FACULTY OF APPLIED SCIENCES
UNIVERSITY
OF WEST BOHEMIA

DEPARTMENT OF
COMPUTER SCIENCE
AND ENGINEERING

Software Architecture

Jakub Pavlíček & Štěpán Faragula

PILSEN, CZECH REPUBLIC

10. 5. 2025

Contents

Introduction 1

1.1 Purpose

This document presents a comprehensive overview of the software architecture for a pump control and Application Lifecycle Management (ALM) data management system. The architecture is designed to provide a robust, scalable and user-friendly solution for managing pump operations, such as extracting structured project data and integrating it into the SPADe system for further anti-pattern analysis.

1.2 System Overview

The pump system is a complex software solution that is responsible for:

- Extracting data from various ALM tools (Git, GitHub, Jira, etc.).
- Mapping and storing extracted data into a provided MySQL database.
- Providing a web-based user interface to configure pump behavior.
- Enabling real-time notifications of the pumping process via WebSockets.
- Ensuring easy deployment across different environments using containerization.

1.3 Primary Objectives

The primary objectives of the systems include:

- Reliable pump operation, supporting multiple ALM tools with seamless integration.
- Compatibility with existing ALM platforms and full integration with the SPADe data model.
- Scalability and modularity, allowing easy extension to support additional ALM tools.
- Pseudonimized data storage, ensuring compliance with privacy and security requirements, such as NDA.
- User-friendly interface for easy pump configuration.
- Real-time messages for monitoring and processing extraction progress.
- Containerized deployment to ensure consistent deployment across various environments.

Architecture Overview 2

2.1 High-Level Architecture Overview

The frontend of the system is developed using the Next.js framework, utilizing a React application that offers a user-friendly interface written in Radix UI for interacting with the entire system. The user starts the data extraction by sending a request to the backend API, which responds with a HTTP status `202 Accepted`, indicating that the request was successfully received. The core component of the system is the backend Spring Boot API, which initiates the data extraction process from various ALM tools using the proper type of data pump. Once the data has been extracted, it is mapped to the MySQL database, either according to the default configuration or user-specified mapping. Since the whole pumping process is an asynchronous process, the backend also manages messages to the frontend with Spring Boot WebSockets, that notifies the application about the status of the extraction. On the client side, SockJS is used to receive these messages.

The architecture of the proposed system can be seen in the Figure 2.1. Note that the system consists of 3 different Docker containers, each of which can be deployed independently.

2.1.1 Technology Stack

The system is developed using modern technologies and is organized into a layered structure, where each layer serves a specific purpose within the ALM data extraction workflow. This section provides an overview of the technologies used and the purpose of each component.

User Interface Layer (Frontend)

Technology: Next.js, 14.1.0 + React, 18.2.0 + Radix UI, 1.3.4

Purpose:

- Provide a user-friendly interface for initiating ALM data extraction.
- Customize mapping of the collected data.
- Display live updates and feedback from the backend.
- Communicate with backend services.

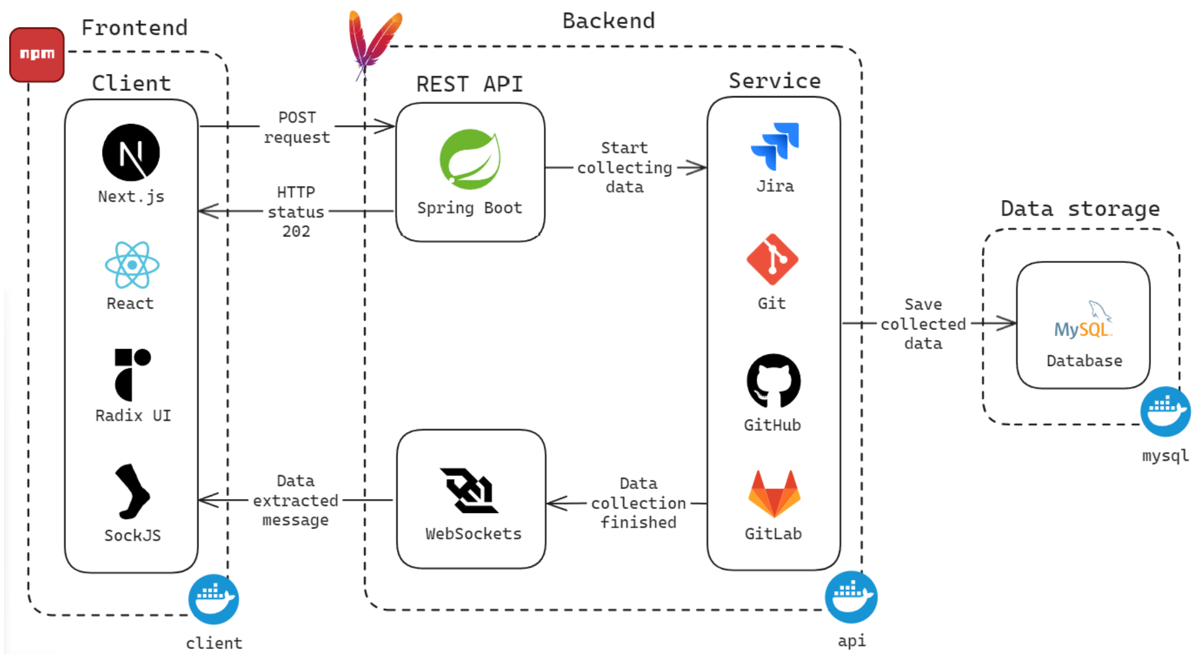

Figure 2.1: High-Level Architecture Diagram

Communication Channels:

- REST API for handling requests.
- Spring Boot WebSockets and SockJS for real-time messaging.

API Layer (Backend)

Technology: Java, 23.0.2 + Maven, 3.9.9 + Spring Boot, 3.4.4

Purpose:

- Act as the interface between the frontend application and backend services.
- Validate and process incoming requests from the frontend.
- Route valid requests to the appropriate pump services for data extraction.
- Handle invalid requests through proper error management.
- Return appropriate HTTP responses to the client.

Service Layer (Backend)

Technology: Java, 23.0.2 + Maven, 3.9.9 + Spring Boot, 3.4.4

Purpose:

- Implement core business logic for pump control operations.

- Process data extraction, user-defined mapping and pseudonimization.
- Ensure secure, transactional data handling.
- Dispatch real-time system messages.
- Resolve errors during data collection.

Database Layer (Data storage)

Technology: MySQL, 9.2.0

Purpose:

- Provide persistent storage for the collected ALM data across multiple projects.
- Store software project details, development lifecycle information and project members.

Messaging (Frontend + Backend)

Technology:

- **Frontend** = SockJS, 1.6.1
- **Backend** = WebSockets from Spring Boot, 3.4.4

Purpose:

- Real-time pump status updates from backend to the frontend.
- Asynchronous error notifications.

Containerization

Technology: Docker, 27.2.0 + Docker Compose

Purpose:

- Ensures consistent environments across development, test and production.
- Simplifies deployment and dependency management.
- Isolates components for better troubleshooting.
- Enables easy scaling of services.
- Configures network.

2.2 Service Layer Overview

The architecture is designed to separate generic data handling logic from tool-specific implementation details. It organizes the system into two main parts: the `Core` package, which contains reusable components and the `SpecificPump` package, which holds implementations of specific ALM extraction tools. This enhances maintainability and makes it easier to add support for new tools by reusing the core service infrastructure.

Figure 2.2 illustrates that the `SpecificPump` class depends on its associated `SpecificPumpService`, which has access to all four core services. When an operation is required, such as saving user data, the `SpecificPumpService` passes this task to the relevant core service (`PumpUserService`). That core service then forwards the request to a dedicated service that operates over a specific repository, for example `PersonService` operating over `PersonRepository`.

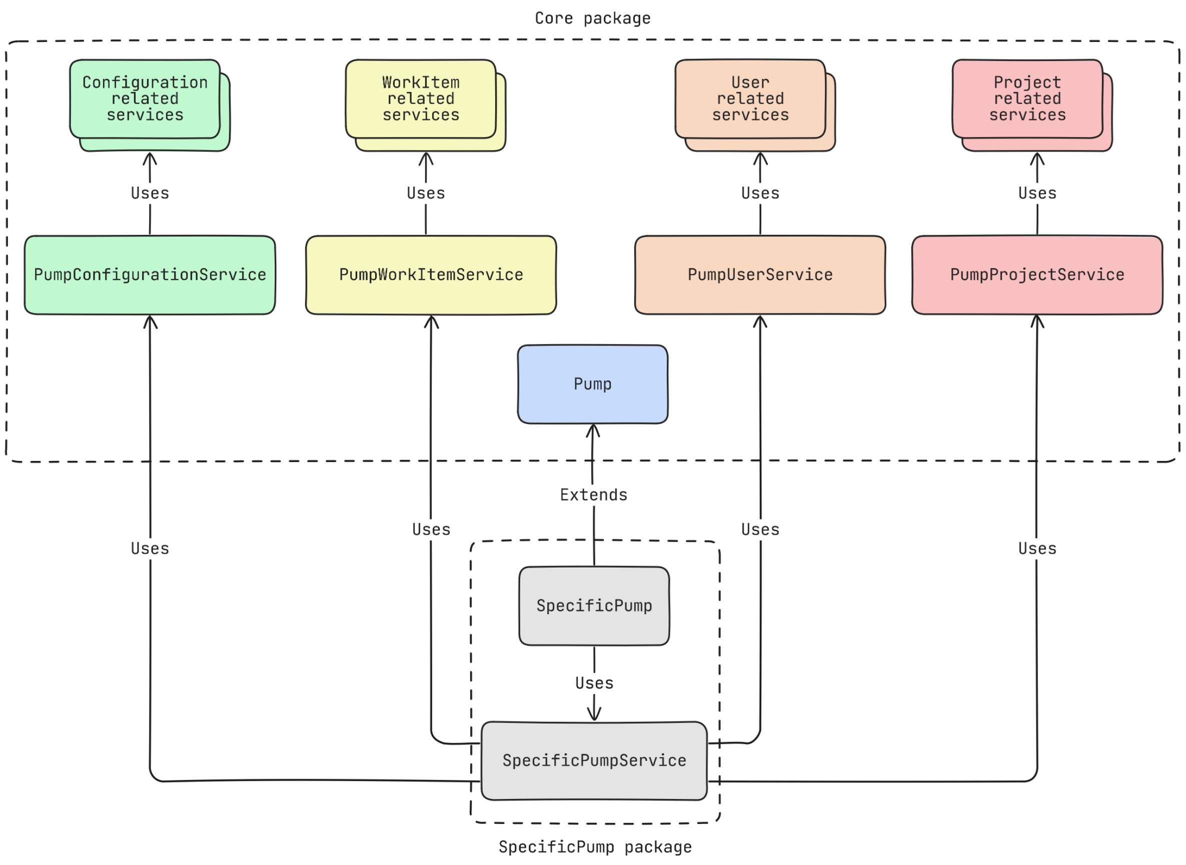

Figure 2.2: Service Structure Diagram

2.2.1 Core Package

The Core package contains the fundamental building blocks of the service layer, including:

- The abstract `Pump` class, which defines the template for all specific pump implementations.
- A set of entity-specific services, providing unified functionality for handling different types of ALM data.

- Four core pump services utilizing its corresponding entity-related services. These services contain reusable business logic for processing and storing data entities independent of their ALM source. These core services are namely:
  - `PumpConfigurationService`
  - `PumpWorkItemService`
  - `PumpUserService`
  - `PumpProjectService`

2.2.2 Specific Pump Package

This represents a specific implementation for an individual ALM data pump. While the implementation details may differ between the tools, every pump should include the following components:

- A `SpecificPump` class that extends the abstract `Pump` class from the Core package, implementing all the required methods for the data extraction lifecycle specific to that tool.
- A dedicated `SpecificPumpService` class encapsulating the business logic of this particular pump.

2.2.3 List of Related Services

As previously mentioned, the Core package relies on four core services, each of which utilizes its respective related services. These services are organized according to the entity relationships in the SPADe database and the goal is to create a new service for every entity within the database that operates under a single repository. For example, the `CommitService` should be the only service operating under the `CommitRepository` and any operations requiring multiple repositories should be implemented in the core service `PumpConfigurationService`. Below is a list of related entities (services) according to their relationships in the SPADe database.

PumpConfigurationService

- Branch
- Commit
- CommittedConfiguration
- Configuration
- ConfigurationBranch
- ConfigurationChange
- ConfigurationPerson Relation
- Tag

PumpWorkItemService

- Artifact
- Category
- FieldChange
- Priority
- PriorityClassification
- Relation
- RelationClassification
- Resolution
- ResolutionClassification

- Severity
- SeverityClassification
- Status
- StatusClassification
- WorkItem
- WorkItemChange
- WorkItemRelation
- WorkUnit
- WorkUnitCategory
- WuType
- WuTypeClassification

PumpUserService

- Competency
- GroupMember
- Identity
- PeopleGroup
- Person
- PersonCompetency
- PersonRole
- Role
- RoleClassification

PumpProjectService

- Activity
- Criterion
- Iteration
- Milestone
- MilestoneCriterion
- Phase
- Program
- Project
- ProjectInstance
- ToolInstance

ALM Data Pump Implementation

3

3.1 Package-Level Overview

The component diagram in Figure 3.1 illustrates the high-level package structure of the backend system architecture. It showcases the modular design employed to handle data extraction from different ALM tools. The architecture is divided into several main components, each represented as a package:

- **CORE:** This central package encapsulates the core functionalities and shared components of the system. It includes sub-packages for handling common concerns such as configuration (config), data mapping (mapper), API request handling (controller), data representation (entity), data persistence (repository) and core business logic (service). The service sub-package itself contains components for managing specific group of services like configuration, work_item, user and project.
- **JIRA:** This package contains the specific logic required to interact with the Jira ALM tool. It includes its own config, mapper, parser and service components tailored for Jira data extraction. This package utilizes the functionalities provided by the CORE package.
- **GITHUB:** This component handles interactions with the GitHub platform. It contains config, mapper, parser, service and domain specific to GitHub and it also depends on the CORE package.
- **GIT:** This package is responsible for extracting data directly from Git repositories. It contains config, mapper and service components for this purpose and relies on the shared CORE package.

This modular structure, where tool-specific packages (Jira, GitHub, Git) depend on a central CORE package, promotes code reusability and separation of concerns. It allows for easier maintenance and extension, as adding support for a new ALM tool would primarily involve creating a new tool-specific package that utilizes the existing CORE functionalities.

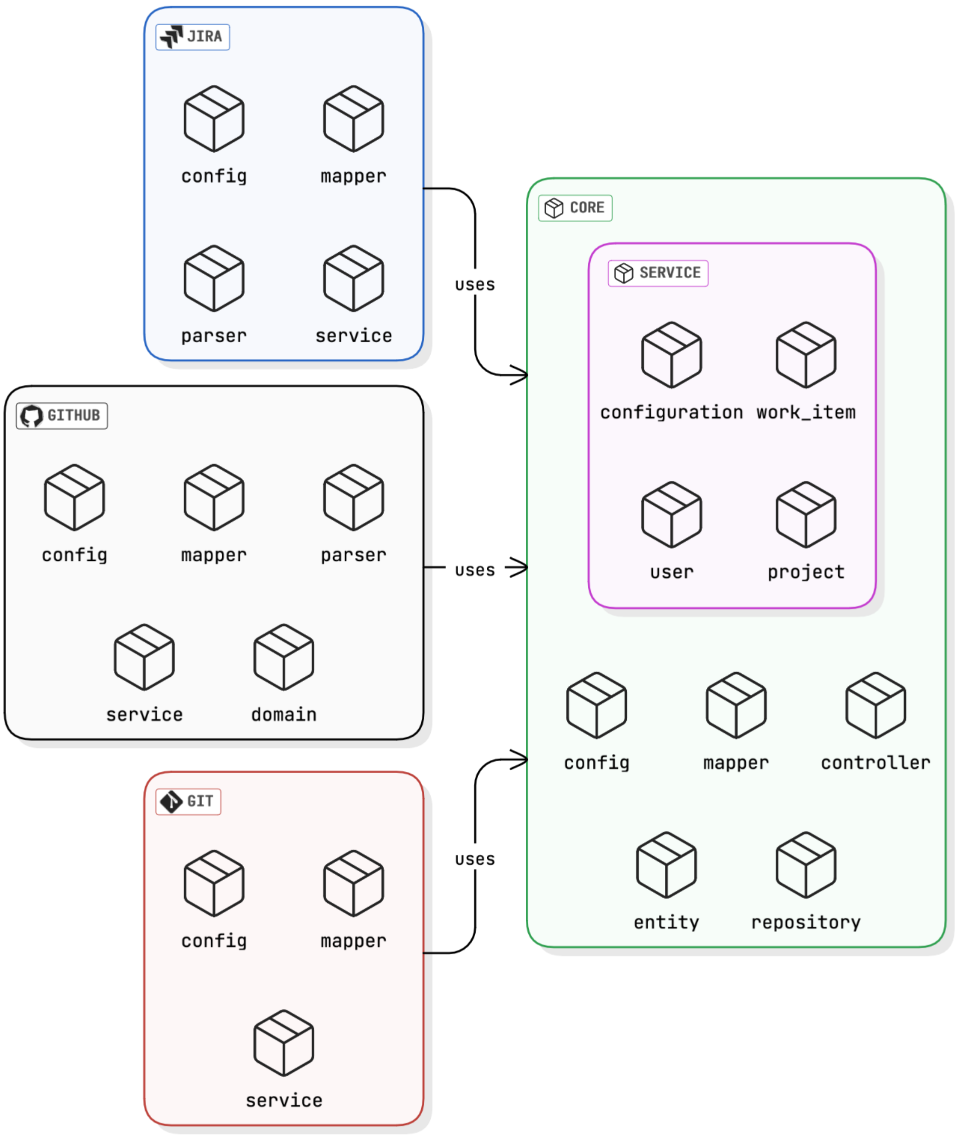

Figure 3.1: Package Structure Diagram

3.2 Class-Level Overview

The class diagram presented in Figure 3.2 outlines the core structure for the data pumping mechanism, emphasizing the strategy pattern used to handle different ALM data sources.

At the heart of the design is the abstract class Pump. This class defines a common template and interface for all specific pump implementations. It declares protected properties (properties) and defines the essential steps involved in the data pumping process through methods like `collectAlmData()`, `connectToProvider()`, `initializeProject()`, `collectDetailedData()`, `postProcessData()` and `cleanup()`. These methods represent the standardized lifecycle of any data extraction operation.

Concrete implementations are provided by subclasses that inherit from Pump:

- **JiraPump:** Specializes in extracting data from Jira. It holds a reference to a JiraRestClient and implements the abstract methods defined in Pump according to Jira's specific API and data structure.
- **GitPump:** Handles data extraction from Git repositories. It contains a reference to a Git object and provides the Git-specific implementation for the pumping steps.
- **GitHubPump:** Focuses on extracting data from GitHub. It uses a GitHub object reference and implements the necessary logic for interacting with the GitHub platform.

Each concrete pump class (JiraPump, GitPump, GitHubPump) is associated with multiple service classes (JiraServices, GitServices, GitHubServices). These service classes encapsulate the lower-level details of interacting with the respective ALM tool's API or data source, holding necessary clients or repositories (e.g., JiraRestClient, Git, GHRepository). Furthermore, these specific services cooperate with a general service classes (PumpServices).

This design promotes modularity and extensibility. By defining a common abstract Pump and associated service structure, the system can easily incorporate support for new ALM tools in the future by simply creating new subclasses inheriting from Pump and implementing their corresponding service logic, without altering the core pumping workflow.

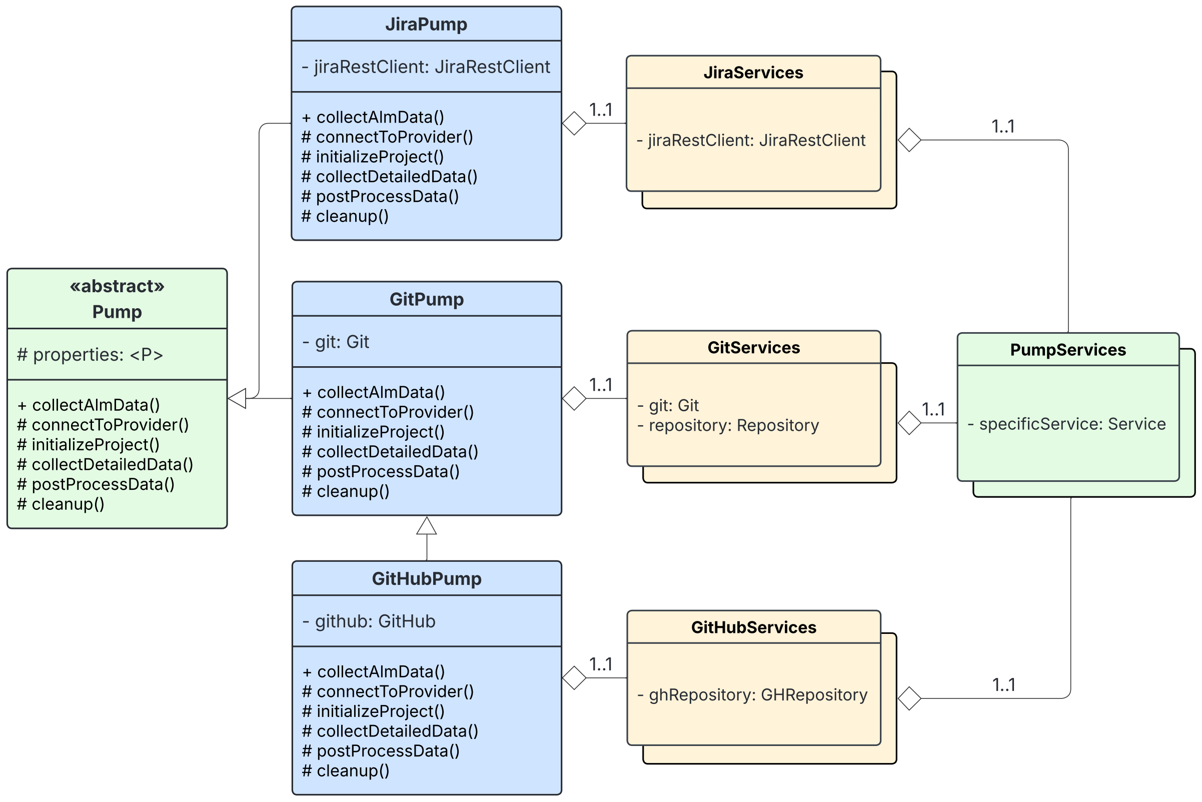

Figure 3.2: Class Diagram

API 4

This chapter covers the RESTful API provided by the backend system. The API serves as the primary communication channel for the frontend application to interact with the data pump and project management functionalities. The API is defined using the OpenAPI 3.1.0 Specification, ensuring a standardized and well-documented interface.

4.1 Endpoints

4.1.1 Pumps

Operations related to initiating data extraction processes.

POST /pumps/{tool}

- Initiates an asynchronous data extraction process (pump) for a specified ALM tool.
- The tool is identified by the tool path parameter, which can be either of `jira`, `git` or `github`.

4.1.2 Projects

Operations for managing projects within the SPADe system.

GET /projects

- Retrieves a list of projects currently stored in the SPADe system. Supports pagination via optional page and size query parameters.

DELETE /projects/{projectId}

- Initiates an asynchronous process to delete a specific project, identified by projectId in the path, along with all its associated data from the system.

4.1.3 Tools

Operations related to querying supported ALM tools.

GET /tools

- Retrieves a list of all ALM tools currently supported by the data pump system.

4.2 API Design Best Practices

The API follows industry-standard design patterns and conventions, including:

- **Data Transfer Objects (DTOs):** Structured schema definitions ensure consistent resource representation across endpoints while abstracting internal data models.
- **Pagination:** Supports standardized pagination for large datasets, including metadata such as page number, size, total records and sorting criteria to enable efficient data retrieval.
- **Error Handling:** Errors are reported using the ProblemDetail schema, aligning with RFC 7807 for standardized HTTP problem details. This includes fields like type, title, status, detail and instance.

List of Figures