



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 L<sup>A</sup>T<sub>E</sub>X 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 L<sup>A</sup>T<sub>E</sub>X, for more information check out <https://en.wikibooks.org/wiki/LaTeX>.



# **Kurzfassung**

Das ist eine deutsche Kurzfassung meiner in Englisch verfassten Masterarbeit.



# Contents

<b>Acknowledgements</b>	i
<b>Abstract</b>	iii
<b>Kurzfassung</b>	v
<b>List of Tables</b>	ix
<b>List of Figures</b>	xi
<b>List of Algorithms</b>	xiii
<b>Listings</b>	xv
<b>1. Introduction</b>	1
<b>2. Background and Related Work</b>	3
2.1. AT Protocol . . . . .	3
2.2. ActivityPub . . . . .	3
2.3. Related Work . . . . .	3
2.3.1. Bridgy Fed . . . . .	3
2.3.2. Wafrn . . . . .	3
<b>3. Fedisky – An ActivityPub Federation Sidecar for Bluesky PDS</b>	5
3.1. Overview . . . . .	5
3.1.1. System Context . . . . .	5
3.1.2. Technology Stack . . . . .	6
3.1.3. Subsystems . . . . .	6
3.2. Core Flows . . . . .	9
3.2.1. Actor and Identity Discovery . . . . .	9
3.2.2. Outbound: ATPROTO Post to ActivityPub Note . . . . .	11
3.2.3. Inbound: ActivityPub Activity to ATPROTO Record . . . . .	15
3.3. Data Model . . . . .	16
3.3.1. Database Schema . . . . .	16
3.3.2. Migration Strategy . . . . .	17
3.4. Bridge Account System . . . . .	17
3.4.1. Why Two Bridge Accounts? . . . . .	17
3.4.2. Attribution Model . . . . .	17

## *Contents*

3.5. Inbound Federation . . . . .	18
3.5.1. ActivityPub Endpoints . . . . .	18
3.5.2. Inbox Processing . . . . .	20
3.5.3. Reply Bridging . . . . .	20
3.5.4. Post Mapping for Reply Threading . . . . .	20
3.5.5. Engagement Notifications . . . . .	20
3.6. Conversion Layer . . . . .	20
3.6.1. Post Converter . . . . .	20
3.6.2. HTML ↔ Rich Text . . . . .	20
3.6.3. Media Handling . . . . .	20
3.6.4. Edge Cases . . . . .	20
3.7. Observability & Operations . . . . .	20
3.7.1. Wide Events Logging . . . . .	20
3.7.2. Testing . . . . .	20
3.7.3. Deployment . . . . .	20
<b>4. Results</b>	<b>21</b>
<b>5. Discussion</b>	<b>23</b>
5.1. Sidecar Architecture . . . . .	23
<b>6. Conclusion</b>	<b>25</b>
<b>Bibliography</b>	<b>27</b>
<b>A. Appendix</b>	<b>29</b>

## List of Tables



# List of Figures

3.1. System Context Diagram showing how the Fedisky sidecar interacts with other systems and users. . . . .	6
3.2. Sequence diagram showing the actor and identity discovery flow when a Mastodon user tries to follow a Bluesky user on the PDS. . . . .	10
3.3. Sequence diagram showing the outbound ATProto post to ActivityPub note conversion flow. . . . .	12
3.4. Sequence diagram showing the inbound ActivityPub activity to ATProto record conversion flow. . . . .	14



## List of Algorithms



# Listings

3.1. WebFinger response for @alice@fedisky.social . . . . .	9
---	---



# **1. Introduction**



## **2. Background and Related Work**

### **2.1. AT Protocol**

### **2.2. ActivityPub**

ActivityPub [5]

### **2.3. Related Work**

#### **2.3.1. Bridgy Fed**

#### **2.3.2. Wafrn**



## 3. Fedisky – An ActivityPub Federation Sidecar for Bluesky PDS

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. 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.

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<sup>1</sup>, 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.

---

<sup>1</sup><https://constellation.microcosm.blue/>

### 3. Fedisky – An ActivityPub Federation Sidecar for Bluesky PDS

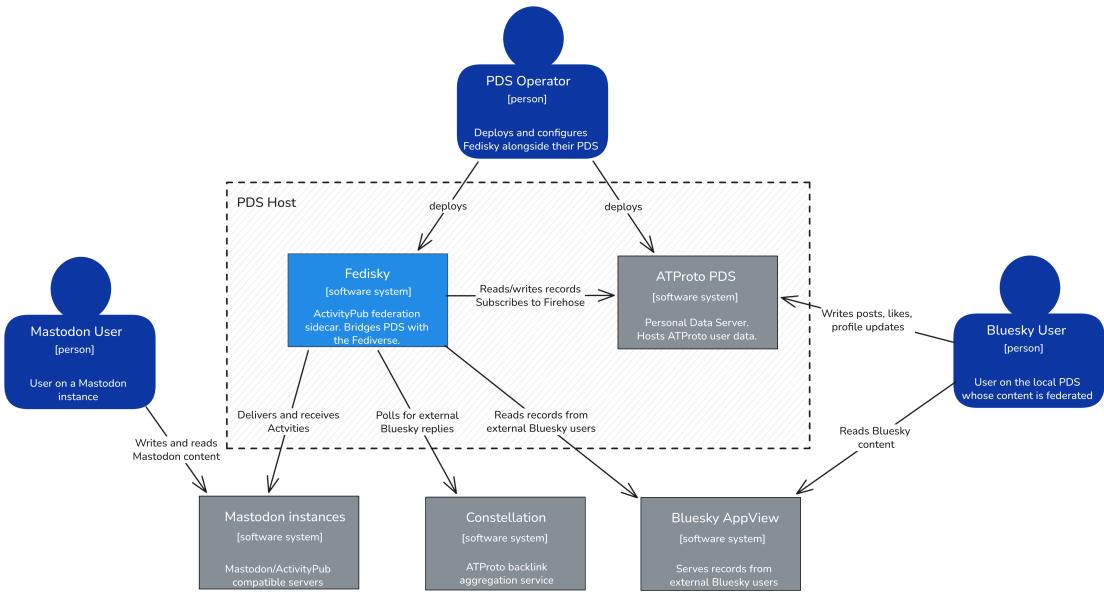


Figure 3.1.: System Context Diagram showing how the Fedisky sidecar interacts with other systems and users.

#### 3.1.2. Technology Stack

Fedisky is implemented in TypeScript and runs on Node.js. It stores its data in a SQLite<sup>2</sup> database using Kysely<sup>3</sup> as a type-safe query builder. For federation functionality, Fedisky uses Fedify<sup>4</sup>, 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 and verifying HTTP requests, providing type-safe objects for Activity Vocabulary<sup>5</sup> 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.

#### 3.1.3. Subsystems

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, `ApplicationContext`, which holds references to all major services:

<sup>2</sup><https://sqlite.org/>

<sup>3</sup><https://kysely.dev/>

<sup>4</sup><https://fedify.dev/>

<sup>5</sup><https://www.w3.org/TR/activitystreams-vocabulary/>

### 3.1. Overview

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 three background processors described below.

#### 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.

#### 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 adds 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.

#### Federation Layer

The federation layer is implemented using Fedify as exposed via an Express HTTP integration. It registers the WebFinger, actor, inbox, outbox, followers, following, and NodeInfo endpoints described in subsection 3.5.1. Fedify handles the low-level ActivityPub 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.

#### Inbox Handlers

The inbox module registers Fedify listeners for each supported inbound activity type: `Follow`, `Undo`, `Delete`, `Like`, `Announce`, and `Create`. Each handler performs the rele-

### 3. Fedisky – An ActivityPub Federation Sidecar for Bluesky PDS

vant database mutation and, in case of Follow, dispatches an `Accept` response. The `Create(Note)` handler additionally invokes the post converter to produce an ATProto record and write it to the PDS via the Mastodon bridge account, as described in subsection 3.2.3.

#### Conversion Layer

The conversion layer provides bidirectional translation between ATProto records and ActivityPub objects. A `RecordConverterRegistry` maps ATProto collection identifiers to a converter instance implementing a common interface with `toActivityPub` and `toRecord` methods. The post converter is the most complex: in the outbound direction it transforms ATProto facets into ActivityPub `Mention` tags, embeds media attachments, converts self-labels into content warnings, and constructs the correct `inReplyTo` chain. In the inbound direction it parses the HTML content of an ActivityPub `Note`, reconstructs ATProto facets, and prepends the attribution line. Supporting utilities handle blob downloading and upload, HTML parsing, and label mapping. A more detailed explanation of the conversion logic can be found in section 3.6.

#### Bridge Account System

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.

#### 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.

## 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.

## Database Layer

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.

## 3.2. Core Flows

### 3.2.1. 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 ActivityPub 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.

```

1  {
2      "subject": "acct:alice@fedisky.social",
3      "aliases": ["https://fedisky.social/users/did:plc:
4          n3jcidccul6u3lif5q4rh42x"],
5      "links": [
6          {
7              "rel": "self",
8              "type": "application/activity+json",
9              "url": "https://fedisky.social/users/did:plc:
10             n3jcidccul6u3lif5q4rh42x"
11          }
12      ]
13  }
```

### 3. Fedisky – An ActivityPub Federation Sidecar for Bluesky PDS

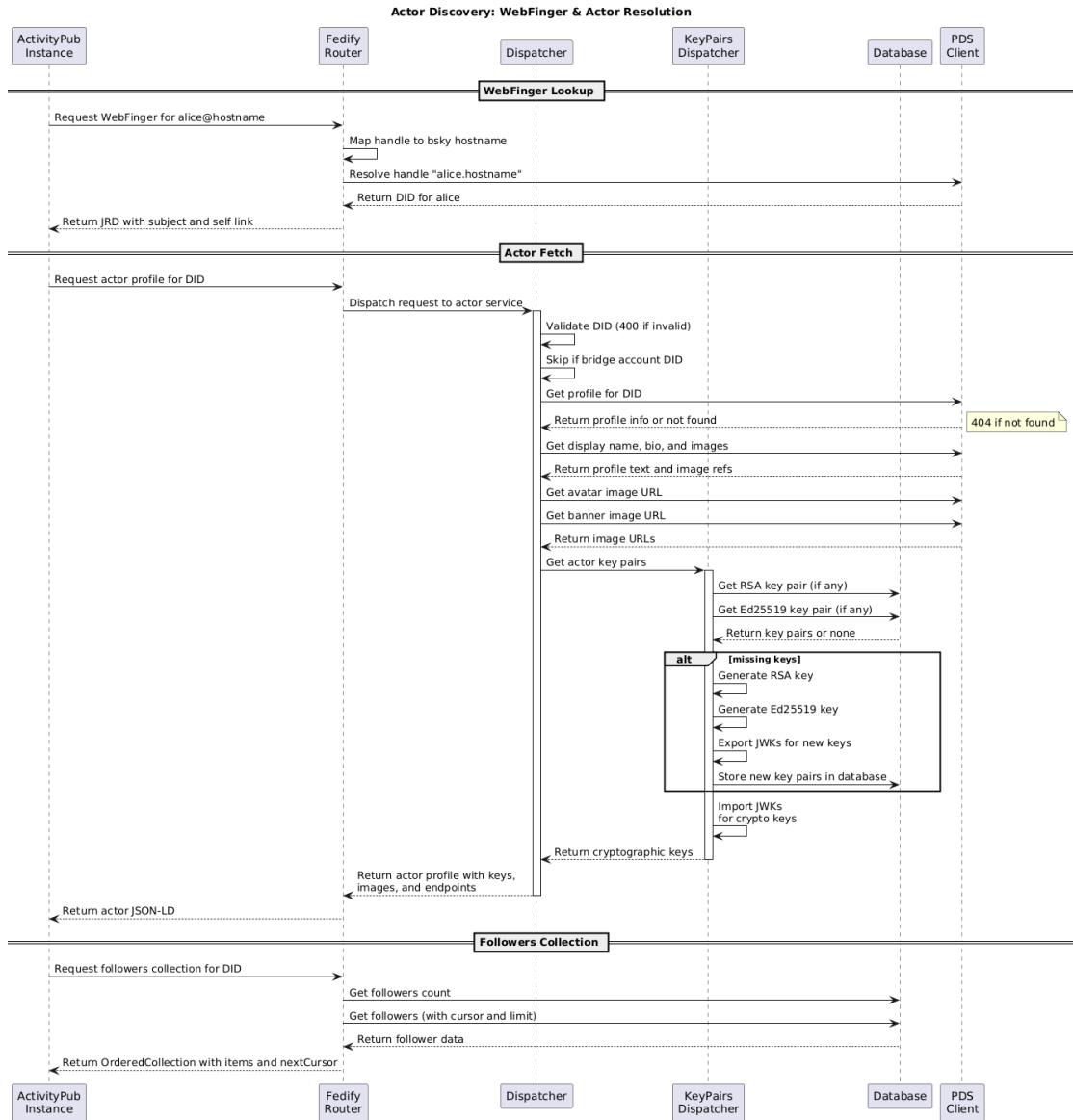


Figure 3.2.: Sequence diagram showing the actor and identity discovery flow when a Mastodon user tries to follow a Bluesky user on the PDS.

```

7   "href": "https://fedisky.social/users/did:plc:
8     n3jcidccul6u3lif5q4rh42x",
9     "type": "application/activity+json"
10    },
11    {
12      "rel": "http://webfinger.net/rel/profile-page",
13      "href": "https://bsky.app/profile/alice.fedisky.social"
14    },
15    {
16      "rel": "http://webfinger.net/rel/avatar",
17      "href": "https://fedisky.social/xrpc/com.atproto.sync.getBlob?did=
18        did%3Aplc%3An3jcidccul6u3lif5q4rh42x&cid=
19        bafkreiahfo2qkotgr475q2kx4psbffojwup6fkchnw43y2i44uhanmy2em",
20      "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** type as defined in [4]. 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.

### 3.2.2. 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<sup>6</sup>-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

<sup>6</sup><https://atproto.com/specs/data-model>

### 3. Fedisky – An ActivityPub Federation Sidecar for Bluesky PDS

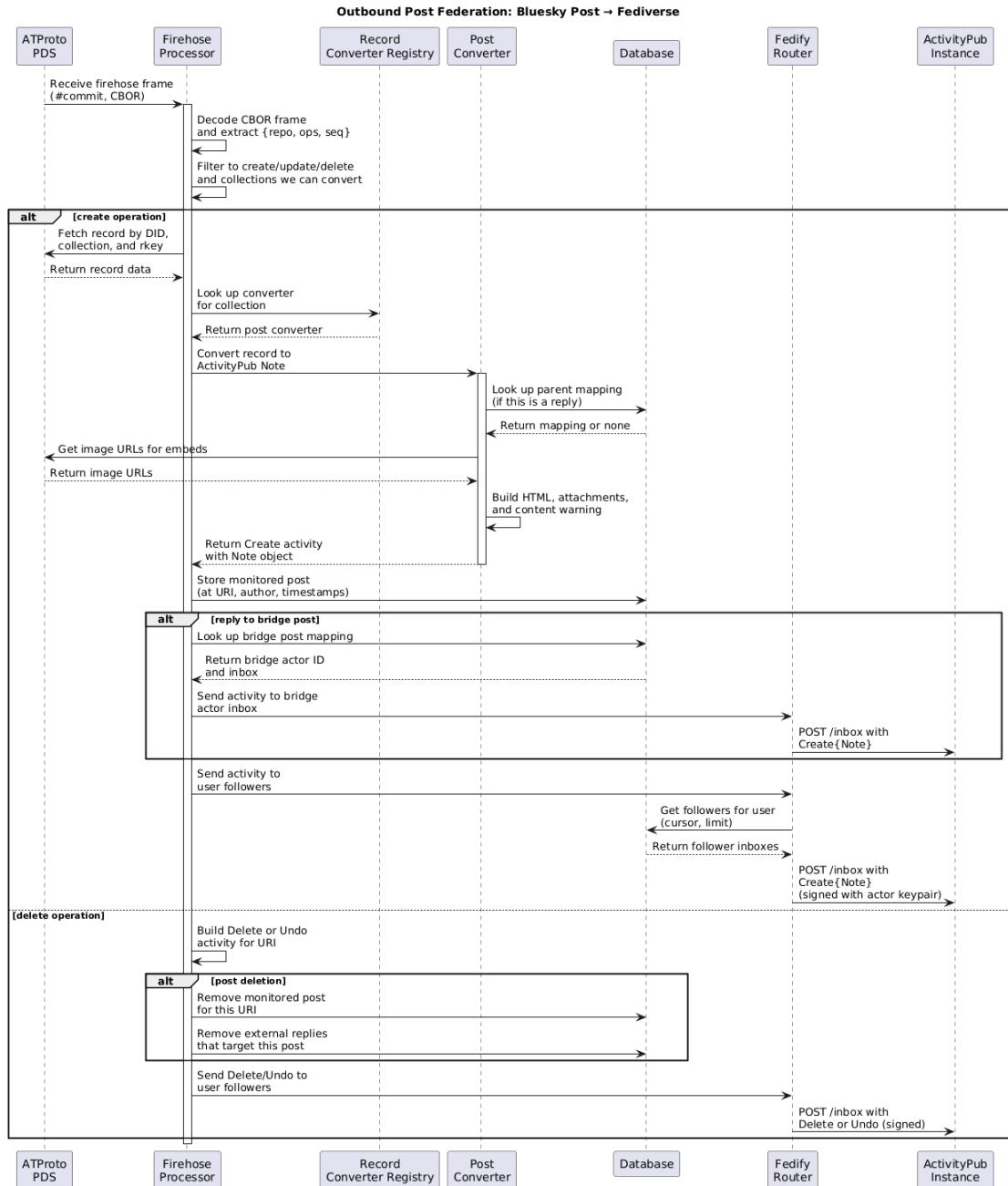


Figure 3.3.: Sequence diagram showing the outbound ATPROTO post to ActivityPub note conversion flow.

### 3.2. Core Flows

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.

After conversion completes, Fedisky stores 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 an 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. Fedisky – An ActivityPub Federation Sidecar for Bluesky PDS

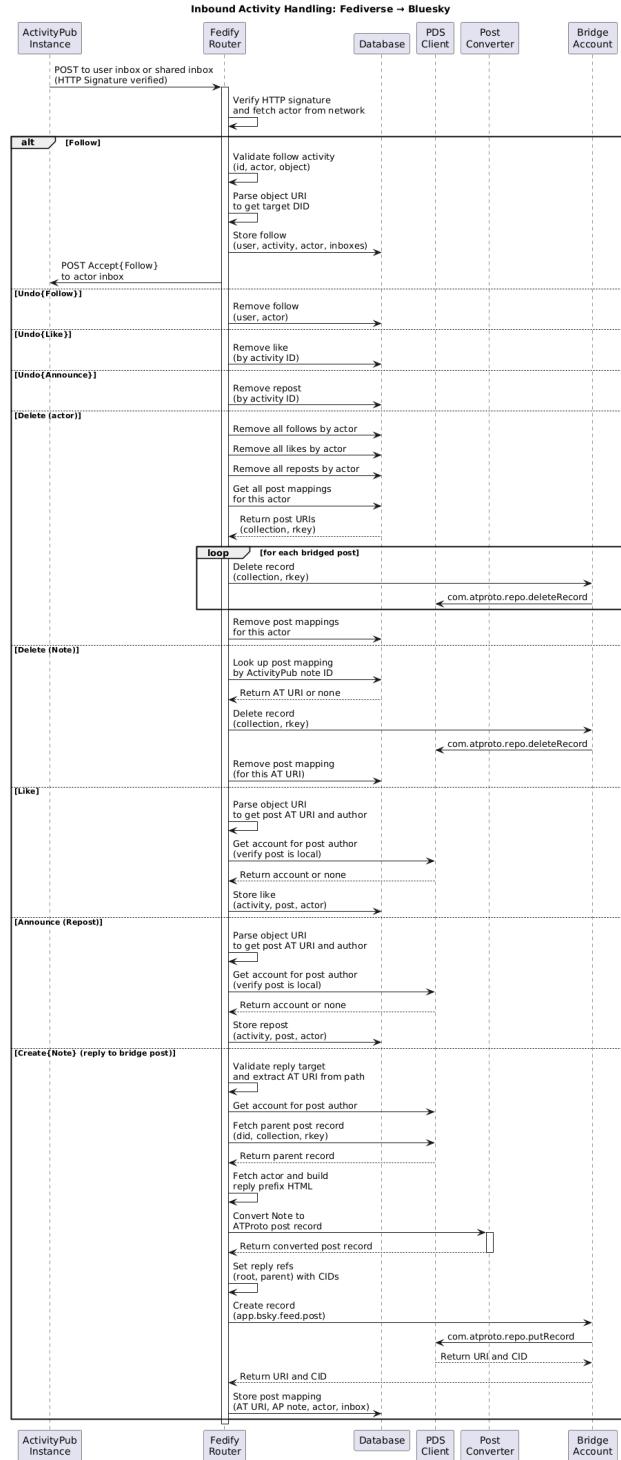


Figure 3.4.: Sequence diagram showing the inbound ActivityPub activity to ATProto record conversion flow.

### 3.2.3. Inbound: ActivityPub Activity to ATProto Record

`??` 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.

#### 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.

### 3. Fedisky – An ActivityPub Federation Sidecar for Bluesky PDS

#### 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.

## 3.3. Data Model

### 3.3.1. Database Schema

#### 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_bsky_bridge_account` – Singleton tables that store the credentials for the two bridge accounts (see section 3.4).

#### 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.

### 3.4. Bridge Account System

- `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.

#### 3.3.2. Migration Strategy

Migrations are implemented using Kysely's built-in migration system, which allows us to define schema changes in a type-safe manner. Each migration is defined as a numbered TypeScript file exporting `up()` and `down()` functions. On service startup, pending migrations are automatically applied in order. This approach allows us to evolve the database schema over time while ensuring data integrity and providing a clear history of schema changes.

### 3.4. Bridge Account System

#### 3.4.1. 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.

#### 3.4.2. 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.

### 3. Fedisky – An ActivityPub Federation Sidecar for Bluesky PDS

“@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.

## 3.5. Inbound Federation

### 3.5.1. ActivityPub Endpoints

Using Fedify together with its Express HTTP integration<sup>7</sup>, we expose the following ActivityPub 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.

---

<sup>7</sup><https://fedify.dev/manual/integration#express>

### 3.5. Inbound Federation

- 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.

- 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<sup>8</sup> 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.

---

<sup>8</sup><https://nodeinfo.diaspora.software/>

*3. Fedisky – An ActivityPub Federation Sidecar for Bluesky PDS*

3.5.2. **Inbox Processing**

3.5.3. **Reply Bridging**

3.5.4. **Post Mapping for Reply Threading**

3.5.5. **Engagement Notifications**

**3.6. Conversion Layer**

3.6.1. **Post Converter**

3.6.2. **HTML ↔ Rich Text**

3.6.3. **Media Handling**

3.6.4. **Edge Cases**

**3.7. Observability & Operations**

3.7.1. **Wide Events Logging**

3.7.2. **Testing**

**Unit Tests**

**End-to-End Tests**

3.7.3. **Deployment**

## 4. Results



# 5. Discussion

## 5.1. Sidecar Architecture

As described in the previous chapter, Fedisky is designed as a *sidecar container* as defined in [2, 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 [2, 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 [3]. 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] 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.
- [3] *HTTP Reference / Bluesky*. URL: <https://docs.bsky.app/docs/category/http-reference> (visited on 02/12/2026).
- [4] 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).
- [5] 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

here you can put further things you want to add like transcripts, questionnaires, raw data...