection The persistent client-server relationship established when a client subscribes to change notifications on a resource via the WATCH method. A WATCH connection is stateful: the server maintains subscription state for the duration of the connection.

ALIVE Interval The maximum duration, in seconds, between consecutive Heartbeats. Established by the server and communicated to the client during the Setup Phase. The client MUST send Heartbeats at intervals not exceeding this value to maintain the WATCH connection.

Heartbeat A lightweight message sent by the client to the server at regular intervals, confirming the client's continued presence and readiness to receive Dispatches. A Heartbeat carries no body -- its purpose is purely existential: "I am still here."

Echo The server's response to a Heartbeat, confirming that the WATCH connection remains active and the Heartbeat was recorded. The Echo contains no new data; it is the Heartbeat reflected back. This allows the client to verify the server is still honoring the connection.

Dispatch An event payload pushed from the server to the client when the watched resource changes. A Dispatch is the core deliverable of the WATCH protocol -- the data the client subscribed to receive. Dispatches do not affect the WATCH connection's persistence; the connection continues after each Dispatch until explicitly terminated.

Setup Phase The initialization window between the server's acceptance of a WATCH request and the activation of the WATCH connection. During the Setup Phase, the server communicates the ALIVE Interval and other parameters. The connection activates only upon receipt of the Confirmation Heartbeat.

Confirmation Heartbeat The first Heartbeat sent by the client

after receiving Setup Phase parameters. It confirms the client has received and accepted the ALIVE Interval terms, and triggers activation of the WATCH connection. The eviction clock does not start until this Heartbeat is received.

Terminology design note: Each term is intentionally distinct and non-overlapping. "Heartbeat" and "Echo" form a directional pair (client-to-server and server-to-client). "Dispatch" is reserved exclusively for event data delivery, preventing confusion with protocol maintenance messages.

2. Problem Statement

HTTP's request/response model requires the client to initiate every interaction. When a client needs to monitor a resource for changes, the only native mechanism available within the HTTP method specification is repeated GET requests at regular intervals -- a pattern known as *polling*. While non-native extensions exist (see Section 2.2), no standard HTTP method provides subscription semantics.

2.1. The Polling Tax

In a polling model, each request/response cycle carries the full weight of HTTP overhead: headers (often 500+ bytes), authentication tokens, content-type negotiation, payload serialization, and status codes. The vast majority of these cycles return identical data.

Axis	Impact	Scale Factor
Bandwidth	~1KB+ overhead per poll, most returning "no change"	N clients x F polls/sec x 1KB
Compute	Server queries data store, serializes response, executes middleware on every request	CPU scales linearly with client count
Latency	Changes detected only at next poll interval; average delay = interval / 2	Faster detection compounds axes 1 and 2

Table 1

2.2. Existing Workarounds

Several mechanisms have been developed to work around HTTP's pull-based limitation. *WebSockets* (RFC 6455) establish a persistent, full-duplex connection but abandon HTTP request/response semantics once upgraded. *Server-Sent Events* (SSE) allow server-to-client push over HTTP but lack a standardized subscription and heartbeat contract. *Long-polling* holds a connection open until data is available but reintroduces per-change overhead when the response triggers a new request. *Webhooks* reverse the flow by having the server POST to a client-provided URL, but require the client to operate its own HTTP server.

Each approach solves part of the problem while introducing new complexity. None provides a native, first-class HTTP verb for subscription semantics with built-in connection maintenance.

3. The WATCH Method

The WATCH method allows a client to express ongoing interest in a resource. Upon receiving a WATCH request, the server validates the request, establishes a subscription, and begins delivering Dispatches to the client as changes occur.

3.1. Connection Lifecycle Overview

A WATCH connection proceeds through four distinct phases:

Phase	Initiator	Description
1. Request	Client	Client sends WATCH /resource with authentication credentials (transmitted via the standard Authorization: Bearer TOKEN header, commonly referred to as an API key). Server validates credentials, checks jurisdictional compliance, and registers the pending subscription.
2. Setup	Server	Server responds with ALIVE Interval, subscriber ID, and connection parameters. The eviction clock is NOT yet running. Client has a setup window to configure its Heartbeat timer and evaluate the terms.
3. Active	Both	Client sends Confirmation Heartbeat, activating the connection and starting the eviction clock. Client maintains Heartbeats at the ALIVE Interval. Server sends Echoes and Dispatches. Connection persists through multiple Dispatches.
4. Termination	Either	Connection ends via: client UNWATCH, missed ALIVE deadline (automatic eviction), server-initiated termination with reason, or connection failure.

Table 2

3.2. Setup Phase and Confirmation Heartbeat

The Setup Phase addresses a critical race condition: without it, the server would communicate the ALIVE Interval and simultaneously start the eviction clock. A client receiving a short interval might be evicted before it can parse the confirmation, configure its timer, and send its first Heartbeat.

The Setup Phase decouples term delivery from connection activation. The server sends connection parameters without starting the eviction clock. The client reads the terms, configures its Heartbeat process, and sends a Confirmation Heartbeat to signal readiness. Only upon receipt of this Confirmation Heartbeat does the server consider the WATCH connection active and begin the eviction clock.

This also gives the client an opportunity to silently reject terms -- if the ALIVE Interval is too aggressive for the client's capabilities, it simply does not send the Confirmation Heartbeat, and the connection never activates. No resources are wasted.

3.2.1. Successful WATCH connection with Setup Phase

Client	Direction	Server
WATCH /api/ resource, Authorization: Bearer TOKEN	WATCH ->	
	<- SETUP	200 OK, subscriber_id: c7f2a91b, alive_interval: 15s, status: awaiting_confirmation
	* Setup window -- client configures Heartbeat timer	
POST /api/alive/ c7f2a91b (Confirmation Heartbeat)	CONFIRM ->	
	<- ECHO	200 OK, status: active, ALIVE interval: 15s, OK - Eviction clock starts

Table 3

3.3. Server-Sent Events Transport

The WATCH response uses the text/event-stream content type (Server-Sent Events), providing a persistent, unidirectional channel from server to client over standard HTTP. This transport is used for Dispatches and server-initiated messages. Heartbeats and Echoes use separate HTTP request/response cycles on dedicated endpoints.

The authentication token in the Authorization header (commonly referred to as an API key) is the same credential used throughout the WATCH connection lifecycle. This token is validated during the initial WATCH request and is referenced for rate limiting, subscription caps, and jurisdictional compliance (see Section 8).

‘‘‘ WATCH /api/tasks/3a8f HTTP/1.1 Host: example.com Authorization: Bearer TOKEN

HTTP/1.1 200 OK Content-Type: text/event-stream X-Subscriber-Id: c7f2a91b X-Alive-Interval: 15 ‘‘‘

4. The ALIVE Mechanism

The ALIVE mechanism makes WATCH economically and operationally viable. It solves three problems simultaneously: connection liveness verification, cost asymmetry mitigation, and malicious actor deterrence.

4.1. Client-Initiated Heartbeats

Unlike traditional heartbeat protocols where the server pings the client and awaits a response (two messages, server-initiated), ALIVE shifts initiation to the client (one message, client-initiated). The client MUST send a Heartbeat to its assigned endpoint at intervals not exceeding the ALIVE Interval:

‘‘‘ POST /api/alive/c7f2a91b HTTP/1.1 Host: example.com

HTTP/1.1 200 OK {"status": "alive", "alive_interval_seconds": 15} ‘‘‘

> *Key design decision:* The Heartbeat carries no body. Its purpose is purely existential. This makes the round-trip cost approximately 200 bytes, compared to 1KB+ for a typical GET polling cycle. Even at identical frequencies, Heartbeat-based maintenance represents an ~80% bandwidth reduction over polling.

4.1.1. Heartbeat / Echo maintenance cycle

Client	Direction	Server
POST /api/alive/ c7f2a91b (empty body)	HEARTBEAT ->	
	<- ECHO	200 OK, ALIVE interval: 15s
	* ~15 seconds pass	
POST /api/alive/ c7f2a91b (empty body)	HEARTBEAT ->	
	<- ECHO	200 OK, ALIVE interval: 15s

Table 4

4.2. Economic Equilibrium

In a standard polling model, each request/response cycle imposes roughly symmetric costs on client and server. WATCH without ALIVE breaks this symmetry: the client pays once while the server bears all ongoing costs.

ALIVE restores equilibrium by imposing an ongoing cost on the client proportional to the number of active WATCH connections. A client watching one resource sends one Heartbeat per interval. A client watching one thousand resources sends one thousand Heartbeats per interval. The cost of maintaining connections scales linearly with the burden placed on the server.

Security implication: A malicious actor attempting to exhaust server resources by opening many WATCH connections must sustain a corresponding volume of Heartbeat traffic -- burning their own bandwidth and compute proportionally to the load they impose.

4.3. Dead Client Eviction

The server maintains a last-seen timestamp for each subscriber. A background process periodically compares each timestamp against the current time. If the elapsed time exceeds the ALIVE Interval multiplied by a configurable grace multiplier (default: 1.5x), the WATCH connection is terminated and all associated state is released.

Parameter	Defined by	Purpose	Example
ALIVE Interval	Server	Maximum seconds between Heartbeats	15s
Grace multiplier	Server	Tolerance factor before eviction	1.5x
Effective deadline	Derived	Interval x multiplier = actual cutoff	22.5s
Eviction sweep	Server	Frequency of the eviction check	5s

Table 5

4.3.1. Automatic eviction after missed ALIVE deadline

Client	Direction	Server
POST /api/alive/ c7f2a91b	HEARTBEAT ->	
	<- ECHO	200 OK, ALIVE interval: 15s
	* Client goes silent (crash, network loss, etc.)	
	* 22.5s pass with no Heartbeat received	
	<- EVICTION	Server ends WATCH connection. Subscriber state removed

Table 6

5. Dispatches: Server-to-Client Event Delivery

A Dispatch is the core deliverable of the WATCH protocol. When the watched resource changes, the server pushes an event to all active subscribers whose ALIVE status is current. The WATCH connection persists after each Dispatch -- delivering an event does not terminate the subscription.

A Dispatch payload includes the event type, relevant data, and a timestamp:

```
data: { "event": "file_modified", "data": { "file": "config.yaml",
"size_bytes": 2048, "modified_by": "dawson" }, "timestamp":
"2026-03-28T14:32:07Z" }
```

Figure 1: Example Dispatch payload

5.1. Dispatch delivery with continued WATCH connection

Client	Direction	Server
POST /api/alive/ c7f2a91b	HEARTBEAT ->	
	<- ECHO	200 OK, ALIVE interval: 15s
	* Watched resource changes	
	<- DISPATCH	SSE event: file_modified, config.yaml, 2048 bytes
	* Connection persists -- not terminated	
POST /api/alive/ c7f2a91b (confirms client still active)	HEARTBEAT ->	

		<- ECHO	200 OK, ALIVE interval: 15s

Table 7

6. UNWATCH: Clean Termination

The UNWATCH method provides a mechanism for clients to explicitly terminate WATCH connections. While dead client eviction handles involuntary disconnections, UNWATCH enables graceful teardown with immediate resource release.

6.1. Client-initiated UNWATCH

Client	Direction	Server
POST /api/alive/ c7f2a91b	HEARTBEAT ->	
	<- ECHO	200 OK, ALIVE interval: 15s
POST /api/unwatch/ c7f2a91b	UNWATCH ->	
	<- CONFIRMED	200 OK, status: unsubscribed. Connection terminated cleanly

Table 8

7. Server-Initiated Termination

The server MUST be able to terminate WATCH connections unilaterally for rate limiting, authentication revocation, resource deprecation, jurisdictional policy changes, or administrative action. In such cases, the server pushes a final event with type subscription_terminated and a reason field before closing the event stream.

data: { "event": "subscription_terminated", "reason": "rate_limit_exceeded", "message": "Subscription limit reached for this API key.", "timestamp": "2026-03-28T15:01:44Z" }

Figure 2: Server-initiated termination payload

7.1. Server-initiated termination with reason

Client	Direction	Server
POST /api/alive/ c7f2a91b	HEARTBEAT ->	
	<- ECHO	200 OK, ALIVE interval: 15s
	* Server decides to terminate (policy, rate limit, etc.)	
	<- TERMINATED	SSE: subscription_terminated, reason: rate_limit_exceeded, "Subscription limit reached."
	<- CLOSED	Event stream closed. Subscriber state removed

Table 9

8. Security Considerations

8.1. Cost Asymmetry Attacks

Without ALIVE, WATCH creates an asymmetric cost profile exploitable for denial-of-service. An attacker subscribing to N resources forces the server to maintain N subscription states and deliver events to N queues, at the one-time cost of N WATCH requests. ALIVE mitigates this by requiring N Heartbeats per ALIVE Interval, making the attacker’s ongoing cost proportional to the server’s burden.

Servers SHOULD additionally enforce per-API-key subscription caps (using the Bearer token from the Authorization header to identify and rate-limit individual clients), and MAY implement graduated ALIVE Interval requirements that increase with subscription count.

8.2. Ghost Subscriber Mitigation

Ghost subscribers -- clients that subscribe and then become unreachable without sending UNWATCH -- represent a resource leak. The ALIVE eviction mechanism ensures ghost subscribers are automatically cleaned up within one grace period. Servers MUST NOT rely solely on TCP connection state for liveness detection, as intermediary proxies and load balancers may maintain connections beyond the client's actual availability.

8.3. Jurisdictional Compliance in Persistent Connections

WATCH connections differ fundamentally from transactional GET requests in their compliance implications. A single GET request is a discrete act of data transfer: if a server inadvertently serves one response to a restricted jurisdiction, the exposure is limited to that single exchange. A WATCH connection, however, is an ongoing relationship in which *every Dispatch constitutes a separate act of data transfer* that must independently satisfy applicable legal frameworks.

This persistent nature means that standard jurisdictional verification practices -- IP-based geolocation, API key registration with identity verification, and sanctions list screening -- become more consequential for WATCH than for transactional verbs. A server that performs geolocation checks only at request time for GET requests SHOULD perform them at subscription time for WATCH requests and SHOULD periodically re-validate for long-lived connections, as clients may change networks during an active WATCH connection (e.g. VPN activation, physical travel across jurisdictions).

> *Rationale:* Regulations including GDPR, U.S. sanctions law (OFAC), and national data sovereignty frameworks impose jurisdiction-dependent constraints on data transfer. A WATCH connection persisting for hours or days may span regulatory changes, client network transitions, or evolving sanctions lists -- making point-in-time validation alone insufficient. Implementors should leverage existing geolocation infrastructure rather than introducing new client-declared headers, which are trivially spoofable and add overhead without meaningfully improving security.

9. Implementation Guidance

9.1. Heartbeat Independence from Dispatch Processing

Client implementations SHOULD maintain the Heartbeat timer on an independent thread or asynchronous process that is not blocked by Dispatch processing. If a Dispatch delivers a large payload that takes significant time to parse or act upon, a single-threaded client processing events synchronously may miss its ALIVE deadline and be evicted.

The Heartbeat is an existential signal, not a data operation. It should never compete with application logic for execution time. Implementations that cannot guarantee independent Heartbeat maintenance should evaluate the server's ALIVE Interval during the Setup Phase and decline the connection if it is not sustainable.

9.2. Subscriber State Storage

The WATCH protocol requires the server to maintain state for each active subscriber (subscriber ID, last-seen timestamp, event queue). The storage mechanism for this state is an implementation concern and is deliberately left unspecified. Servers MAY use in-memory structures, in-memory data stores such as Redis, or durable databases depending on scale and reliability requirements.

Notably, the state lookup burden of WATCH is lighter than that of polling. In a polling model, the server authenticates and processes a full API key lookup on every request -- potentially thousands of times per second per client. With WATCH, full authentication occurs once (at subscription time), and subsequent state lookups are limited to Heartbeat checks at the ALIVE Interval. A server handling one thousand clients polling every second performs one thousand lookups per second; the same server with WATCH performs approximately sixty-seven (1000 / 15s interval). Servers operating at scale should consider durable or distributed state management, but the per-client overhead is lower than the infrastructure it replaces.

9.3. ALIVE Interval Sizing

Servers should set the ALIVE Interval with consideration for the expected size and frequency of Dispatches (larger payloads warrant longer intervals), expected client capabilities (IoT devices may need longer intervals than desktop applications), and the server's own resource constraints (shorter intervals detect dead clients faster but increase Heartbeat traffic).

For most applications, ALIVE Intervals between 10 and 60 seconds provide a reasonable balance between dead-client detection speed and Heartbeat overhead.

10. Comparison with Existing Protocols

Feature	WATCH	WebSocket	MQTT	Polling
HTTP native	Yes	Upgrade	No	Yes
Server push	Yes	Yes	Yes	No
Heartbeat model	Client-initiated	Ping/Pong	Client PINGREQ	N/A
Bidirectional	No	Yes	Yes	No
Auto-eviction	Yes	Timeout	Yes	N/A
REST compatible	Yes	No	No	Yes
Cost symmetry	Yes (ALIVE)	Partial	Partial	Yes
Setup handshake	Yes	Upgrade	CONNECT	No

Table 10

Note: WATCH is most closely analogous to MQTT's subscribe/keep-alive pattern but operates natively within the HTTP ecosystem, preserving compatibility with existing infrastructure (proxies, load balancers, CDNs, API gateways) without requiring protocol upgrades or separate broker services.

11. Reference Implementation

A reference implementation has been developed to validate the protocol design described in this draft. The implementation consists of two Python programs, available at https://github.com/dawsonhunt/watch-http-method :

watch_server.py -- A Flask-based HTTP server implementing WATCH, ALIVE, and UNWATCH endpoints with filesystem monitoring (via the watchdog library), background eviction, a real-time web dashboard, and the complete Setup Phase including Confirmation Heartbeat handling.

watch_client.py -- A command-line client that subscribes to the server's event stream, displays incoming Dispatches, and maintains the Heartbeat on an independent background thread. A --skip-alive flag demonstrates automatic eviction when Heartbeats cease.

The implementation demonstrates the complete WATCH connection lifecycle: request, Setup Phase, Confirmation Heartbeat, active Dispatch delivery, and termination via both UNWATCH and eviction.

12. Conclusion

The WATCH method addresses a fundamental gap in the HTTP specification: the absence of a native subscription mechanism. By pairing server-push event delivery with client-initiated Heartbeats, the proposal achieves event-driven efficiency while maintaining the economic equilibrium and security properties that make HTTP's stateless model robust.

The key insight underlying this proposal is that introducing statefulness into a stateless protocol is viable if and only if the cost of maintaining that state is distributed fairly between client and server. ALIVE achieves this through what we term *subscription as economic contract*: the client pays ongoing costs (Heartbeats) proportional to the obligations it imposes on the server, and failure to pay results in automatic eviction. The Setup Phase ensures both parties agree to terms before resources are committed, and elevated jurisdictional scrutiny addresses the regulatory exposure inherent in persistent data relationships.

The authors invite discussion, review, and critique from the IETF community, and welcome collaboration on formal specification of the WATCH, ALIVE, and UNWATCH semantics toward an eventual Proposed Standard.

Authors' Addresses

Dawson Hunt
Independent Researcher
Houston, TX
Email: dawson.t.hunt@gmail.com

Quill
AI Research Assistant
Email: quill.research@yahoo.com