EPiC Series in Computing

Volume 109, 2026, Pages 220–230

Proceedings of EUNIS 2026 Annual Congress

Bridgit: A Lightweight Solution for FAIR Research Data Management in the Enterprise File Sync & Share Service Sciebo

Daniel Müller[1], Holger Angenent[2], Holger Przibytzin[3] and Raimund Vogl[4]

[1] University of Münster daniel.mueller@uni-muenster.de
[2] University of Münster holger.angenent@uni-muenster.de
[3] University and State Library Münster holger.przibytzin@uni-muenster.de
[4] University of Münster rvogl@uni-muenster.de

Abstract

This paper introduces bridgit, a modern web application designed to provide basic functionalities required for managing research data in accordance with the FAIR[2] principles[1]. It also explains its technical architecture. Bridgit was developed as part of the DFG-funded Sciebo Research Data Services project[2] in collaboration with the universities of Münster and Potsdam.

Bridgit's core objective is to provide researchers with basic research data management (RDM) functions that are tailored to their needs and integrated into their familiar working environment: Sciebo[3]. It also combines established RDM services to provide researchers with user-friendly, end-to-end workflows.

Bridgit allows researchers to annotate research objects with structured metadata, create and maintain data management plans and upload data sets to external repositories, such as Zenodo and OSF. Technically, bridgit is designed for seamless integration into institutional infrastructures and is embedded in the Enterprise File Sync & Share (EFSS) service sciebo.

A key contribution of the sciebo RDS project is the design and implementation of a modular architecture that takes into account heterogeneous research workflows and diverse third-party integrations. As a result, bridgit consists of several loosely coupled services that communicate via a messaging system that enables automated workflows and extensibility. The system supports configurable metadata profiles and a dynamic metadata editor. To enable publication to heterogeneous external systems, bridgit introduces extensible connectors that allow interaction with arbitrary target platforms. A central challenge is the integration of external authorization mechanisms; the platform therefore provides a robust approach for handling OAuth2 and additional authorization schemes required by third-party services. The architecture further emphasizes low-configuration deployment and maintainability within Kubernetes environments.

Finally, we outline future work including sharing of projects, realtime collaboration and AI-assisted metadata extraction.

[1] https://www.go-fair.org/fair-principles/
[2] https://sciebo-rds.github.io/
[3] https://hochschulcloud.nrw/en/

1 Introduction

As part of the sciebo RDS (sciebo research data services) project[1], we embed basic research data management functionalities directly into sciebo – the sync-and-share service that researchers at Universities in North Rhine-Wesphalia already use for storing and organising their data. With more than 200,000 users, sciebo is one of the largest scientific sync-and-share platforms (funded by the state of North-Rhine-Westphalia) and is purpose-built for collaborative work in research groups. This large and widely adopted user base creates ideal conditions for the acceptance of lightweight RDM functions that help researchers manage their data easily, efficiently and FAIR[4].

Bridgit (the product name for the outcome of the sciebo RDS project, it refers to the bridge it creates between the sync and share system and research data repositories) offers user-friendly functions that focus on providing a smooth user experience (UX) for data preparation, annotation, and the seamless transfer of datasets to publication repositories or long-term archives. Bridgit is built on a modular architecture that strictly follows standards, formats and interfaces, making administration straightforward and allowing software developers to create their own RDS with minimal effort.

2 Architectural Overview

Bridgit follows a modular, component-based architecture designed for extensibility, maintainability, and seamless integration into institutional infrastructures. From a structural perspective, the system consists of a TypeScript-based frontend and a backend composed of multiple Python components, all containerized and deployed within Kubernetes environments.

Conceptually, bridgit acts as a middleware layer between institutional cloud storage environments and external research repositories, as illustrated in Figure 1. The data annotation is realized with an integrated metadata editor (see figure 2) while the connections to the research data repositories can also be configured within the interface (figure 3). Researchers typically manage their working data within institutional cloud systems such as sciebo. Bridgit integrates into this environment and provides additional research data management functionality, including dataset and file annotation, and the creation of data management plans. Through this intermediary role, bridgit enables projects, their associated metadata, and contained files to be transferred from institutional storage infrastructures to a variety of target repositories through a unified interface, allowing researchers to distribute their research data and associated metadata to multiple platforms from a single integrated environment.

The frontend is implemented using Vue 3 and PrimeVue 4 and communicates exclusively through a unified messaging protocol based on socket.io. This protocol is consistently implemented in both TypeScript and Python, ensuring uniform communication semantics across all system components. Instead of exposing a conventional REST API, bridgit adopts a request/response model augmented by event-driven updates. Clients send commands to the backend (e.g., user creation or metadata updates) and receive structured responses, while state changes can additionally be propagated asynchronously through event messages. This approach enables responsive user interaction while maintaining loose coupling between components.

[4]https://www.ub.uzh.ch/en/unterstuetzung-erhalten/tutorials/forschungsdaten-management/daten-fair-machen.html

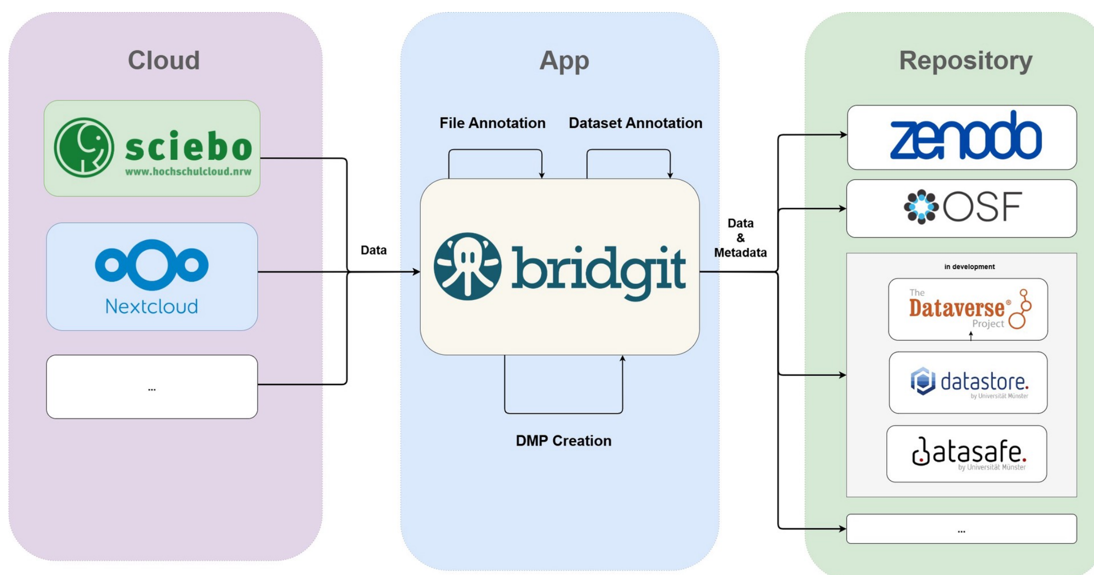

Figure 1: Architectural concept of bridgit. The system acts as middleware between a sync-and-share platform such as Nextcloud and external research data repositories.

At the core of the backend is the Server, which acts as the central orchestration and data management component. It receives and processes commands from the frontend, manages persistent data, and coordinates workflows by delegating specific tasks to other components. Among these are dedicated Connector components, which encapsulate interactions with external repositories like OSF and Zenodo. These were the first research data repositories to be made accessible via separate connectors. Others like Dataverse or University-internal services like the archiving service datastore are following. This separation isolates repository-specific logic and allows new target platforms to be integrated without modifying the core system.

A specialized auxiliary component, Domo, addresses a critical integration challenge related to external authorization flows. Since many third-party platforms permit only a single redirect URI, Domo acts as an intermediate redirection handler. It resolves the originating "home system" from an authorization state parameter and forwards the response accordingly. This design enables multi-tenant operation with minimal configuration per host.

All components are built as Docker containers and deployed using Kubernetes charts and Helm templates. The deployment architecture emphasizes low configuration overhead and reproducibility, supporting both reliability and maintainability in institutional cloud environments.

3 Main challenges

The development of bridgit was shaped by the heterogeneity of scientific disciplines and research use cases, which naturally leads to diverse research data management requirements. Different communities rely on distinct metadata standards, domain-specific conventions, and publication workflows. In addition, target repositories expose non-uniform APIs, authorization mechanisms, and data models, further increasing integration complexity.

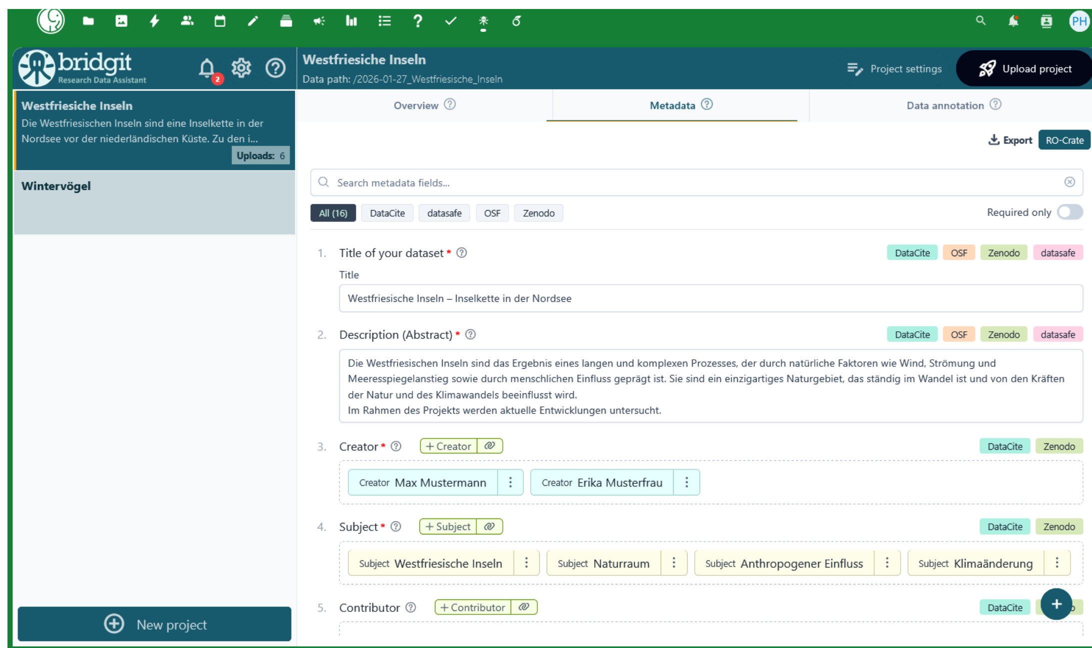

Figure 2: Bridgit's user interface in sciebo. On the left hand side, the user's projects are shown while on the right hand side, each project has a menu for the meta data, annotation and an optional data management plan. The app in inclueded in the sync & share system via a regular app. Here, the Metadata view is shown.

This diversity has direct architectural implications. Bridgit must remain extensible at multiple levels: new Connector components need to be implementable to support additional repositories, and metadata profiles must be adaptable to disciplinary requirements without modifying the system core. Repository-specific logic and metadata semantics therefore have to be clearly separated from central orchestration mechanisms.

At the same time, the platform is intended for institutional deployment and everyday use. A central challenge was balancing architectural flexibility with usability, maintainability, and operational simplicity. Enabling powerful configuration and extensibility while ensuring low-effort deployment in Kubernetes environments was essential to prevent flexibility from turning into administrative overhead.

4 Technical Details

This section outlines the technical foundations and architectural mechanisms of bridgit. It introduces the underlying technology stack, the structure of backend components and multi-tenant support, the handling of dynamic metadata profiles, the Connector abstraction for repository integration, the coordination of external authorization flows, the integration into host systems such as Nextcloud, and the containerized deployment model.

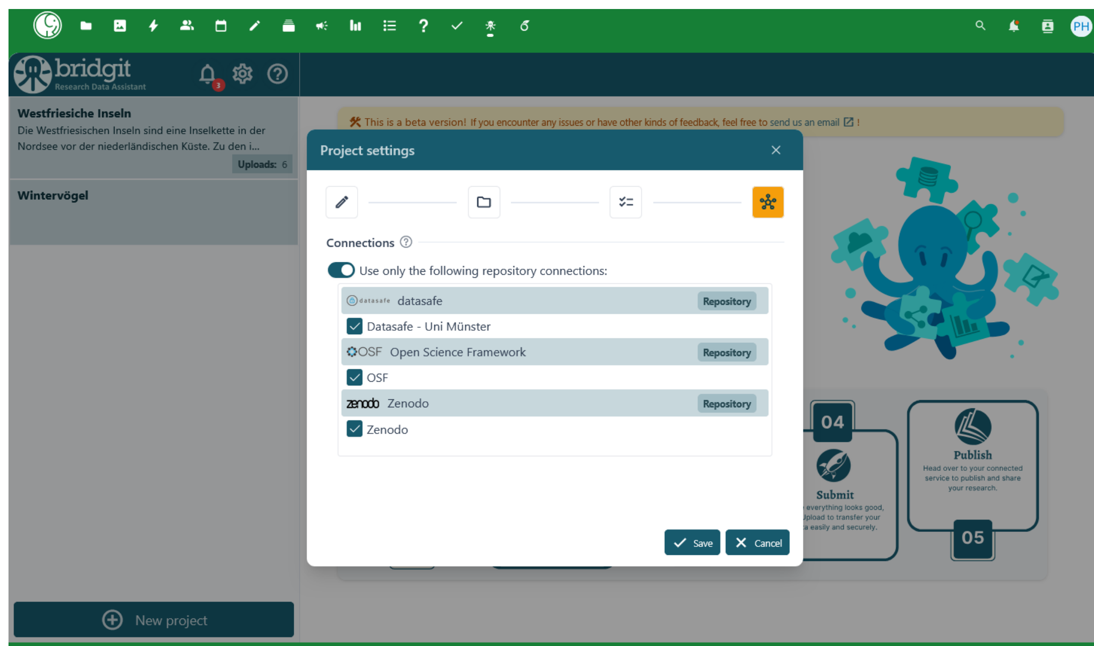

Figure 3: Connectors to research data repositories can be configured on a per project basis.

4.1 Key Technologies Used

Bridgit is built upon a modern technology stack selected with a focus on robustness, maintainability, and long-term sustainability. The frontend is implemented in TypeScript, providing static typing and improved tooling compared to plain JavaScript, thereby increasing code reliability and structural clarity. The user interface is realized using Vue[5] 3 in combination with PrimeVue[6] 4, enabling modular component design and a consistent, maintainable UI architecture.

For network communication, bridgit relies on socket.io[7] as its underlying transport mechanism. The decision against a traditional REST-based HTTP API was motivated by several architectural considerations. Socket.io provides bidirectional communication with automatic handling of reconnections, errors, and transport fallbacks. This eliminates the need for explicit HTTP error handling and simplifies client–server interaction logic. Furthermore, the platform regularly transfers large binary payloads, whose handling is considerably simplified through socket.io's unified communication model. On top of the transport layer, bridgit defines a structured messaging protocol to support command-based interactions and asynchronous event propagation. Due to its structured message format, the protocol can be extended with additional message types without altering the underlying transport mechanism, facilitating incremental feature development.

On the backend side, SQLAlchemy[8] is used as a database abstraction layer. This enables

[5] https://www.vuejs.org
[6] https://www.primevue.org
[7] https://socket.io
[8] https://www.sqlalchemy.org

support for different relational database systems without requiring changes to application logic, thereby allowing bridgit to integrate with database systems that are already established within institutional environments.

4.2 Backend Architecture and Multi-Tenant Design

All backend components share a common structural skeleton that provides essential functionality such as configuration management, message handling, and lifecycle control. This shared foundation accelerates development and ensures consistent behavior and functionality across components, ultimately reducing implementation effort and simplifying the understanding of the system as a whole.

The backend operates on an asynchronous request/response model augmented by event-driven updates. Components react to incoming commands — typically originating from the frontend or other components — while state changes and additional notifications can be propagated through events. This approach combines the predictability of request/response interactions with the flexibility of event-driven workflows, enabling responsive and robust system behavior due to its asynchronous nature.

Network communication is treated as a core architectural concern. Messages are highly structured, supporting both command execution and event propagation. The transport layer natively handles binary payloads, and the overall messaging design simplifies the management of complex communication patterns. Automated response handling, error propagation, and timeout management further reduce boilerplate and increase consistency in inter-component communication. Furthermore, the structured flow of messages facilitates debugging and tracing, making it easier to follow the sequence of operations across components.

Bridgit supports multi-tenant deployments, allowing a single installation to serve multiple host systems. Each tenant represents an independent host environment with its own configuration, including selected Connectors, metadata profiles, and other preferences. Tenants can be added or removed easily, enhancing system adaptability. To support correct operation in multi-tenant scenarios, a specialized auxiliary component, Domo, handles the routing of authorization requests to the appropriate tenant context. This mechanism ensures that authorization flows, which are discussed in detail in a later chapter, are correctly directed and associated with the intended host.

4.3 Metadata Model and Dynamic Profiles

A central capability of bridgit is the support for adaptable metadata profiles tailored to different disciplinary and project-specific requirements. Instead of embedding fixed metadata schemas directly into the application logic, metadata structures are defined through external JSON-based profile descriptions. This approach allows new profiles to be created or adjusted without requiring programming effort, as no modifications to the source code are necessary.

Within these profiles, the system supports a broad spectrum of metadata field types, ranging from scalar values such as numbers, strings, and dates to more complex and composite structures. These may include nested elements or grouped fields that reflect domain-specific requirements. Additionally, fields can be marked as mandatory, search providers for identifiers such as ORCID or GND can be configured, and custom descriptions can be added.

Metadata profiles exist at two distinct levels: project-level metadata describing the overall

research project, and object-level metadata assigned to individual files or folders. These profile types are defined and managed independently, allowing different structures and requirements at the project and object level.

On the user-facing side, the frontend dynamically renders metadata forms based on the selected profiles. Users can choose from the set of profiles that are available within their tenant context and appropriate for their project. Once selected, the corresponding metadata structure determines the form layout. This design enables the system to accommodate heterogeneous metadata requirements while maintaining a consistent user interaction model.

4.4 Connectors and External Repository Integration

Bridgit enables users to transfer their research projects to external target systems directly from within the application. These targets include heterogeneous platforms for scientific dissemination as well as simpler long-term storage systems. From the user's perspective, the upload process is transparent and integrated into the bridgit workflow: projects, including their associated metadata and contained files, can be transferred to a selected target system with minimal user effort.

Technically, this functionality is realized through dedicated Connector components. The Server delegates upload-related commands to the appropriate Connector, which encapsulates all target-specific logic. A Connector is responsible for transforming the internal metadata representation into the format required by the target system, transferring files and adapted metadata via the respective target API, and handling error scenarios. In addition, Connectors regularly query the status of previously transferred projects to detect existing records and prevent unintended duplicate uploads.

To preserve extensibility, Connectors can be implemented independently and integrated into an existing deployment with minimal effort. Bridgit provides a Python-based Connector framework that supplies the necessary structural foundation, including configuration handling and communication interfaces. Implementing a new Connector primarily involves mapping the target system's API and data model to bridgit's internal structures. This approach enables developers to implement additional Connectors for target systems that are not supported out of the box with comparatively low development effort.

Target systems often require elevated authorization mechanisms. Some platforms rely on simple API tokens, while others employ more complex flows such as OAuth2. Authorization is initiated through the frontend but executed and coordinated by the Server. The Connector itself receives the required authorization credentials or tokens from the Server once the process has been completed. The underlying authorization flow, which involves additional routing and redirect handling, is described in detail in the following section.

4.5 Authorization Handling

Bridgit distinguishes between two authorization domains: (1) access to the host system and (2) access to external target systems via Connectors.

In both cases, authorization is initiated by the user through the frontend. The user must grant bridgit permission to access the host system in order to retrieve project files and related resources. Likewise, when transferring a project to an external target system, the user must authorize bridgit to act on their behalf within that system. Although credentials are required

by backend components, the authorization process itself always originates from the user-facing interface.

Authorization credentials are stored in a tenant- and user-specific manner. Since users operate within a defined tenant context, all issued tokens are associated with both the corresponding tenant and the authenticated user.

Currently, two authorization mechanisms are supported, depending on the requirements of the respective target system.

API Token-Based Authorization Some systems rely on static API tokens. In this case, the user manually provides the token via the frontend interface. The Server stores the token and includes it in subsequent API requests issued by the corresponding component. This mechanism is comparatively straightforward, as it does not require redirect handling or multi-step negotiation.

OAuth2-Based Authorization More complex target systems rely on OAuth2-based authorization. This process involves coordinated interactions between the frontend, the bridgit Server, the authorization server of the target system, and the auxiliary routing component Domo.

A central architectural challenge results from the redirect-based nature of OAuth2. Since only a single redirect URI can typically be registered with the target system, but bridgit operates in a multi-tenant environment, the authorization response must be routed back to the correct tenant and user context.

To address this, bridgit encodes the necessary contextual information within the OAuth2 `state` parameter. This parameter contains sufficient information to re-establish the association with the initiating user and tenant after the redirect.

The OAuth2 flow in bridgit proceeds as follows:

1. The user initiates authorization via the frontend.
2. The Server generates the authorization URL and embeds tenant and user context into the `state` parameter.
3. The user authenticates at the target system.
4. The target system redirects to a fixed redirect URI handled by Domo, including the authorization code and the original `state` parameter.
5. Domo evaluates the returned `state` parameter and forwards the authorization response to the appropriate tenant context.
6. The Server exchanges the authorization code for an access token and, if provided, a refresh token.
7. The obtained tokens are stored in a tenant- and user-specific manner and managed throughout their lifecycle.

4.5.1 Token Management and Lifecycle

For API token-based integrations, the stored token is used directly for subsequent API calls. For OAuth2-based integrations, the Server monitors token validity and performs refresh operations when required. If a token expires and cannot be refreshed, it is automatically removed. In such cases, the user must repeat the authorization process to restore access. This design centralizes credential management while maintaining strict user-initiated authorization and tenant-aware routing within a multi-tenant architecture.

4.6 Host System Integration

Bridgit is designed to allow researchers to access research data management capabilities directly from their familiar working environment. By coupling the platform to a host system, users can manage and annotate projects without migrating or duplicating their existing files.

For each supported host system (currently only Nextcloud-based platforms such as sciebo), a lightweight integration app is provided. This app primarily embeds bridgit as an iframe within the host interface and handles essential internal tasks, such as passing required user information (e.g., user ID) to the bridgit Server. Multi-tenant support is an integral part of this integration, ensuring that the Server can correctly associate requests and data with the appropriate tenant context. For host systems not yet supported, similar integration apps can be implemented with minimal effort, highlighting bridgit's adaptable and flexible design.

To efficiently transfer project files from the host system to external target systems, bridgit employs a WebDAV-based approach. Instead of routing large files through the Server, which would be resource-intensive, the Server provides each Connector with the necessary access information. The Connector then downloads the files directly from the host system via WebDAV and uploads them to the selected target system. This approach, which is illustrated in Figure 4, minimizes overall data movement, reduces network overhead, and enables the handling of very large datasets efficiently.

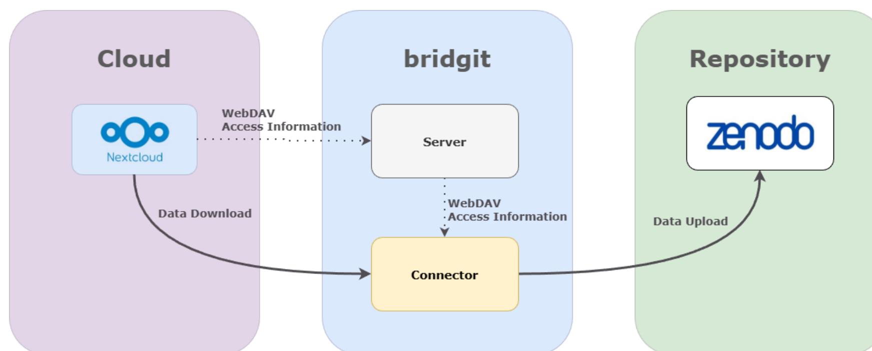

Figure 4: Overview of the WebDAV-based data transfer architecture. Data is transferred directly from host storage systems to the repository through the connector, while the application server is not involved in the data transfer.

Overall, host system integration ensures that researchers can leverage their existing storage infrastructure while gaining structured research data management functionalities, lowering barriers to adoption and simplifying day-to-day workflows.

4.7 Deployment in Docker and Kubernetes

Bridgit is fully containerized, with each component packaged as a Docker container. This container-based approach simplifies installation, ensures reproducibility, and isolates dependencies, allowing different components to be updated independently.

Deployment is supported through Kubernetes using pre-defined Helm charts that provide low-configuration installation for institutional environments. The charts define deployments and services, enabling bridgit to run reliably in multi-tenant scenarios with minimal administrative effort.

By default, bridgit supports multiple storage backends, but persistent storage configuration is left to administrators. This approach allows integration with pre-existing database systems or storage infrastructures typically available at research institutions, without imposing additional assumptions on the environment.

Overall, containerized deployment and Kubernetes orchestration enable bridgit to be installed and maintained efficiently within existing cloud infrastructures, ensuring robustness, reproducibility, and maintainability.

5 Future outlook

- Develop additional connectors for linking to other repositories to enable further use.
- Obtain more feedback from researchers, FDM teams, administrators, and software developers to determine the acceptance and demand for the developed solution and, if appropriate, initiate structured further development.
- Sharing of bridgit projects; comes with a long list of other required features, like true user management (including individual rights and their proper enforcement), prevention of race conditions (i.e., conflicting edits), and more
- Live collaboration is yet another planned feature, tightly coupled to sharing of projects; requires new strategies to reliably and swiftly share a state across participating users ("traditional" message transports usually are not the best approach here)
- Investigate the potential for integrating the Coscine Metadata Profile Service[9].
- More dynamism: Metadata profiles, connectors, and many other aspects still need to be made even more dynamic to properly serve the many different use cases; this especially involves connectors being able to query remote data to adjust dynamically to the target system (e.g., query and convert metadata profiles)
- A visual creator for metadata profiles; right now, their JSON files have to be written manually
- Certain AI features should be looked into; extracting a project description and metadata from project objects (files) are two such examples
- Quality of Life features: Things like searching for projects, tagging and grouping of files are just a few examples

[9] https://coscine.rwth-aachen.de/coscine/apps/aimsfrontend//

References

[1] Raimund Vogl, Anne Thoring, Dominik Rudolph, Holger Angenent, Jürgen Hölters, and Markus Blank-Burian. The sciebo.rds project: Who says research data management has to be complicated? [online], 2019. Available at https://www.eunis.org/download/2019/EUNIS_2019_paper_12.pdf.

[2] Mark D. Wilkinson, Michel Dumontier, and IJsbrand Jan Aalbersberg. The fair guiding principles for scientific data management and stewardship. *Scientific Data*, 3:160018, 2016.