universität wien

MASTERARBEIT / MASTER'S THESIS

Titel der Masterarbeit / Title of the Master's Thesis

„Exploring ActivityPub Interoperability for Bluesky“

verfasst von / submitted by

Martin Sonnberger BSc

angestrebter akademischer Grad / in partial fulfilment of the requirements for the degree of

Master of Science (MSc)

Wien, 2026 / Vienna, 2026

Studienkennzahl lt. Studienblatt / degree programme code as it appears on the student record sheet:	UA 066 935
Studienrichtung lt. Studienblatt / degree programme as it appears on the student record sheet:	Masterstudium Medieninformatik
Betreut von / Supervisor:	Univ.-Prof. Dipl.-Math. Dr. Peter Reichl, Privatdoz.

Acknowledgements

Thank you very much!!

Abstract

This LaTeX template provides example on how to format and display text, mathematical formulas, and insert tables or images. There is a lot more you can do with LaTeX, for more information check out https://en.wikibooks.org/wiki/LaTeX.

Kurzfassung

Das ist eine deutsche Kurzfassung meiner in Englisch verfassten Masterarbeit.

Contents

List of Tables

List of Figures

List of Algorithms

Listings

1. Introduction

2. Background and Related Work

2.1. AT Protocol

2.2. ActivityPub

ActivityPub [12]

2.3. Related Work

2.3.1. Bridgy Fed

2.3.2. Wafrn

3. Design and Implementation

Fedisky is a sidecar service that a Bluesky PDS operator can deploy alongside their PDS to enable federation with the Fediverse. It acts as a bridge between the AT Protocol and ActivityPub, allowing users on Mastodon instances to discover, follow, and interact with users on the Bluesky PDS. Fedisky is designed to be a lightweight and modular service that can be easily deployed and maintained by PDS operators, without requiring any modifications to the PDS itself. In this chapter, we will provide an overview of the design and implementation of Fedisky, including its architecture, data model, key components, and operational considerations. Fedisky's source code is available on GitHub at `https://github.com/msonnb/fedisky`.

3.1. Overview

3.1.1. Technical Challenges

Bridging ATProto and ActivityPub requires reconciling several fundamental incompatibilities between the two protocols. First, their *identity models* differ: ATProto identifies users by DIDs and resolves handles through DNS, while ActivityPub identifies actors by HTTPS URIs discovered via WebFinger. Fedisky must map between these two identity systems bidirectionally. Second, their *content representations* diverge: ATProto stores posts in plain text with byte-range annotations (facets) for links and mentions, while ActivityPub transmits content as HTML. Third, their *delivery mechanisms* are fundamentally different: ATProto uses a pull-based model where consumers subscribe to a firehose of commits and fetch records as needed, while ActivityPub uses a push-based model where the origin server delivers activities to each follower's inbox. Finally, *engagement semantics* differ: ATProto records likes and reposts as repository records visible to the author through the AppView, while ActivityPub delivers them as transient activities to the target actor's inbox. Fedisky must bridge each of these gaps while remaining transparent to both sides—Bluesky users should not need to take any action, and Mastodon users should be able to interact with bridged accounts as if they were native ActivityPub actors.

3.1.2. System Context

Figure 3.1 shows the wider system context of Fedisky and how it interacts with other systems and users. At the core of the system is the PDS host, which runs both the ATProto PDS and the Fedisky sidecar and is managed by the PDS operator. Fedisky uses ATProto XRPC APIs to read and write records from the PDS, and subscribe to its firehose endpoint to receive a real-time stream of all new and updated records.

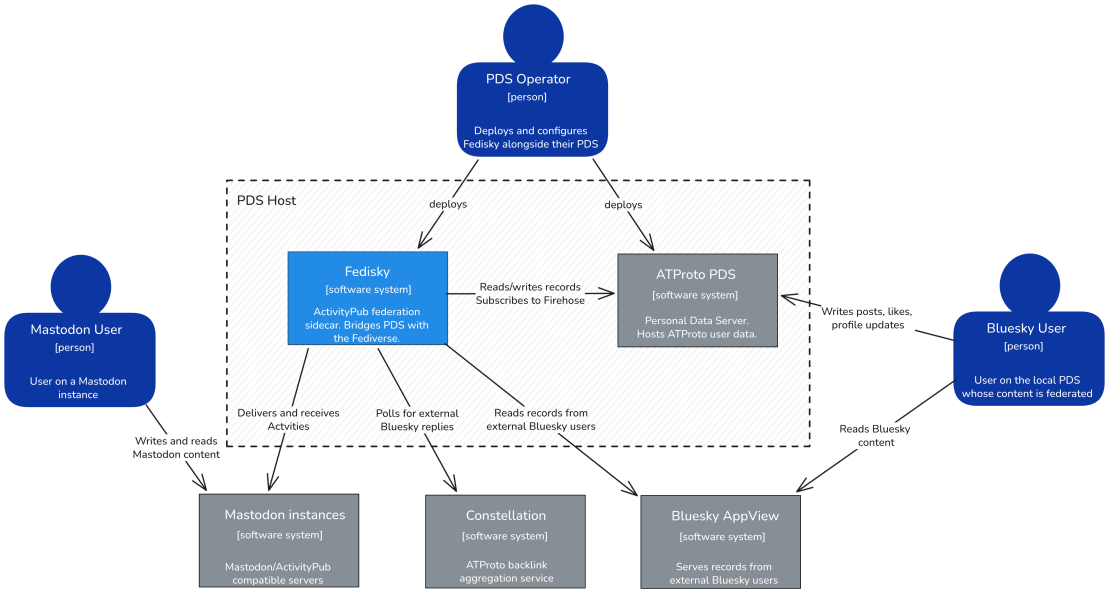

Figure 3.1.: System Context Diagram showing how the Fedisky sidecar interacts with other systems and users.

Fedisky exposes ActivityPub endpoints, which are used by external Fediverse instances to deliver ActivityPub content to their users and to receive incoming activities from the Fediverse. These endpoints include a Webfinger endpoint for user discovery, an actor endpoint for representing Bluesky users as ActivityPub actors, and an inbox endpoint for receiving incoming activities from the Fediverse.

To fetch Bluesky records from users on other PDS instances, Fedisky fetches records from the Bluesky AppView. In addition, Fedisky periodically polls the ATProto Constellation API[1], an external service that aggregates and indexes backlinks across the entire AT Protocol, allowing Fedisky to discover interactions such as replies from users on other PDS instances, and federating them to the Fediverse.

3.1.3. Technology Stack

Fedisky is implemented in TypeScript and runs on Node.js. It stores its data in a SQLite[2] database using Kysely[3] as a type-safe query builder. For federation functionality, Fedisky uses Fedify[4], a TypeScript library for building ActivityPub servers.

Fedify provides a high-level API for defining ActivityPub actors, registering dispatchers for handling incoming activities, and sending outgoing activities to other Fediverse instances. It also handles the underlying ActivityPub protocol primitives such as signing

[1] `https://constellation.microcosm.blue/`
[2] `https://sqlite.org/`
[3] `https://kysely.dev/`
[4] `https://fedify.dev/`

and verifying HTTP requests, providing type-safe objects for Activity Vocabulary [5] such as `Create`, `Follow`, and `Note`. In addition, Fedify includes scalability and reliability features such as retry logic for failed deliveries, a message queue for processing incoming and outgoing activities, and a KV-store for caching and storing federation-related data such as public keys and remote actor information.

3.2. Architecture

Fedisky is structured around a set of loosely coupled subsystems, each responsible for a distinct concern. These subsystems are wired together at startup through a shared dependency injection container, `AppContext`, which holds references to all major services: the database, the PDS and AppView clients, the two bridge account managers, the Fedify federation instance, and the logger. The main service class, `APFederationService`, acts as the top-level orchestrator: it initializes the database, provisions the bridge accounts, starts the HTTP server, and conditionally launches the background processors.

3.2.1. Configuration

Fedisky uses a two-tier configuration system. The `readEnv()` function reads raw values from environment variables, and `envToConfig()` transforms them into a strongly typed `APFederationConfig` object, applying defaults where values are not provides. Only three variables are required, the PDS URL, PDS hostname, and PDS admin token. All other setting have sensible defaults derived from these.

The public URL is inferred from the hostname: if the hostname is `localhost`, the URL uses HTTP with the configured port; otherwise, it constructs an HTTPS URL, reflection the assumption that production deployments will use TLS. Bridge account handles default to `mastodon.{hostname}` and `bluesky.{hostname}`, and their email addresses are generated as `noreply+{handle}@{hostname}`, since the PDS requires a unique email for each account. Background processors are enabled by default and their poll intervals are configurable. The Constellation processor defaults to 60 seconds, the DM notification processor to 5 minutes with a 10-minute batch delay. The database location defaults to an in-memory SQLite database, which is useful for testing but must be overriden with a file path for production use.

An `allowPrivateAddress` flag, disabled by default, permits Fedify to fetch resources from private IP ranges. This is necessary for the end-to-end test environment, where all services run on local addresses, but must remain disabled in production to prevent SSRF attacks. A complete list of all supported environment variables and their defaults is provided in section A.1.

[5] `https://www.w3.org/TR/activitystreams-vocabulary/`

3.2.2. PDS and AppView Clients

The `PDSClient` and `AppViewClient` modules provide thin wrappers around the ATProto XRPC APIs. The PDS client is used for all write operations and local record lookups as well as identity resolution and blob retrieval. The AppView client is used exclusively for read operations against the Bluesky AppView, such as fetching posts from users on external PDS instances that are not accessible through the local PDS. Separating the two clients reflects the architectural distinction in AT Protocol between the PDS as the authoritative data store and the AppView as a read-optimized aggregation layer.

3.2.3. Firehose Processor

The `FirehoseProcessor` subscribes to the PDS's `com.atproto.sync.subscribeRepos` WebSocket endpoint and drives the outbound federation pipeline. Each incoming frame is decoded from its CBOR binary encoding and parsed into a structured commit event containing the repository DID, a sequence number, and a list of operations. The processor filters operations to only those affecting collections for which a converter is registered, and skips events originating from the bridge accounts to prevent bridging loops. For create operations, it fetches the complete record from the PDS, converts it into an ActivityPub activity using the converter registry, and dispatches it to followers via Fedify. For delete operations, it constructs the corresponding `Delete` or `Undo` activity and dispatches it similarly. The processor also add newly created local posts to the `ap_monitored_post` table for later processing by the Constellation processor. On connection loss, the processor waits five seconds before automatically reconnecting.

3.2.4. Constellation Processor

The `ConstellationProcessor` runs as a periodic background job, polling at a configurable interval (default 60 seconds). On each run, it fetches a batch of up to 50 entries from the `ap_monitored_post` table, which tracks local posts that have been bridged to the Fediverse and may have received replies from users on other PDS instances. For each post, it queries the Constellation API for backlinks of type `app.bsky.feed.post reply.parent.uri`, which identifies posts on external PDS instances that reply to the monitored post. Each discovered reply is checked against the `ap_external_reply` deduplication table. New replies are fetched from the AppView, converted into `Create(Note)` activities attributed to the Bluesky bridge account, and delivered to the original post author's followers via Fedify. The `ap_external_reply` record is then created and the monitored post's `lastChecked` timestamp is updated.

3.2.5. DM Notification Processor

The `DMNotificationProcessor` polls the database at a configurable interval (default 5 minutes) for likes and reposts that have not yet triggered a notification and whose timestamp is older than a configurable batch delay (default 10 minutes), ensuring that a burst of engagements on a single post results in a single summary message rather than

individual notifications. Engagements are grouped by post author and then by post, and a summary direct message is sent to each author via the Bluesky Chat API using the Mastodon bridge account as the sender. The message includes the display names of the engaging Fediverse actors, resolved by fetching their ActivityPub profiles, and a truncated excerpt of the post content.

3.2.6. Database

All persistent state is stored in a SQLite database through Kysely. A second SQLite database is used exclusively by Fedify for its internal KV store and message queue. Schema changes are managed through numbered migration files, each exporting `up()` and `down()` functions, which are applied automatically on service startup.

Identity & Cryptography

- `ap_key_pair` – Stores the cryptographic key pairs used for HTTP signature signing. Each local PDS user gets two key pairs generated on first access: one RSA for compatibility with older ActivityPub implementations, and one Ed25519 for modern servers. The keys are stored as PEM-encoded strings.
- `ap_bridge_account` and `ap_bluesky_bridge_account` – Singleton tables that store the credentials for the two bridge accounts (see section 3.3).

Social Graph

- `ap_follow` – Records which ActivityPub actors follow which local PDS users. This is the core table for activity delivery, as it determines which users should receive which activities based on their follow relationships. It stores the follower's inbox URL and shared inbox URL for efficient delivery.

Content Mapping

- `ap_post_mapping` – Maps ATProto post URIs to their original ActivityPub Note IDs and author information. This table is essential for correct reply threading.
- `ap_external_reply` – Stores external Bluesky replies from other PDS instances that have been federated via the Constellation processor. Primarily for deduplication as the Constellation API is polled repeatedly.
- `ap_monitored_post` – The work queue for the Constellation processor, which tracks which posts need to be checked for new external replies.

Engagement Tracking

- `ap_like` and `ap_repost` – Stores incoming ActivityPub `Like` and `Announce` activities targeting local posts. The tables' main purpose are to track engagements for display (e.g. showing like counts) and batching engagement for DM notifications.

3.3. Bridge Account System

3.3.1. Why Two Bridge Accounts?

Fedisky uses two special "bridge accounts" to facilitate bridging between the AT Protocol and ActivityPub. The first is the Mastodon bridge account, which is hidden from users in ActivityPub and is used to post incoming Fediverse replies as ATProto posts on the PDS, so that they appear in the Bluesky user's thread and can be replied to and interacted with like normal posts. Its handle and display name can be configured by the operator, with the default being `mastodon.{hostname}` and "Mastodon Bridge" respectively.

The second one is the Bluesky bridge account, which is used to federate replies from Bluesky users on other PDS instances to the Fediverse. This allows users on Mastodon to see and interact with replies from users on other PDS instances, which would otherwise be invisible to the Fediverse. The Bluesky bridge account's default handle is `bluesky.{hostname}` and display name is "Bluesky Bridge", again configurable by the operator via environment variables.

3.3.2. Account Lifecycle

The two bridge accounts are managed by the `MastodonBridgeAccountManager` and `BlueskyBridgeAccountManager`, both subclassing an abstract `BridgeAccountManager`. On startup, each manager checks the database for stored credentials. If found, it attempts to refresh the session, falling back to a password login if the refresh token is expired. If no account record exists, it creates a new PDS account using the configured handle and a random password. Once initialized, each manager exposes the account's DID and a valid access token for use in XRPC calls. The Mastodon bridge account is hidden from ActivityPub discovery, while the Bluesky bridge account is exposed as a regular actor.

3.3.3. Attribution Model

Since both bridge accounts post content on behalf on users and do not carry any user identity in their handle or display name, we need to ensure proper attribution of content to the original authors. For incoming Fediverse replies posted by the Mastodon bridge account, we include the original author's handle in the first line of the post content, e.g. "@bob@mastodon.social replied:", followed by the actual reply content. Similarly, federated Bluesky replies from third-party PDS instances sent by the Bluesky bridge account include an attribution line with the original author's Bluesky handle, e.g. "alice.bsky.social replied:", followed by the reply content. In both cases, the handle is a clickable link to the original profile, allowing users to easily find and follow the original author if they wish. Additionally, since Mastodon content is formatted in HTML, we need to ensure all content is properly escaped to prevent Cross-Site-Scripting (XSS) vulnerabilities.

3.4. Core Flows

3.4.1. Actor and Identity Discovery

When a Mastodon user tries to follow a federated Bluesky user on the PDS, the Mastodon instance needs to discover the corresponding ActivityPub actor for that user in order to send the follow request. This flow is illustrated in Figure 3.2. The Mastodon instance first queries the WebFinger endpoint with the user's handle (e.g. `@alice@fedisky.social`) to discover the corresponding ActivityPub actor URL. Fedisky first constructs the AT-Proto handle by prepending the localpart (in this case, `alice`) to the PDS's hostname (e.g. `fedisky.social`), resulting in `alice.fedisky.social`. Note that the Activity-Pub handle domain and PDS domain do not have to match, but in this example we use the same domain for simplicity. Fedisky then resolves this handle using the PDS's `com.atproto.identity.resolveHandle` API, which returns the corresponding DID if a matching user is found. If a user is found, Fedisky constructs the ActivityPub actor URL using the user's DID, resulting in `https://fedisky.social/users/{did}`. This URL is returned to the Mastodon instance in the WebFinger response, as shown in Listing 3.1. In addition to the actor URL, the response also includes references to the user's profile page and avatar.

```
{
  "subject": "acct:alice@fedisky.social",
  "aliases": ["https://fedisky.social/users/did:plc:
    n3jcidccul6u3lif5q4rh42x"],
  "links": [
    {
      "rel": "self",
      "href": "https://fedisky.social/users/did:plc:
    n3jcidccul6u3lif5q4rh42x",
      "type": "application/activity+json"
    },
    {
      "rel": "http://webfinger.net/rel/profile-page",
      "href": "https://bsky.app/profile/alice.fedisky.social"
    },
    {
      "rel": "http://webfinger.net/rel/avatar",
      "href": "https://fedisky.social/xrpc/com.atproto.sync.getBlob?did=
    did%3Aplc%3An3jcidccul6u3lif5q4rh42x&cid=
    bafkreiahfo2qkotgr475q2kx4psbffojwup6fkchnw43y2i44uhanmy2em",
      "type": "image/jpeg"
    }
  ]
}
```

Listing 3.1: WebFinger response for `@alice@fedisky.social`

After receiving the WebFinger response, the Mastodon instance can then query the actor endpoint to fetch the user's actor document, which includes the user's profile information, public keys, and inbox URLs. The object conforms to the Activity Vocabulary `Person`

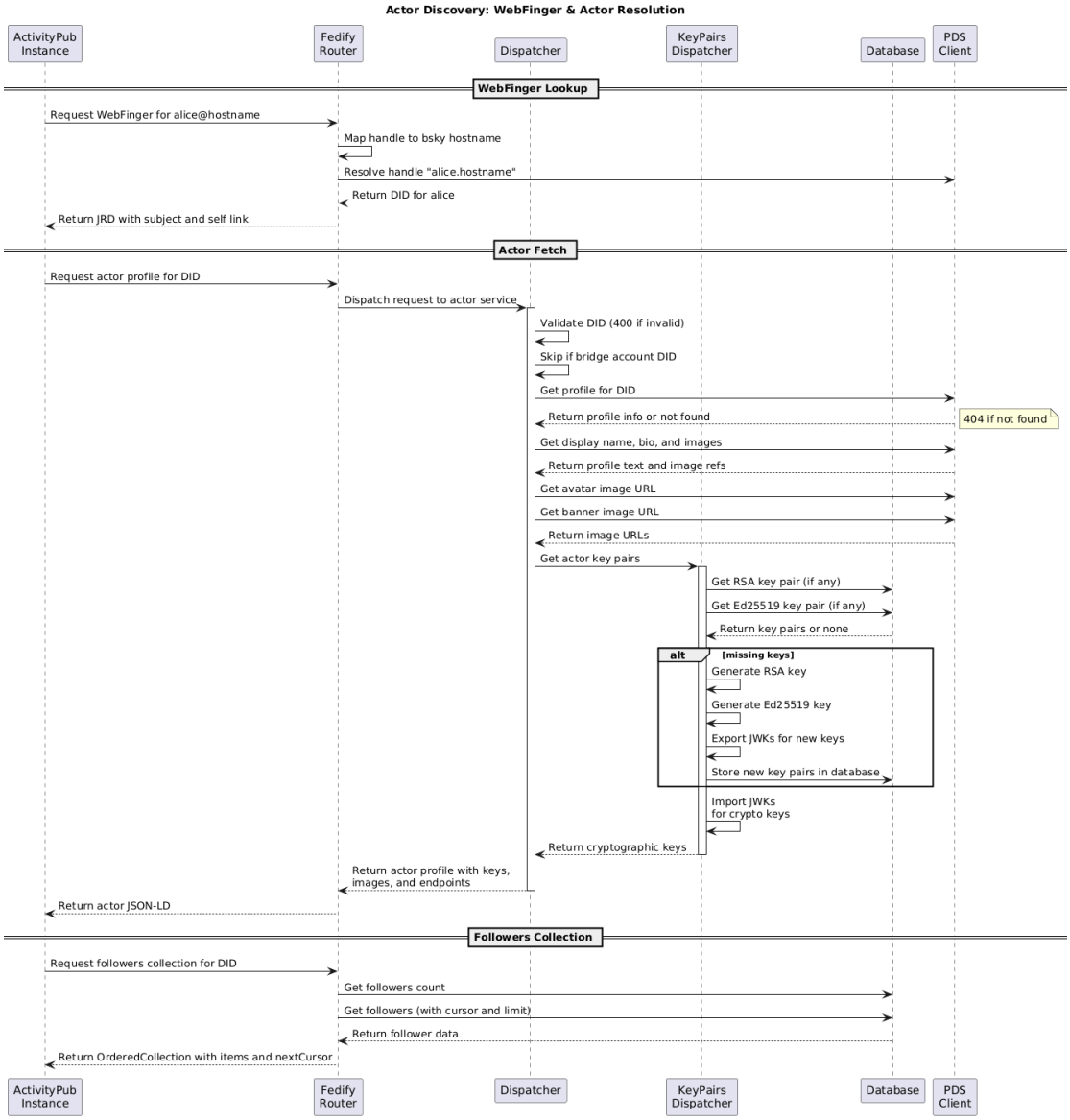

Figure 3.2.: Sequence diagram showing the actor and identity discovery flow when a Mastodon user tries to follow a Bluesky user on the PDS.

type as defined in [11]. To construct the actor document, Fedisky fetches the user's profile record from the PDS using the `com.atproto.repo.getRecord` API. From the profile record, Fedisky extracts the user's display name, description, as well as avatar and banner images. Finally, Fedisky retrieves the user's public keys from the database. If no keys exist yet, Fedisky generates new RSA and Ed25519 key pairs and stores them in the database.

In an effort to link accounts referring to the same identity, Barrett [1] proposes the use of account links. In this approach, instead of relying on a separate "meta account" that links all the user's accounts together, accounts reference each other. Platforms can then use these references to show highlighted links to these other accounts on different platforms. Fedisky follows this approach and includes the user's ATProto URI in the `alsoKnownAs` field of the actor document, allowing Fediverse instance to link the ActivityPub actor back to the original Bluesky user and profile.

3.4.2. Outbound: ATProto Post to ActivityPub Note

When a Bluesky user creates or deletes a post on the PDS, Fedisky propagates these changes to the Fediverse so that Mastodon users following that user receive updates. This flow is illustrated in the sequence diagram in Figure 3.3.

Fedisky receives real-time updates from the ATProto firehose, a WebSocket stream that delivers commits as CBOR[6]-encoded frames. Each frame contains repository operations (create, update, or delete) along with metadata such as the repository DID and sequence number. The firehose processor decodes the CBOR frame and extracts the repository DID, operations, and sequence information. It then filters operations to only process those that represent record creation or deletions for collections that Fedisky can convert to ActivityPub activities.

For create operations, the processor first fetches the complete record from the PDS using the `com.atproto.repo.getRecord` API, providing the repository DID, collection identifier (e.g. `app.bsky.feed.post`), and record key. Once the record is retrieved, the processor queries the record converter registry to obtain the appropriate converter for the collection type. The registry returns a converter instance that implements the conversion logic for that specific ATProto record type.

The converter then transforms the ATProto record into an ActivityPub `Create` activity containing a `Note` object. During conversion, if the post is a reply, the converter queries the database to look up the parent's post ActivityPub mapping, which maps the ATProto URI to the corresponding ActivityPub object ID and actor inbox. This mapping is necessary to construct the `inReplyTo` property of the ActivityPub Note. If the post contains embedded images or media, the converter builds the blob URLs using the PDS's `com.atproto.sync.getBlob` API, and includes them as attachments in the Note. Additionally, the converter transforms plain text into HTML and ATProto self-labels into ActivityPub content warnings if necessary. A detailed explanation of the conversion logic can be found in section 3.6.

[6] `https://atproto.com/specs/data-model`

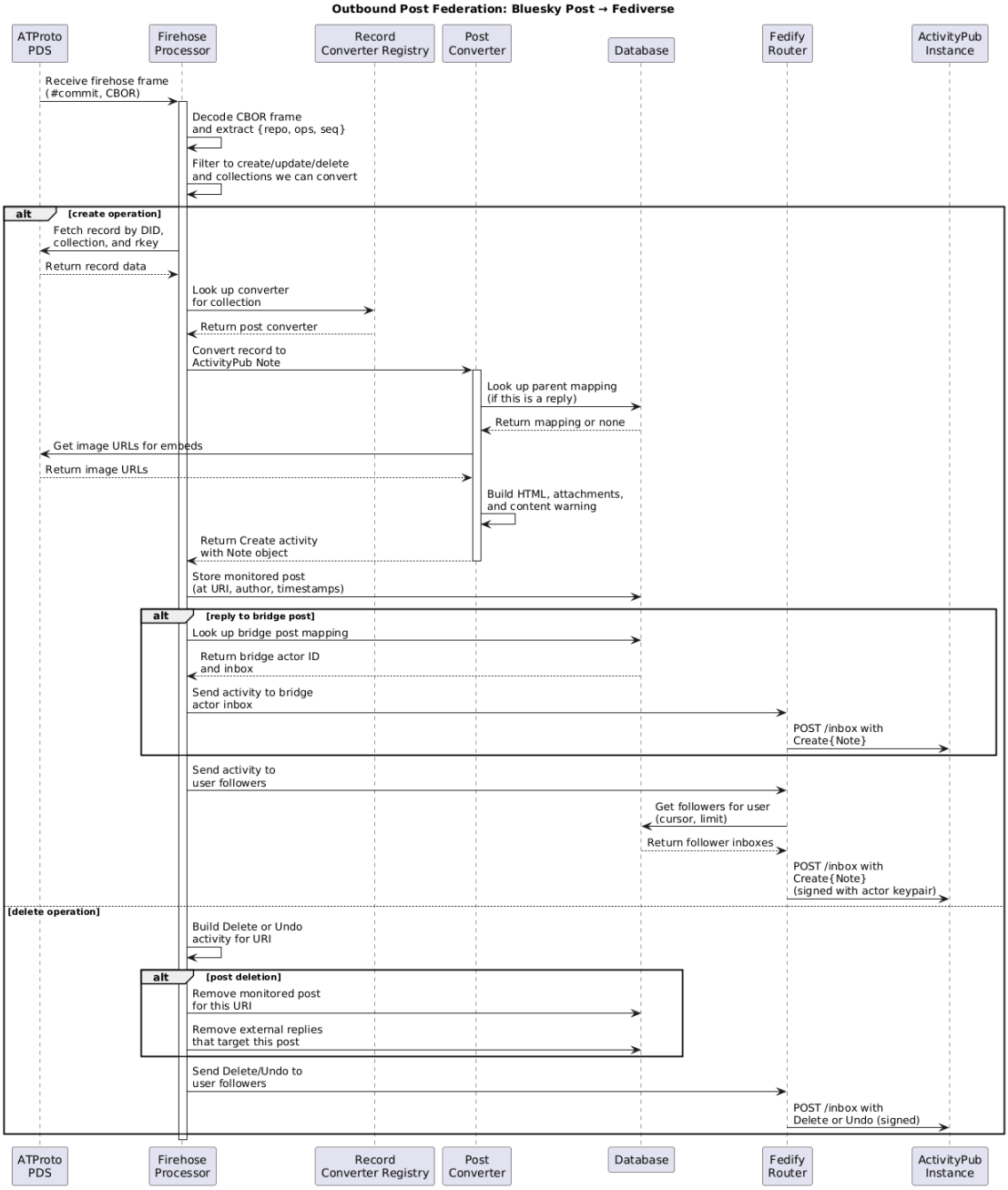

Figure 3.3.: Sequence diagram showing the outbound ATProto post to ActivityPub note conversion flow.

After conversion completes, Fedisky store the post mapping in the database, recording the ATProto URI, author DID, and creation timestamp. This mapping enables Fedisky to track which posts have been federated and to handle subsequent operations like deletions. If the post is a reply to a bridge account post (i.e. a post originally created by a Fediverse user and relayed back to Bluesky), Fedisky performs and additional step: It looks up the bridge post mapping to retrieve the bridge actor's ID and inbox URL, then sends the `Create` activity directly to that actor's inbox. This ensures that the original Fediverse author receives notifications about replies to their content.

Finally, Fedisky delivers the `Create` activity to all the author's federated followers. The Fedify router queries the database to retrieve the list of followers and for each follower, it determines the appropriate inbox URL (either the follower's personal inbox or their instance's shared inbox) and sends a request containing the activity to that inbox endpoint. The activity is signed using the author's private key to authenticate the request according to ActivityPub's HTTP signature specification.

For delete operations, the flow is simpler. The processor constructs either a `Delete` activity (for post deletions) or an `Undo` activity (for undoing a like or repost) targeting the ActivityPub object corresponding to the deleted ATProto record. If the deletion is for a post, Fedisky also cleans up any associated database entries related to that post. The `Delete` or `Undo` activity is then delivered to all followers using the same inbox delivery mechanism as create operations, ensuring that federated instances are notified of the deletion.

3.4.3. Inbound: ActivityPub Activity to ATProto Record

Figure 3.4 shows the inbound flow for incoming ActivityPub activities. All incoming activities from the Fediverse are delivered to Fedisky's inbox endpoints by the remote ActivityPub instance. Fedify first verifies the HTTP signature on each request and fetches the sending actor from the network before dispatching the activity to the appropriate handler. The handling logic then branches based on the activity type as follows:

Follow

When Fedisky receives a `Follow` activity, it validates the activity's required fields (identifier, actor, and object), then parses the object URI to extract the target user's DID. The follow relationship is persisted to the database, recording the follower's actor URI and their personal and shared inbox URLs for later use during activity delivery. Fedisky then sends an `Accept(Follow)` activity back to the actor's inbox, completing the handshake and activating the follow relationship on the remote instance.

Undo(Follow), Undo(Like), Undo(Announce)

These three undo variants are handled symmetrically. Fedisky locates the original record in the database by actor URI or activity ID, and removes it. No further action is required on the PDS.

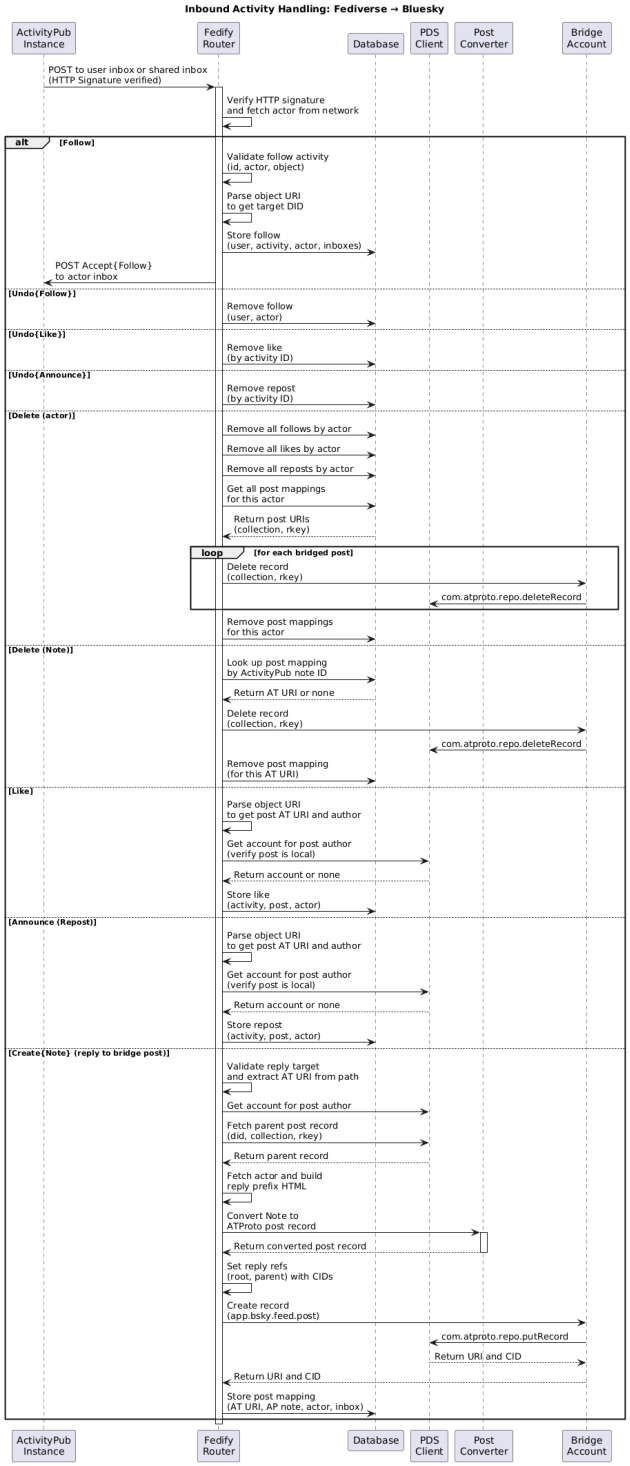

Figure 3.4.: Sequence diagram showing the inbound ActivityPub activity to ATProto record conversion flow.

Delete(Actor)

When an actor deletion is received, Fedisky performs a full cleanup of all data associated with that actor. It removes all follow, like, and repost records from the database. It then queries the database for all post mappings belonging to that actor, iterating over each bridged post and issuing a `com.atproto.repo.deleteRecord` call via the bridge account client to remove the corresponding ATProto record from the PDS. Finally, all remaining post mappings for the actor are removed from the database.

Delete(Note)

For individual note deletions, Fedisky looks up the post mapping by the ActivityPub note ID to retrieve the corresponding ATProto collection and record key. If a mapping is found, Fedisky issues a `com.atproto.repo.deleteRecord` call to the PDS to remove the record, then removes the post mapping entry from the database.

Like and Announce

When a `Like` or `Announce` activity is received, Fedisky parses the object URI to determine the target post's ATProto URI and the author's DID. It then queries the PDS to verify that an account with that DID is local to the PDS, discarding the activity if not. If the post is confirmed to be local, the engagement is stored in the database in the respective `ap_like` or `ap_repost` table, recording the activity ID, post URI, and actor URI.

Create(Note)

The most involved inbound case handles incoming Fediverse replies to posts that were originally bridged onto the PDS by the Mastodon bridge account. Fedisky first validates the reply target and extracts the parent post's ATProto URI from the path. It then fetches the account for the post's author and retrieves the full parent post record from the PDS using the `com.atproto.repo.getRecord` API, obtaining the record's CID required for correct reply threading. Next, Fedisky fetches the remote actor's profile and constructs an attribution prefix in HTML (e.g. "@bob@mastodon.social replied:") to be prepended to the post content. The `Note` is then passed to the post converter, which transforms it into an ATProto `app.bsky.feed.post` record. After conversion, Fedisky sets the reply references (`root` and `parent`), resolving the CIDs of both the root and parent records so that the reply is correctly threaded in the Bluesky client. The record is then written to the PDS via the `com.atproto.repo.putRecord` API using the bridge account client, which returns the new record's URI and CID. Finally, Fedisky stores a post mapping in the database, recording the ATProto URI, the original ActivityPub note ID, the actor URI, and the actor's inbox URL, enabling future operations such as deletions to locate the bridged post.

3.5. ActivityPub Interface

Using Fedify together with its Express HTTP integration[7], Fedisky exposes the following ActivityPub endpoints. Fedify handles the low-level protocol concerns: verifying and signing HTTP requests, and managing the message queue and KV store backed by a dedicated SQLite database. The application layer registers dispatchers that Fedify calls for each activity type or collection lookup, which in turn delegate to the appropriate database queries or converter logic.

3.5.1. Endpoints

- `GET /.well-known/webfinger`
  The WebFinger endpoint for user discovery. When a Mastodon instance encounters a handle such as `@alice@fedisky.social`, it will query this endpoint to discover the corresponding ActivityPub actor URL. Fedisky resolves the handle using the PDS's `com.atproto.identity.resolveHandle` API, and if a matching PDS user is found, constructs an ActivityPub actor URL using the user's DID as the unique identifier, i.e. `https://fedisky.social/users/{did}`.

- `GET /users/{did}`
  The actor endpoint. Returns an ActivityPub `Person` object representing the Bluesky user with the given DID. Includes the user's inbox, outbox, followers, and following URIs, as well as their public keys (both RSA and Ed25519), and profile information such as display name and avatar. It also includes a `alsoKnownAs` field with the user's ATProto URI, in order to link the ActivityPub actor back to the original PDS user and identity. The Mastodon bridge account does not have an ActivityPub actor representation.

- `POST /users/{did}/inbox`
  The inbox endpoint for receiving incoming activities from the Fediverse. This is where we receive activities such as `Follow` requests. When an activity is received, we verify the HTTP signature to ensure it is from a trusted source, and then dispatch it to the appropriate handler based on the activity type.

- `POST /inbox`
  A server-wide shared inbox endpoint that some Fediverse instances support for more efficient delivery. Fedisky also supports this endpoint for incoming activities, and dispatches them in the same way as the user-specific inbox.

- `GET /users/{did}/outbox`
  Paginated collection of posts, likes, and reposts, aggregated from the PDS using its `com.atproto.repo.listRecords` API. This allows Mastodon instance to fetch the user's content and engagements for display on their profile and timelines.

[7] `https://fedify.dev/manual/integration#express`

- `GET /users/{did}/followers` and `GET /users/{did}/following`
  Paginated collections of followers and following, based on the `ap_follow` table. This allows Mastodon instances to display the user's followers and following lists, and to determine which users they should receive activities from.

- `GET /posts/{uri}`
  Endpoint for fetching a specific post by its ATProto URI, used by Mastodon instances to fetch the content and metadata of a post when displaying it or when a user clicks on a link to the post. Fedisky resolves the ATProto URI using the PDS's `com.atproto.repo.getRecord` API, and returns an ActivityPub Note object with the post content, author information, and any media attachments.

- `GET /nodeinfo/2.1`
  The NodeInfo[8] endpoint providing metadata about the Fedisky instance, such as software name and version, and supported features. This is used by Mastodon instances to determine compatibility and capabilities of the Fedisky bridge.

3.5.2. Outbox

The outbox dispatcher aggregates records from all three bridged ATProto collections (posts, likes, and reposts) into a single, chronologically ordered ActivityPub outbox collection. For a given user, it queries the PDS's `com.atproto.repo.listRecords` API for each collection registered in the converter registry, then merges the results into a unified list. To produce a consistent chronological ordering across collections, the records are sorted by their record key, which in ATProto is a TID (timestamp-based identifier) that sorts lexicographically by creation time. The outbox is paginated using cursor-based pagination: each page contains up to 50 items, and the cursor for the next page is derived from the last record's key. An extra record is fetched beyond the page limit to determine whether additional pages exist.

Each record in the page is passed to its corresponding converter, which transforms it into an ActivityPub activity. Records whose conversion fails are filtered from the response, and errors logged via the wide events system. A separate object dispatcher handles direct lookups of individual posts by their ATProto URI. When a Mastodon instance fetches a post URL (e.g. `https://fedisky.social/posts/{uri}`), the dispatcher parses the ATProto URI from the path, fetches the record from the PDS, and returns the converted ActivityPub `Note` object.

3.6. Conversion Layer

The conversion layer provides bidirectional translation between ATProto records and ActivityPub objects. A `RecordConverterRegistry` maps ATProto collection identifiers (e.g. `app.bsky.feed.post`) to converter instances implementing a common `RecordConverter` interface. Each converter exposes two methods: `toActivityPub`, which transforms an

[8] `https://nodeinfo.diaspora.software/`

ATProto record into an ActivityPub activity and/or object, and `toRecord`, which performs the inverse transformation. The registry is populated at startup with converters for the three bridged collection types: posts, likes, and reposts.

3.6.1. Post Converter

The post converter handles the `app.bsky.feed.post` collection and is the most involved of the three converters, as it must handle bidirectional conversion including rich text, media, reply threading, mentions, and content warnings.

Outbound (ATProto to ActivityPub)

The `toActivityPub` method receives an ATProto post record and constructs a `Create(Note)` activity. The note's ID is derived from the ATProto URI, and its `to` field is set to the ActivityPub public collection to make the post visible to all Fediverse users. The author's followers URI is included in the `cc` field to ensure delivery.

If the post is a reply, the converter queries the database for the parent post's ActivityPub mapping. If a mapping exists (i.e. the parent was originally an ActivityPub note bridged onto the PDS), the original ActivityPub note ID is used as the `inReplyTo` value, preserving the correct reply threading on the Fediverse side. Otherwise, the converter constructs an ActivityPub object URI from the ATProto parent URI.

The post's plain text is converted to HTML by splitting on double newlines to produce `<p>` elements, with all content escaped using the `escape-html` NPM package[9] to prevent Cross-Site Scripting (XSS) vulnerabilities.

Inbound (ActivityPub to ATProto)

The `toRecord` method receives an ActivityPub note object and transforms it into an `app.bsky.feed.post` record. The HTML content is parsed in to plain text using the `html-to-text` NPM package[10] (see subsection 3.6.3 below), and the resulting text is truncated to 3000 bytes if necessary to respect Bluesky's post size limits [5]. If the note includes attachments, they are downloaded and uploaded to the PDS as blobs (see subsection 3.6.4). If the note specifies a `replyTarget`, the converter parses the ATProto URI from the ActivityPub URL to construct the reply reference. A record key is generated using ATProto's TID (timestamp-based identifier) scheme, and the record's CID is computed using Lexicon CBOR hashing.

3.6.2. Like and Repost Converters

The like and repost converters handle the `app.bsky.feed.like` and `app.bsky.feed.repost` collections respectively. Both converters are outbound-only: they implement `toActivityPub`

[9] `https://www.npmjs.com/package/escape-html`
[10] `https://www.npmjs.com/package/html-to-text`

but return null from `toRecord`, since incoming likes and reposts from the Fediverse are handled directly by the inbox handlers rather than through the conversion layer.

The like converter transforms an ATProto like record into an ActivityPub `Like` activity, while the repost converter produces an `Announce` activity. Both converters first check whether the subject post belongs to a local PDS user using the `isLocalUser` utility, which queries the PDS for the account. If the subject post is from an external PDS, the activity is discarded to avoid generating engagement notification for posts that are not managed by this Fedisky instance.

3.6.3. HTML ↔ Rich Text

ATProto and ActivityPub use fundamentally different text representations. ATProto stores post content as plain text with separate annotations called *facets*, which specify byte ranges and their semantic type (link, mention, or tag). ActivityPub, and more specifically Mastodon, stores content as HTML with inline anchor elements for links and mentions. When converting ATProto posts to ActivityPub notes, the converter splits the plain text on double newlines and wraps each paragraph in `<p>` tags. All text content is HTML-escaped to prevent injection of arbitrary markup.

The inverse conversion uses the `html-to-text` library with a custom `anchorCollector` formatter. This formatter walks the HTML DOM tree and, for each anchor element, extracts both the visible text content and the `href` URL. It also inspects the element's CSS classes: Mastodon marks mention links with the `mention` class, which allows the converter to distinguish mentions from regular hyperlinks. Elements with the `invisible` class (used by Mastodon to hide portions of long URLs for display purposes) are skipped during text extraction to produce clean output. The extracted links are collected separately and used in a subsequent step to reconstruct ATProto facets.

After HTML parsing, the converter reconstructs ATProto facets from the collected links. For regular hyperlinks, it locates the link's visible text within the plain text output using positional search, then computes the UTF-8 byte offsets required by ATProto's facet index format. For mention links, the converter extracts the DID from the ActivityPub actor URL (which follows the pattern `/users/{did}`), verifies that the mentioned user is local to the PDS, and constructs a mention facet with the resolved DID.

If the ActivityPub note specifies a language via the `LanguageString` type, the converter extracts it and includes it in the ATProto record's `langs` array, preserving language information across the bridge.

3.6.4. Media Handling

Outbound

When an ATProto post contains embedded images or video, the converter constructs ActivityPub `Document` attachments by building blob URLs using the PDS's `com.atproto.sync.getBlob` endpoint. Each attachment includes the blob's MIME type and alt text. The converter supports both the `app.bsky.embed.images` type and the `app.bsky.embed.video` type.

Inbound

For incoming ActivityPub notes with attachments, the converter downloads each remote media file and uploads it to the PDS as a blob. The blob handler enforces several security and resource constraints: it validates that URLs use HTTP or HTTPS and rejects private IP addresses (RFC 1918 ranges, localhost, and link-local addresses) to prevent Server-Side Request Forgery (SSRF) attacks. Downloads are subject to a 30-second timeout and a 10 MB size limit, enforced both via the `Content-Length` header and by checking the actual downloaded data size. The downloaded blobs are then classified by MIME type: image blobs are mapped to `app.bsky.embed.images` (limited to four images, matching Bluesky's constraint), while video blobs are mapped to `app.bsky.embed.video` (limited to a single video) [5, 3, 4].

3.6.5. Content Warning and Label Mapping

ATProto and ActivityPub represent content sensitivity differently. ATProto uses *self-labels*, machine-readable string identifiers such as `porn`, `sexual`, `nudity`, and `graphic-media` that moderation systems use to filter or blur content [2]. ActivityPub uses a boolean `as:sensitive` flag and a free-text `summary` field that Mastodon displays as a content warning [10].

When an ATProto post carries self-labels, the converter maps each recognized label to a human-readable description (e.g. `porn` becomes "Adult Content (Porn)"). The descriptions are joined into a comma-separated string and set as the note's `summary`, with the `as:sensitive` flag set to true. This allows Mastodon to display an appropriate content warning and give users the option to view the content if they choose.

In the reverse direction, the converter scans the content warning text for keywords using regular expressions (e.g. matching "nsfw", "adult content", or "explicit" to the `sexual` label). If the `as:sensitive` flag is set but no keywords match, the converter defaults to the `sexual` label as a general-purpose fallback. The matched labels are assembled into an ATProto structure with the `com.atproto.label.defs#selfLabel` type and attached to the post record.

3.7. Observability & Operations

3.7.1. Wide Events Logging

Fedisky implements the *wide events* pattern, where each request or background operation emits a single, context-rich structured log event at completion rather than scattering multiple log statements throughout the code path. This pattern enables powerful debugging and analytics because all relevant context for a given operation is contained in one log line, making it straightforward to filter, search, and correlate events [6, 9].

The implementation centers on the `WideEvent` class, a builder that accumulates key-value pairs over the lifetime of an operation. It supports dot-notation for nested fields (e.g. `actor.did`, `activity.type`), dedicated methods for attaching user context (`setUser`),

error information (`setError`), and outcome classification (`setOutcome`, which accepts `success`, `error`, or `ignored`). Each event automatically records a timestamp at creation and computes the operation's duration at emission time. Environment metadata such as the service name, package version, Git commit hash, and Node.js version is captured once at startup and included in every event, ensuring that log lines can be correlated with specific deployments.

For HTTP request, an Express middleware creates a `WideEvent` of type `http_request` at the start of each request, populating it with the HTTP method, path, user agent, and remote address. A request ID is extracted from standard tracing headers (`X-Request-Id`, `X-Correlation-Id`, `X-Trace-Id`, `Traceparent`), or generated as a UUID if none is present. The event is then made available to all downstream handlers and middleware through Node.js's `AsyncLocalStorage`, which propagates the event through the entire asynchronous call chain without requiring explicit parameter passing. Any handler in the request chain can retrieve the current event via `getWideEvent()` and enrich it with domain-specific fields. When the response finishes, the middleware automatically sets the HTTP status code and outcome, and emits the event. For background operations such as firehose message processing, events are created directly using `createWideEvent()`, enriched throughout the operation, and explicitly emitted upon completion.

Fedisky uses LogTape[11] as its logging framework, configured with two sinks: a console sink for local development, and an OpenTelemetry sink that exports structured logs to an observability backend. The OpenTelemetry integration is activated by importing the OpenTelemetry auto-instrumentation script at startup, which enables automatic instrumentation of HTTP requests, database calls, and other supported libraries like Fedify without code changes. This allows operators to connect Fedisky to any OpenTelemetry-compatible backend for distributed tracing and log aggregation.

3.7.2. HTTP Security

Fedisky applies several HTTP-level security measures. The `helmet`[12] middleware sets security-related HTTP headers (such as `X-Content-Type-Options`, `Strict-Transport-Security`, and `Content-Security-Policy`) to mitigate common web vulnerabilities. Request body size is limited to 256 KB to prevent denial-of-service through oversized payloads. Two rate limiters are applied: a general limiter allowing 1000 requests per 15-minute window, and a stricter inbox-specific limiter allowing 100 requests per minute per IP address, since the inbox endpoints are the primary target for unsolicited traffic from the Fediverse. Rate limit headers allow clients to inspect their remaining quota.

[11] `https://logtape.org/`
[12] `https://www.npmjs.com/package/helmet`

3.7.3. Testing

Unit Tests

The unit test suite uses Vitest[13] as the test runner and covers the core logic modules of the system: the conversion layer (post, like, and repost converters), the wide events logging system, the database layer, the firehose processor, the federation inbox handler, and the DM notification system. Tests are co-located with their source code in `tests/` subdirectories within each module. The conversion tests verify bidirectional translation correctness—for example, that an ATProto post with facets, embeds, and content labels is correctly converted to an ActivityPub note and back.

End-to-End Tests

The end-to-end test suite validates complete federation flows across service boundaries. The test environment is orchestrated by a shell script that provisions the following infrastructure:

- A real ATProto PDS instance running in Docker, configured with a test hostname (`bsky.test`).
- A Caddy reverse proxy providing TLS termination with locally-trusted certificates, since ActivityPub requires HTTPS for HTTP signatures and actor resolution.
- A mock ActivityPub server that simulates a Mastodon instance (`mastodon.test`), capable of sending and receiving ActivityPub activities and exposing an API for test assertions.
- A mock Constellation server that simulates the external reply discovery service.
- A native Fedisky instance connected to the PDS and configured with an in-memory database.

The test suite covers four categories of federation flows: actor discovery (WebFinger resolution and ActivityPub actor retrieval), follow federation (follow request delivery and acceptance), post federation (outbound post delivery to followers via the firehose pipeline), and Constellation-based external reply federation. Each test creates fresh PDS accounts, establishes follows between mock ActivityPub users and PDS users, performs actions (such as creating posts), and then asserts that the expected ActivityPub activities were delivered to the mock server. Tests use polling with timeouts to account for the asynchronous nature of the system, where activities flow through the firehose, conversion layer, and HTTP delivery pipeline before arriving at the remote server.

[13] `https://vitest.dev`

3.7.4. Deployment

Fedisky is packaged as a multi-architecture Docker image (AMD64 and ARM64) and published to the GitHub Container Registry. The CI/CD pipeline, implemented as a GitHub Actions workflow, builds and pushes the image on pushes to the `release` branch or on version tags. Images are tagged with the semantic version, the `major.minor` version, the full Git SHA, and `latest`.

The production container image uses a multi-stage build: a build stage compiles TypeScript and installs dependencies, and a production stage copies only the compiled output and production dependencies, reducing the final image size. The entrypoint uses `dumb-init`[14] to handle signal forwarding and prevent zombie processes.

The entrypoint script registers a `SIGTERM` handler that calls the service's `destroy()` method, which performs an ordered shutdown: the firehose processor, Constellation processor, and DM notification processor are stopped first, then the HTTP server is terminated using `http-terminator`[15] to allow in-flight requests to complete, and finally the database connection is closed. This ensures that no requests are dropped, and no data is lost during container restarts or rolling deployments.

An installer shell script automates the deployment of Fedisky alongside an existing PDS installation. The script reads the PDS's existing configuration (hostname, admin credentials), prompts the operator for ActivityPub-specific settings (hostname, bridge account preferences), and generates the environment configuration, Caddy reverse proxy rules, Docker Compose service definition, and `systemd` unit file. The Caddy configuration uses path-based routing to direct ActivityPub-specific paths (such as `/users/*`, `/inbox`, and `/.well-known/webfinger`) to the Fedisky container while forwarding all other traffic to the PDS. Requests with an `Accept: application/activity+json` header are also routed to Fedisky, enabling content negotiation on shared paths. A companion updater script handles upgrades by pulling the latest image, backing up the configuration, performing a rolling restart of the Fedisky container, and cleaning up old Docker images. The Docker Compose configuration declares a health dependency between the Fedisky and PDS containers, ensuring that Fedisky only starts after the PDS's health check passes.

3.8. Limitations

[14] `https://github.com/Yelp/dumb-init`
[15] `https://www.npmjs.com/package/http-terminator`

4. Results

5. Discussion

5.1. Sidecar Architecture

As described in the previous chapter, Fedisky is designed as a *sidecar container* as defined in [7, Ch. 3] and runs alongside its *application container*, a Bluesky PDS. Sidecars are a common architectural pattern, where a secondary container provides auxiliary functionality to the primary application container, often without the application knowing. Sidecar containers share system resources and are scheduled to run in sync with the application container [7, p. 21]. This design allows Fedisky to operate independently while still closely integrating with the PDS.

An alternative design could have been a centralized bridging service that bridges *all* of Bluesky into the Fediverse, which is realized by Bridgy Fed (subsection 2.3.1). Compared to a centralized bridging service, a sidecar container running alongside a single Bluesky PDS instance, can share resources with the PDS, most importantly the server's hostname, which allows it to operate under the same domain and thus provide a single shared identity for users across Bluesky and Mastodon, e.g. `alice.fedisky.social` in Bluesky becomes `@alice@fedisky.social` in Mastodon. Additionally, the sidecar pattern allows for operator autonomy, where each PDS operator can choose to deploy Fedisky independently, or not. It also reduces the scope of the bridge, as it only needs to handle traffic for a single PDS instance, simplifying development and maintenance.

Another approach would be to fork and modify the PDS itself, thus integrating bridging functionality directly into the PDS. We considered this approach initially when designing Fedisky, but chose the sidecar pattern instead for several reasons. Forking the PDS would significantly increase the maintenance burden and development complexity, as one would need to keep up with upstream changes constantly to ensure compatibility. The sidecar avoids this by only consuming the PDS's public APIs, which are stable contracts based on AT Protocol Lexicons [8]. In addition, the sidecar pattern allows for better separation of concerns, as all bridging logic is contained within the sidecar, rather than the PDS being responsible for both AT Protocol and ActivityPub compliance and therefore tightly coupling two protocol implementations, making each harder to reason about and test in isolation. Furthermore, the sidecar gives more deployment flexibility, as operators can choose to start or stop the bridge independently of the PDS. Since communication between the sidecar and the PDS is done through HTTP APIs, the sidecar can be developed a different technology stack, and is automatically compatible with any PDS implementation that adheres to the AT Protocol specifications.

6. Conclusion

Bibliography

[1] Ryan Barrett. *Bridging Identity with Account Links*. A New Social. Aug. 13, 2025. URL: https://blog.anew.social/bridging-identity-with-account-links/ (visited on 02/19/2026).

[2] Bluesky Social PBC. *Labels and Moderation | Bluesky*. URL: https://docs.bsky.app/docs/advanced-guides/moderation (visited on 02/24/2026).

[3] Bluesky Social PBC and Contributors. *app.bsky.embed.images Lexicon*. URL: https://github.com/bluesky-social/atproto/blob/main/lexicons/app/bsky/embed/images.json (visited on 02/24/2026).

[4] Bluesky Social PBC and Contributors. *app.bsky.embed.video Lexicon*. URL: https://github.com/bluesky-social/atproto/blob/main/lexicons/app/bsky/embed/video.json (visited on 02/24/2026).

[5] Bluesky Social PBC and Contributors. *app.bsky.feed.post Lexicon*. URL: https://github.com/bluesky-social/atproto/blob/main/lexicons/app/bsky/feed/post.json (visited on 02/24/2026).

[6] Ivan Burmistrov. *All You Need Is Wide Events, Not "Metrics, Logs and Traces"*. A Song Of Bugs and Patches. Feb. 15, 2024. URL: https://isburmistrov.substack.com/p/all-you-need-is-wide-events-not-metrics (visited on 02/25/2026).

[7] Brendan Burns. *Designing Distributed Systems: Patterns and Paradigms for Scalable, Reliable Services, Using Kubernetes*. Second Edition. Sebastopol: O'Reilly, 2024. 200 pp. ISBN: 978-1-0981-5635-0.

[8] *HTTP Reference | Bluesky*. URL: https://docs.bsky.app/docs/category/http-reference (visited on 02/12/2026).

[9] Charity Majors, Liz Fong-Jones, and George Miranda. *Observability Engineering: Achieving Production Excellence*. First edition. Beijing Boston Farnham Sebastopol Tokyo: O'Reilly, 2022. 295 pp. ISBN: 978-1-4920-7644-5.

[10] Mastodon GmbH. *ActivityPub - Mastodon Documentation*. Nov. 21, 2025. URL: https://docs.joinmastodon.org/spec/activitypub/ (visited on 02/24/2026).

[11] James Snell and Evan Prodromou. *Activity Vocabulary*. W3C Recommendation. W3C, May 2017. URL: https://www.w3.org/TR/2017/REC-activitystreams-vocabulary-20170523/ (visited on 02/19/2026).

[12] Jessica Tallon and Christine Lemmer-Webber. *ActivityPub*. W3C Recommendation. W3C, Jan. 2018. URL: https://www.w3.org/TR/2018/REC-activitypub-20180123/ (visited on 02/19/2026).

A. Appendix

A.1. Fedisky Environment Variables

Table A.1 lists all environment variables supported by Fedisky. Variables marked as required must be set for the service to start; all others have default values as specified.

Variable	Type	Default
PDS_URL	String	**Required**
PDS_HOSTNAME	String	**Required**
PDS_ADMIN_TOKEN	String	**Required**
AP_PORT	Integer	2588
AP_HOSTNAME	String	localhost
AP_PUBLIC_URL	String	Derived from hostname
AP_VERSION	String	0.0.0
AP_DB_LOCATION	String	:memory:
AP_FIREHOSE_ENABLED	Boolean	true
AP_FIREHOSE_CURSOR	String	–
AP_MASTODON_BRIDGE_ENABLED	Boolean	true
AP_MASTODON_BRIDGE_HANDLE	String	mastodon.{hostname}
AP_MASTODON_BRIDGE_DISPLAY_NAME	String	Mastodon Bridge
AP_MASTODON_BRIDGE_DESCRIPTION	String	See source
AP_MASTODON_BRIDGE_AVATAR_URL	String	Mastodon logo URL
AP_BLUESKY_BRIDGE_ENABLED	Boolean	true
AP_BLUESKY_BRIDGE_HANDLE	String	bluesky.{hostname}
AP_BLUESKY_BRIDGE_DISPLAY_NAME	String	Bluesky Bridge
AP_BLUESKY_BRIDGE_DESCRIPTION	String	See source
AP_BLUESKY_BRIDGE_AVATAR_URL	String	Bluesky logo URL
AP_CONSTELLATION_URL	String	Empty (disabled)
AP_CONSTELLATION_POLL_INTERVAL	Integer	60000
AP_APPVIEW_URL	String	public.api.bsky.app
AP_DM_NOTIFICATIONS_ENABLED	Boolean	true
AP_DM_NOTIFICATIONS_POLL_INTERVAL	Integer	300000
AP_DM_NOTIFICATIONS_BATCH_DELAY	Integer	600000
AP_ALLOW_PRIVATE_ADDRESS	Boolean	false

Table A.1.: Fedisky environment variables.