Design and Implementation of a CHAOSS-Based Health Assessment System for Open Source Components in SCA Tool

BACHELOR THESIS

Dominik Hoffmann

Submitted on 12 March 2026

Friedrich-Alexander-Universität Erlangen-Nürnberg
Faculty of Engineering, Department Computer Science
Professorship for Open Source Software

Supervisors:
Martin Wagner, M.Sc
Prof. Dr. Dirk Riehle, M.B.A.

Friedrich-Alexander-Universität
Faculty of Engineering

Declaration of Originality

I confirm that the submitted thesis is original work and was written by me. Appropriate credit has been given where reference has been made to the work of others. The thesis was not examined before, nor has it been published. The submitted electronic version of the thesis matches the printed version.

Artificial Intelligence (AI) tools ChatGPT (OpenAI) were used for language editing, source summarization, and code development assistance. All AI-generated content was reviewed, verified, and edited by the author. The ideas, analysis, conclusions, and final interpretations presented in this thesis remain entirely my own work.

Erlangen, 12 March 2026

License

Erlangen, 12 March 2026

Abstract

Open Source Software (OSS) and the communities that maintain it play an everlasting role in the field of software development. The Linux Foundation's *Census of Free and Open Source Software* states that 96% of codebases use Free and Open Source Software (Nagle et al., 2024). As the number of open source software (OSS) components used in a project increases, the project becomes increasingly dependent on the communities that maintain them. This raises questions about community health and sustainability.

This thesis aims to develop an approach for assessing and potentially rating the health of Open Source Projects using standardized metrics introduced by the Community Health Analytics in Open Source Software (CHAOSS) project, and to integrate the results into the Software Composition Analysis tool SCA Tool.

Contents

List of Figures

Acronyms

OSS Open Source Software

CHAOSS Community Health Analytics Open Source Software

UI User Interface

URL Uniform Resource Locator

REST Representational State Transfer

API Application Programming Interface

SQL Structured Query Language

AMQP Advanced Message Queuing Protocol

SBOM Software Bill of Materials

PURL Package URL

1 Introduction

Open-source software has become a fundamental building block of modern software systems. Organizations increasingly depend on numerous external components that are integrated through package managers and distributed development ecosystems. According to the Linux Foundation's *Census of Free and Open Source Software*, 96% of codebases incorporate free and open-source software (Nagle et al., 2024).

While Software Composition Analysis (SCA) tools provide valuable insights into licensing compliance and security vulnerabilities associated with these dependencies, they typically offer limited support for assessing the long-term sustainability and health of the underlying open-source projects.

However, the long-term viability of a software component does not only depend on its security posture or licensing conditions, but also on the activity, responsiveness, and stability of the community maintaining it. Projects with declining contributor activity, slow response times, or concentrated development effort may introduce long-term risks for organizations that depend on them.

To address this gap, this thesis explores the integration of open-source project health assessment into SCA Tool. The goal is to design and implement a system that collects and evaluates health-related metrics for open-source dependencies and presents them in a form that supports informed decision-making for software maintainers and developers. For this purpose, a dedicated microservice, referred to in the following as *Health Assessor*, was developed, which collects and processes selected metrics defined by the Community Health Analytics Open Source Software (CHAOSS) project and integrates the resulting insights into the SCA Tool user interface.

The remainder of this thesis is structured as follows. Chapter 2 provides a literature review covering existing research on open-source project health and related tools. Chapter 3 defines the functional and non-functional requirements for the proposed Health Assessment System. Chapter 4 presents the architecture of the Health Assessor and its integration into the SCA Tool ecosystem. Chapter 5 describes the design and implementation of the system. Chapter 6 evaluates

the implemented solution with respect to the previously defined requirements. Chapter 7 outlines potential directions for future work, and Chapter 8 concludes the thesis.

2 Literature Review

This chapter provides a detailed overview of Open Source Project Health, including its definition and key components. In addition, it introduces the Community Health Analytics Open Source Software (CHAOSS) project and its standardized health metrics. Further, existing tools for retrieving and analyzing these metrics are examined. Based on SCA Tool's use case, the metrics most relevant to this context are identified and highlighted.

2.1 Open Source Project Health

Open-source software has become a critical component of modern software systems. As the number of open-source dependencies within contemporary code bases continues to grow, and as even security-critical systems increasingly rely on such components, it has become increasingly important for maintainers to assess not only the technical properties of these dependencies but also their long-term sustainability and the vitality of the communities that develop and maintain them (S. P. Goggins et al., 2021).

The concept of Open Source Project Health is generally understood in the literature as a multidimensional construct encompassing aspects such as sustainability, development activity, software quality, community viability, and the long-term survivability of a project. These dimensions typically include socio-technical factors—such as contributor participation, development activity, and project leadership—as well as broader contextual elements, including the surrounding technological ecosystem and the financial or organizational resources that support continued project development. However, despite increasing research interest in this area, no universally accepted definition of project health currently exists. Consequently, different studies adopt varying interpretations and methodological approaches when investigating the health and sustainability of open-source communities (S. P. Goggins et al., 2021).

2.2 CHAOSS - Community Health Analytics Open Source Software

The increasing awareness of the importance of open-source ecosystems led to the establishment of the Community Health Analytics in Open Source Software (CHAOSS) project[1] under the Linux Foundation in 2017. The primary objective of CHAOSS is to develop standardized metrics, conceptual models, and supporting tools that enable a systematic assessment of open-source community health (CHAOSS, 2025a). Within this framework, CHAOSS provides formal definitions, descriptions, and documentation of metrics, while the selection, interpretation, and contextual application of these metrics remain the responsibility of the user. As of November 2025, the CHAOSS framework defines and documents 89 standardized metrics.

To ensure conceptual clarity and avoid inconsistencies in terminology and metric definitions, the Health Assessor exclusively relies on metrics defined by CHAOSS. Their standardized structure provides a consistent and well-documented foundation for assessing project health within SCA Tool.

2.3 Metrics

Metrics provide valuable insights for stakeholders, project sponsors, maintainers, and users of an open-source project. According to CHAOSS, a *metric* is defined as a measurement designed to answer a specific question about the health of a project community, while *metric models* represent collections of related metrics that provide broader contextual understanding and enable the analysis of more complex aspects of project health (CHAOSS, 2025a).

Within the CHAOSS framework, metric models group metrics into thematic areas such as Diversity, Equity, and Inclusion (DEI) event badging, community activity, project safety, or funding sustainability. These models allow stakeholders to evaluate project health from multiple perspectives rather than relying on isolated indicators.

2.3.1 Metric Types

The assessment of open-source project health encompasses both quantifiable and less tangible dimensions. Certain metrics, such as *Committers*, *Issue Response Time*, or *Code Changes*, can be measured relatively easily because they are derived from publicly accessible development traces on platforms such as GitHub[2],

[1] https://chaoss.community/
[2] https://github.com/

GitLab[3], or project mailing lists (Dabbish et al., 2012).

In contrast, dimensions such as *Project Awareness* or *Community Welcomingness* are significantly more difficult to quantify, as the relevant information is often not publicly available or cannot be measured reliably using automated methods. For this reason, the Health Assessment within SCA Tool focuses primarily on objective and quantifiable metrics derived from publicly available sources, most notably GitHub repositories.

2.3.2 Metric Collection

For publicly hosted projects on GitHub, basic metrics can be viewed directly on the repository page; however, these provide only limited insight. To obtain more detailed information, the GitHub API[4] can be used to query project activities such as commits, issues, comments, and pull requests (Qiu et al., 2023). Programmatic use of the GitHub API, however, requires substantial effort, particularly when scanning all open-source software (OSS) repositories associated within a large codebase. To solve this problem, CHAOSS is actively involved in the development of dedicated metric collection software, which reduces the complexity of gathering and analyzing such data.

2.3.3 CHAOSS Software

CHAOSS provides two different software tools to derive health insights of OSS communities, Augur[5] and GrimoireLab[6] (CHAOSS, 2025b). Both of these tools are open source and take different approaches.

GrimoireLab

GrimoireLab focuses on integrating multiple streams of input data and emphasizes a visual analysis approach by providing a built-in dashboard for exploring project metrics. However, it offers limited support for customizable APIs, which makes it more accessible for non-technical users but less suitable as a data source for an integrated health assessment system such as the one targeted for SCA Tool.

Augur

In contrast, Augur focuses specifically on analyzing repository traces from platforms such as GitHub and GitLab (Colt, 2023) while also providing extensive

[3]https://gitlab.com/
[4]https://docs.github.com/en/rest
[5]https://github.com/chaoss/augur
[6]https://chaoss.github.io/grimoirelab/

API support. Overall, Augur aligns well with the requirements of the SCA Tool use case, as it shares several core technologies with the existing system architecture. Augur uses *Redis*[7] as a key–value store, *RabbitMQ*[8] for messaging, and *PostgreSQL*[9] as its primary database system—technologies that are likewise used within SCA Tool.

Furthermore, Augur supports containerized deployments through Docker, which allows the Health Assessor to reuse already running infrastructure components such as Redis, RabbitMQ, and PostgreSQL without requiring additional instances of these resource-intensive services.

Consistent with the needs of SCA Tool, Augur allows the submission of repository URLs for analysis. It then collects repository traces and calculates a range of metrics, which are stored in the PostgreSQL schema *augur*. These metrics can subsequently be retrieved either through direct SQL queries or via the built-in REST API[10]. However, Augur does not provide all potentially relevant quantitative repository metrics directly; some metrics must be derived from the collected raw data. Additionally, Augur primarily relies on data originating from GitHub and GitLab and does not natively integrate additional external data sources.

In summary, Augur significantly reduces the effort required to collect and process repository trace data-tasks that would otherwise have required manual interaction with the GitHub API. At the same time, its architecture provides suitable integration capabilities for incorporation into the SCA Tool ecosystem. For these reasons, Augur was selected as the primary data source for health metrics within the Health Assessment system.

2.3.4 Relevant Metrics

Given the 89 available CHAOSS metrics, identifying metrics that are broadly applicable for assessing the health of most open source projects remains challenging. However, to provision a good baseline, CHAOSS has a metric model called *Starter Project Health Metrics Model*[11] that combines four core metrics that can establish a good framework for assessing the health of an OSS community. These metrics will be detailed in the following.

Time To First Response

The *Time to First Response* describes the elapsed time between the creation of a contribution request (e.g., issue, pull request, or similar interaction) and the first

[7] https://redis.io/
[8] https://www.rabbitmq.com/
[9] https://www.postgresql.org/
[10] https://oss-augur.readthedocs.io/en/main/rest-api/api.html
[11] https://chaoss.community/kb/metrics-model-starter-project-health/

reaction by a project member. The metric reflects how responsive a project community is to incoming contributions or questions. Short response times generally indicate active engagement and ongoing maintenance, whereas delayed responses may signal limited availability of maintainers or lower community activity. As such, this metric provides insight into the project's level of responsiveness and its ability to support contributors (CHAOSS, 2024a).

Change Request Closure Ratio

The *Change Request Closure Ratio* measures the relationship between closed and open change requests (e.g., pull requests or merge requests) within a defined time frame. It serves as an indicator of whether a project is able to process incoming contributions in a timely manner. Higher values suggest that submitted changes are handled efficiently, whereas lower values may indicate limited maintenance capacity or an imbalance between incoming contributions and available maintainers (CHAOSS, 2024b).

Contributor Absence Factor

The Contributor Absence Factor, also referred to as the Bus Factor, quantifies how many contributors account for 50% of a project's total contributions within a specified time frame (CHAOSS, 2024c). It serves as an indicator of the project's resilience by highlighting potential concentration of contribution activity. A low value suggests strong dependency on a small number of individuals, which may pose continuity risks if these contributors become unavailable.

In practical implementations, the definition of "contribution" must be specified. Within the Health Assessor for SCA Tool, contributions are operationalized as Git commits, providing a measurable and reproducible basis for calculation.

Release Frequency

The *Release Frequency* describes how often a project publishes new versions of its software within a defined time period (CHAOSS, 2024d). It reflects the cadence at which updates, bug fixes, or feature enhancements are delivered. The interpretation of this metric must consider project context, as release strategies vary significantly across communities. Some projects publish smaller updates frequently, while others bundle changes into larger, less frequent releases.

2.4 Metric Interpretation

Following the examination of the CHAOSS project and its effort to standardize health metrics, it becomes evident that the existence of well-defined metrics

does not automatically translate into meaningful assessments of project health. Metrics can provide precise measurements, yet still fail to address the underlying question stakeholders are primarily concerned with: whether a project is sustainable and resilient. In this sense, reliance on isolated activity indicators risks producing what has been described as the "right answers to the wrong questions". A project may exhibit high commit frequency or growing contributor numbers, but such figures alone do not reveal whether the community structure is stable, governance mechanisms are effective, or long-term sustainability is secured (S. Goggins et al., 2021).

2.4.1 Context

The identical metric value may have different implications depending on the characteristics and context of a given project. Consequently, each metric must be interpreted in relation to the specific project environment and its individual circumstances (S. Goggins et al., 2021).

Lifecycle Context

On the one hand, the lifecycle of a project must always be taken into account, for example whether it is in an early stage of development, actively growing, already mature, or gradually declining. A temporarily low level of activity may indicate stagnation in a young project, while in a mature project it may simply reflect stability and a reduced need for frequent changes. Similarly, strong growth in contributors or commits can suggest positive momentum, but may also introduce coordination challenges (S. Goggins et al., 2021).

Ecosystem Context

The broader ecosystem of a package must also be considered when interpreting its health. This includes the domain in which it is used, the presence of competing solutions, the programming language or framework environment, and its dependencies on other libraries. Activity levels and growth patterns may vary significantly depending on these external factors (S. Goggins et al., 2021).

2.4.2 Interpretation Principles

Beyond contextual factors, the interpretation of metrics should follow a set of guiding principles. First, *comparison* is essential, as metrics gain meaning primarily when contrasted across projects or within a portfolio. Second, *trajectory* emphasizes that developments over time are more informative than isolated point-in-time values. Third, *transparency* requires clear traceability from raw data to

calculated metrics in order to ensure credibility and informed interpretation. Finally, *visualization and narrative* highlight that metrics are not an end in themselves but serve to support structured storytelling about project health rather than merely populating dashboards. (S. Goggins et al., 2021).

2.4.3 From Metrics to Health Narratives

Individual metrics reflect specific aspects of open-source projects, such as activity, contributor diversity, responsiveness, or risk. Although these indicators provide useful information, each of them captures only one part of the overall picture. A meaningful assessment of project health therefore requires combining several metrics and interpreting them in relation to their context.

Project health emerges through contextualized narratives that take comparison, development over time, and ecosystem factors into account. Tools such as Augur (see subsection 2.3.3) support this process by integrating standardized metrics into visual and comparative analyses, thereby enabling structured and evidence-based assessments of sustainability and resilience (S. Goggins et al., 2021).

2.5 OpenSSF Scorecard

Since several tools exist that assess aspects of open-source project quality and sustainability, alternative solutions were evaluated for potential integration into SCA Tool. One such candidate was OpenSSF Scorecard. Its suitability for the open-source project health use case within SCA Tool was therefore examined.

OpenSSF Scorecard primarily focuses on identifying security-related risks, such as the presence of known vulnerabilities, the use of automated testing, or the execution of static application security testing (SAST) (OpenSSF, n.d). While these aspects are valuable from a security perspective, they only partially overlap with the broader concept of open-source project health as defined in this thesis.

The most relevant intersection is Scorecard's "maintained" category, which provides a qualitative risk classification (high, medium, low) based on commit activity within the past 90 days(Schrock et al., 2026). However, this assessment relies solely on commit frequency and does not consider additional socio-technical factors as does CHAOSS (see section 2.3). As a result, it does not provide a sufficiently comprehensive basis for a holistic health evaluation of open-source components.

3 Requirements

This chapter outlines the system requirements relevant to the integration of a health assessment into SCA Tool. It first presents the motivation for introducing a health assessment in SCA Tool and subsequently distinguishes between functional and non-functional requirements.

3.1 Functional Requirements

This section provides an overview of the functional requirements for the Health Assessment System in SCA Tool. These requirements define the core capabilities the system must provide in order to collect, process, persist, and present open-source health metrics in alignment with the objectives of this thesis.

FR-01 The system shall evaluate SCA Tool–scanned packages using open-source health metrics defined by the CHAOSS foundation.

FR-02 The system shall provide health metrics for extracted packages that are associated with a valid and accessible Git repository.

FR-03 The system shall periodically update previously collected health metrics.

FR-04 The system shall persist collected health metrics in the SCA Tool database.

FR-05 The system shall use a suitable tool to collect open-source project health data.

FR-06 The system shall support bulk retrieval of health metrics for multiple packages in a single request.

FR-07 The system shall expose health metrics through a well-defined API that provides a stable interface for the SCA Tool frontend.

FR-08 The system shall allow users to mark individual packages as reviewed by the user within the user interface.

FR-09 The system shall provide users with detailed explanations of collected health metrics and guidance on their interpretation.

3.2 Non-Functional Requirements

This section outlines the non-functional requirements of the Health Assessment System in SCA Tool. These requirements define quality attributes and constraints related to usability, integration, maintainability, and performance that the system must satisfy beyond its core functional behavior.

NFR-01 The system shall be implemented as an independent microservice within the existing SCA Tool architecture.

NFR-02 The system shall reuse the existing backend infrastructure of SCA Tool to the greatest extent possible.

NFR-03 The system shall reuse the existing frontend infrastructure of SCA Tool to the greatest extent possible.

NFR-04 The system shall be integrated into the existing SCA Tool user interface.

NFR-05 The system's user interface shall be aligned with the existing visual design and styling conventions of SCA Tool.

NFR-06 The system shall present collected health metrics in an informative and intuitive manner within the user interface.

NFR-07 The system shall present assessed project health risks using a simple, intuitive color-coded scheme, indicating low risk in green, medium risk in yellow, and high risk in red.

NFR-08 The system shall avoid tight coupling between the Health Assessor and external data providers.

4 Architecture

This chapter presents an architectural overview of the health assessment system within SCA Tool. It begins by outlining the overall system-level workflow, followed by an introduction of the relevant components already present in SCA Tool. In succession the CHAOSS-based system Augur and its HTTP proxy sidecar are described. Finally, the architecture of the newly introduced Health Assessor microservice is presented.

4.1 End-to-End System Workflow

The workflow is initiated through the SCA Tool user interface (see subsection 4.2.7), where the user specifies a repository URL and triggers the analysis process. The Analyzer microservice (see subsection 4.2.2) subsequently performs the analysis and, upon completion, notifies the Health Assessor microservice (see subsection 4.3.3). The Health Assessor then retrieves the repository URLs associated with the packages used in the analyzed codebase from the SCA Tool database. These URLs are forwarded to the Augur service (see subsection 4.3.1) via the HTTP proxy sidecar(see subsection 4.3.2), where the collection and processing of raw data and metrics are initiated.

In parallel, the Health Assessor periodically retrieves all available health-related data relevant to the assessed repositories from Augur and performs additional metric calculations where required. The resulting metrics are persisted in the SCA Tool database (see subsection 4.2.3). Once the scanning process has completed, users can access the Health tab in the SCA Tool UI to obtain an overview of the health status of the individual packages within the analyzed codebase. The user interface retrieves the corresponding health data via the SCA Tool API (see subsection 4.2.6).

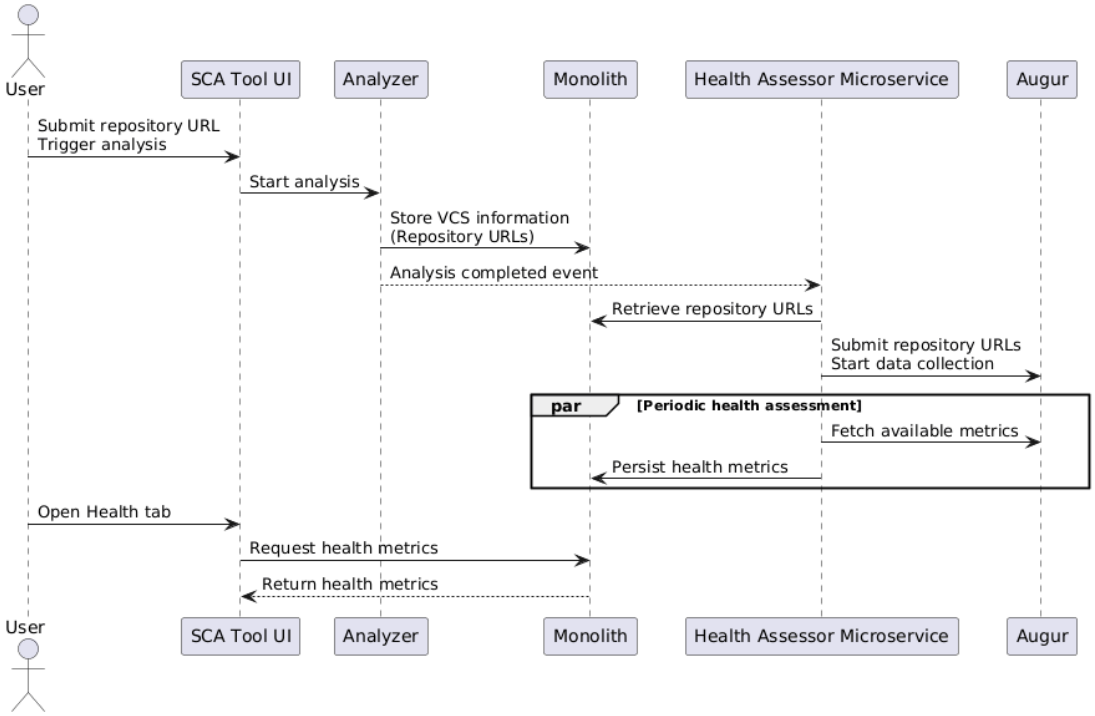

Figure 4.1: System-level sequence diagram of the health assessment workflow in SCA Tool

4.2 SCA Tool

SCA Tool's technology stack consists of a Java-based backend implemented with Spring Boot, a web-based user interface built with React, Next.js and TypeScript, a PostgreSQL database for persistent storage, RabbitMQ for message-based communication, and Redis as a key–value cache. The system is composed of multiple microservices, including an Analyzer responsible for analyzing submitted projects, a central service called Monolith that orchestrates interactions between microservices and manages database persistence, and a Scanner that performs the actual scanning of individual packages within a project.

4.2.1 Monolith

The Monolith serves as the central coordinating microservice of SCA Tool. It orchestrates communication between the Scanner, Analyzer, Health Assessor, and the user interface, and acts as the sole component responsible for accessing and managing the SCA Tool database.

4.2.2 Analyzer

The Analyzer is an independent microservice that processes analysis requests received from the UI. Analysis requests are received asynchronously via RabbitMQ (see subsection 4.2.4), enabling decoupled and scalable task execution.

For each request, the Analyzer launches an isolated workload, either as a Docker container (locally) or a Kubernetes job (deployed), to ensure strict separation between analyses. Within this environment, the target codebase is retrieved and analyzed, and the resulting artifacts are written to persistent storage using presigned URLs.

Upon completion of the analysis task, the Analyzer emits a completion event via RabbitMQ to notify the Monolith and Health Assessor that the analysis has finished, thereby triggering subsequent processing steps within the system.

4.2.3 Database

SCA Tool utilizes a PostgreSQL instance that hosts multiple database schemas. As part of the Health Assessor integration, an additional schema is introduced within this instance (see subsection 5.1.1). As described in subsection 4.2.2, the Analyzer persists the extracted analysis data in the SCA Tool database schema. Upon completion of the health assessment workflow, the Health Assessor further stores the calculated metrics and relevant raw data in the database, thereby extending the existing persistence layer.

4.2.4 Messaging and Caching

SCA Tool uses the messaging framework RabbitMQ for interactions between microservices. This allows events, statuses, and data to be exchanged between the various services. The Health Assessor requires a trigger, the event that the Analyzer has finished analyzing, to ensure that the repository URL is already in the database.

In addition SCA Tool already runs with the key-value storage Redis, which acts as a cache. The Health Assessor and required instances also use Redis.

4.2.5 Docker

SCA Tool is a container-based application that deploys all necessary instances, including the aforementioned PostgresSQL, RabbitMQ and Redis, in a Docker Compose stack. This Compose stack operates on a shared Docker network. The Health Assessor also uses Docker with this network to utilize the existing instances and avoid duplicate deployments.

4.2.6 API

SCA Tool provides an OpenAPI specification between the Monolith and the React-based user interface, which defines and enforces a clear API contract. This specification is also required for the integration of the Health Assessor. Based on the OpenAPI definition, interfaces, data structures, and related elements are generated, which must be implemented accordingly to ensure correct interaction and the intended system behavior.

4.2.7 Web Interface

SCA Tool's web interface is browser-based and uses the React JavaScript framework with Next.js. It provides the option to view scan results grouped into several tabs located in the side navigation. This thesis extends the existing interface by adding an additional tab that enables a more detailed exploration of open-source community health of the underlying project of a package.

4.3 Health Assessor Microservice

The Health Assessor was designed as a microservice due to its clearly defined responsibility, logical separation from SCA Tool, and reliance on an independent data source. Initially, the structure of the CHAOSS-based software Augur is considered, which functions as the central data source for the microservice. Thereafter, the architecture of the Health Assessor is discussed.

4.3.1 Augur

Augur is a software system that automatically analyzes and stores health metrics for git repositories. Providing a GitHub or GitLab repository URL is sufficient to initiate the analysis process.

Command Line Interface

At the time of writing, Augur does not provide an HTTP endpoint for submitting repositories. Instead, repositories are imported by executing a command via the command-line interface (CLI), which accepts a CSV file containing the repository URLs as input and registers them in Augur.

API

Augur exposes HTTP endpoints that provide access to information such as the list of repositories with their corresponding Augur identifiers, as well as selected metrics.

Database

All collected data related to the respective repositories, along with their associated metadata, is stored in Augur's internal database, which is integrated into SCA Tool as a dedicated schema. The Health Assessor additionally utilizes this raw data to compute its own metrics.

4.3.2 HTTP Proxy Sidecar

As discussed in subsection 4.3.1, Augur does not expose a direct HTTP endpoint for repository ingestion. Consequently, a sidecar-based approach was implemented for security reasons. This component accepts a list of repository URLs and forwards them to Augur internally by invoking the corresponding command-line interfaces.

4.3.3 Health Assessor Microservice

The Health Assessor microservice is responsible for calculating and aggregating health metrics for software repositories based on data provided by Augur.

The Health Assessor is implemented as a Spring Boot application in order to align with the existing technology stack. Once the Analyzer signals completion, the Health Assessor retrieves the package URLs from the SCA Tool database and forwards them to Augur for analysis via the HTTP proxy sidecar. Subsequently, metrics are either obtained periodically through Augur's HTTP endpoints or derived directly from raw data stored in Augur's database. The resulting metrics are then persisted in the SCA Tool database.

Database

The Health Assessor interacts with both the Augur and SCA Tool databases. Access to the Augur database is limited to read-only operations, whereas the Health Assessor persists computed metrics in the repositories and repository-metrics tables of the SCA Tool database. The Monolith subsequently retrieves these metrics from the database and exposes them through its API.

5 Design and Implementation

This chapter describes the modifications and extensions made to the SCA Tool codebase to enable the integration of the Health Assessor. All described changes are based on the state of the SCA Tool repository starting from commit `837dd83`.

The chapter begins with the deployment and configuration of the Augur service, which serves as the primary data source for repository health metrics. It then presents the design and implementation of the Health Assessor microservice. Subsequently, the required adaptations to the SCA Tool Monolith are described, followed by an overview of the changes made to the front-end components in order to integrate the health assessment functionality into the user interface.

5.1 Augur Service

As described in subsection 4.3.1, Augur aggregates metrics and persists them in a database, which necessitates its execution alongside the Health Assessor microservice. Accordingly, Augur is started automatically when the microservice is initialized, using a shell script. To enable the reuse of these shared instances, the Monolith must be running before the Augur instance is started.

5.1.1 Database

To enable the Augur service to reuse the existing PostgreSQL instance hosting the SCA Tool database, as described in subsection 4.3.1, a dedicated database schema for Augur must first be created using the SQL commands `CREATE DATABASE` and `CREATE USER`. The corresponding database user is granted full access rights to this schema via `GRANT ALL PRIVILEGES`. These initialization steps were integrated into an existing shell script that is executed during the startup of the Monolith.

5.1.2 Docker Compose

While Augur's source code includes a Docker Compose configuration, this file required substantial modification to enable the reuse of existing PostgreSQL, Rab-

bitMQ, and Redis instances. The required configuration parameters, including URLs, passwords, and authentication tokens, are supplied during local startup via a `.env` file and incorporated through the customized configuration. In addition, the HTTP proxy sidecar (more details are explained in subsection 5.1.4) is launched as part of the same Docker Compose stack and shares a volume with the Augur service. To operate correctly, all services in this compose stack must be in the same docker network.

5.1.3 Augur Startup Script

The shell script first creates a `v_host` and a `user` with the necessary permissions for Augur in the already running RabbitMQ instance. Then, git pull is used to pull the desired Augur release into the working directory. In the local deployment scenario, an `.env` file is injected into the directory with the Augur repository together with the customized `docker-compose` file (see subsection 5.1.2). Next, docker compose up is called to start the Augur services, including the HTTP proxy sidecar (see subsection 5.1.4). Finally, Augur is instructed to initialize its database schemas via `docker exec`.

5.1.4 HTTP Proxy Sidecar

As described in subsection 4.3.2, the HTTP proxy sidecar exposes an HTTP-based facade for submitting repositories to Augur. This design was adopted to mitigate security risks by preventing the microservice from accessing the Docker socket, which would otherwise grant root-equivalent privileges on the host system. Additionally, the sidecar approach preserves a clear separation of concerns by decoupling the microservice's application logic from Augur's operational and infrastructural responsibilities. The sidecar pattern was selected to ensure that it operates within the same Docker network as Augur and shares a common volume. Since the `augur db add-repos` command requires a CSV file containing repository URLs as input, a shared volume is necessary to allow both the sidecar and Augur to access this file.

Technology Selection

Since the sidecar should be as lightweight and resource-efficient as possible, Go was chosen for implementation due to its conciseness and mature Docker API support. Go also provides native support for HTTP communication via the `net/http` package. In contrast, a Java Spring Boot–based implementation would introduce unnecessary complexity and resource overhead for a component with limited functionality. A Python-based solution was also considered; however, it would either require additional runtime dependencies or rely on external libraries, which would increase the container footprint and reduce deployment predictability.

Configuration and Initialization

The sidecar is configured at startup via environment variables that specify the listening port and the name of the Augur Docker container. During initialization, an `ExecFacade` component is instantiated, which is provided with a Docker client and a reference to the Augur container and encapsulates the exported functions `execInitAugurRepoGroups` and `execAddRepos` that can execute commands inside the augur container as well as the facade-scoped helper function `execCommand`, which assembles a `docker exec` command with the Docker API. The `execInitAugurRepoGroups` function creates a only once, on startup, a single repository group for SCA Tool, as Augur requires each repository to be associated with a repository group. This grouping mechanism, however, is not of direct relevance to the use case addressed in this thesis.

Repository Submission Workflow

An HTTP handler is registered for the `/addRepos` endpoint using `http.HandleFunc`, which exclusively accepts POST requests and delegates request handling to the `ExecAddRepos` method. The service is then started on the configured port using `http.ListenAndServe`. Upon receiving a POST request containing a JSON array of repository URLs in the request body, `ExecAddRepos` parses the payload and generates a CSV file within the volume shared with Augur, associating the repositories with the previously created repository group. Subsequently, the `augur db add-repos` command is invoked with the path to the generated CSV file. This command is executed inside the Augur container via `docker exec`, resulting in the creation of the repositories within Augur. Finally, the sidecar returns the exit code produced by the Augur command to the calling microservice.

5.2 Health Assessor Microservice

The Health Assessor microservice is integrated into the existing microservice architecture of SCA Tool. It is implemented as an independent Spring Boot application and maintains connections to both the Augur database and that of SCA Tool. Its primary responsibility is to retrieve data from Augur, perform metric calculations where required, and persist the resulting data for subsequent presentation in the frontend interface.

5.2.1 Configuration

This subsection outlines the additional configuration required to adapt a standard Spring Boot project to the specific requirements of the Health Assessor and the infrastructure of SCA Tool.

Lifecycle Configuration

To `ensure` that Augur is initialized during the startup of the microservice, a custom bootstrap initializer was registered within the Spring Boot lifecycle. This was accomplished by configuring the `org.springframework.boot.BootstrapRegistryInitializer` property to reference the `AugurInitializer` class (see subsection 5.2.2). By integrating this initializer into the bootstrap phase, the required setup logic is executed early in the application startup sequence, prior to the full initialization of the main application context. This allows Augur to startup during booting the microservice.

Multi-DataSource Configuration

The Health Assessor microservice requires access to two separate databases: the SCA Tool database, which is used to persist calculated health metrics and related application data, and the Augur database, which stores the raw project and community metrics collected by Augur. As these databases are logically and physically distinct, the microservice must be configured to interact with both data sources concurrently.

To support this, a multi-datasource configuration was implemented for the microservice. Two independent data sources are defined, each with its own connection properties, connection pool, and `JdbcTemplate`. The primary data source is configured for the SCA Tool database and is marked as the default for general database operations, while a secondary data source is configured for accessing the Augur database. Separate `JdbcTemplate` instances are used to ensure that queries are executed against the appropriate database explicitly, thereby avoiding unintended cross-database access and improving maintainability and clarity of data access logic.

Database Tables

The database schema required by the Health Assessor microservice is provisioned using a versioned, declarative, SQL-based schema migration managed by `Flyway`. This migration is executed during the startup of the SCA Tool Monolith to ensure that the required database structures are present and up to date before the Health Assessor begins operation.

The migration creates two dedicated tables, `repositories` and `repository_metrics`, within the SCA Tool database. These tables constitute the persistence layer for all health-related data managed by the Health Assessor. The migration uses conditional statements such as `CREATE TABLE IF NOT EXISTS`, which allows it to be executed multiple times without causing errors and ensures that the required database schema is created consistently across different deployment environments.

The `repositories` table stores normalized metadata for version control repositories, including repository identifiers, names, URLs, and optional references to corresponding Augur identifiers. The `repository_metrics` table persists health metrics associated with each repository via a foreign key relationship and records the time at which metrics were retrieved. Metric values are stored in a `JSONB` column, as the Augur REST API returns metric data in JSON format. The use of `JSONB` allows flexible storage of various metric structures while maintaining efficient querying and indexing capabilities.

RabbitMQ Configuration

Within SCA Tool, inter-service communication is predominantly implemented using the RabbitMQ messaging system. Upon completion of an analysis, the Analyzer emits an event indicating that the process has finished. This event acts as a trigger for the Health Assessor to initiate the retrieval of new version control information, specifically Git repository URLs. To enable this interaction, RabbitMQ was configured in the Health Assessor microservice using the same foundational setup employed by existing SCA Tool components, such as the Monolith and the Scanner.

Building on this configuration, a message binding was introduced using the AMQP (Advanced Message Queuing Protocol) abstraction provided by the `org.springframework.amqp.core` library. The binding routes completion events from the `AnalyzerCompletionQueue` to a dedicated HealthAssessor queue. The Health Assessor microservice subscribes to this queue, consumes the events, and subsequently retrieves the corresponding repository URLs from the SCA Tool database.

Supporting Configuration

In addition to the core service and data access configuration, several auxiliary configuration components were introduced to support communication and data processing within the Health Assessor microservice.

A centrally configured `WebClient` builder is provided to enable HTTP-based communication with external services, such as the HTTP proxy sidecar and the Augur API. The configuration increases the maximum in-memory buffer size to accommodate larger payloads returned by Augur, thereby preventing runtime errors during data retrieval.

Furthermore, a dedicated `ObjectMapper` bean is defined to standardize JSON serialization and deserialization across the microservice. This ensures consistent handling of structured data exchanged between components and simplifies integration with external APIs.

Finally, Augur-specific configuration parameters are externalized using a strongly typed configuration properties class. This approach allows endpoint URLs and command runner settings to be defined declaratively via application configuration files, improving maintainability and enabling environment-specific customization without code changes.

5.2.2 Startup and Environment Initialization

During the startup of the microservice, the component referred to as the `AugurInitializer` is executed. This component implements the `ApplicationListener` interface and reacts to the `ApplicationReadyEvent` provided by the `org.springframework.context` library. The event indicates that the Spring Boot application has completed its initialization phase and is ready for operation.

Once this event is received, the AugurInitializer uses a `ProcessBuilder` to invoke the shell script described in subsection 5.1.3, thereby initiating the startup of the Augur service. If an error occurs during this process, the current thread is interrupted, preventing the microservice from completing its startup sequence.

5.2.3 Repository Discovery and Registration

When the Analyzer microservice (see subsection 4.2.2) emits an `AnalyzerCompletionEvent`, the Health Assessor microservice receives this event via a predefined RabbitMQ routing configuration (see subsection 5.2.1). Event consumption is handled by the `AnalyzerResultMessageListener` component, which defines an event handler method annotated with a suitably configured `@RabbitListener`. Upon receipt of the event, this handler invokes the `RepoDatabaseListener` component and its primary method, `pollVcsData`, which processes the repository URLs previously extracted by the Analyzer and stored in the SCA Tool database.

To retrieve repository URLs, the `RepoDatabaseListener` directly queries the packages table of the SCA Tool database using a `JdbcTemplate`. This approach was deliberately chosen instead of defining dedicated Spring Data repository interfaces and entity mappings for the existing SCA Tool schema. As the Health Assessor only requires read access to a limited subset of fields and does not own the lifecycle of the `packages` table, introducing additional domain entities and repository abstractions would have resulted in unnecessary complexity, tighter coupling between services, and increased maintenance effort. Direct JDBC-based access therefore provides a lightweight and efficient way of integrating with the existing database structure.

The retrieved repository URLs are normalized to a uniform `HTTPS` format, as this

is the only format accepted by Augur. Each valid and accessible URL is persisted to the `repositories` table using an upsert strategy, with the repository URL serving as the unique identifier.

Newly discovered repository URLs are subsequently forwarded to Augur via the HTTP proxy sidecar (see the corresponding subsection 5.2.4). Augur then independently performs the collection of repository data and the extraction of relevant metrics. Following successful submission, the Health Assessor queries the Augur service to retrieve the internally assigned Augur repository identifiers. These identifiers are again upserted to the `repositories` database table, thereby establishing a persistent association between the internal repository records and their corresponding representations within Augur.

5.2.4 Augur Integration

This subsection describes the mechanisms by which the microservice interacts with Augur, differentiating between communication via Augur's internal REST API, interaction through the HTTP proxy sidecar, and direct access to the Augur database using JDBC.

These distinct access mechanisms are required due to the heterogeneous interfaces provided by Augur. Health metrics that are computed internally by Augur can only be retrieved via its REST API, while the addition of new repositories is exclusively supported through command-line–based interactions. Conversely, raw data collected from repositories is only accessible through direct database queries, which enables the Health Assessor to calculate additional metrics independently.

Consequently, the Health Assessor microservice incorporates two separate client components: one responsible for HTTP-based communication with Augur and another that performs direct database access using `JDBC`. These client components are detailed in the following subsections.

REST-Based Integration via WebClient

HTTP-based communication with Augur is implemented using the Spring Boot `WebClient` provided by the `org.springframework.web.reactive.function.client` library. This functionality is encapsulated within a dedicated client service of the Health Assessor microservice, which internally maintains two separate `WebClient` instances and additionally serves as a wrapper for the database client described in the subsequent subsection.

The use of two `WebClient` instances is required to address multiple HTTP endpoints. One client communicates directly with Augur's internal HTTP facade to retrieve metrics that are already calculated by Augur, while the second client interacts with the HTTP proxy sidecar (see subsection 5.1.4) to submit repository

URLs for ingestion.

The client service exposes a set of focused methods that encapsulate REST-based interactions with Augur. These include operations for submitting repositories via `HTTP POST` requests as well as retrieving precomputed metrics via `HTTP GET` requests. For example, metrics such as issue duration or contributor statistics are obtained by invoking repository-specific REST endpoints and returning the corresponding responses. All REST responses are retrieved synchronously and processed as JSON payloads, enabling seamless integration with the subsequent metric processing and persistence logic of the Health Assessor.

Augur Database Integration

Direct database access is encapsulated within a dedicated client component that uses a `JdbcTemplate` configured for the Augur database (see subsection 5.2.1. This approach was deliberately chosen over object–relational mapping or repository abstractions, as the Health Assessor performs read-only queries on a well-defined subset of Augur's schema and does not own or modify the underlying data structures. Using JDBC allows precise control over executed queries while keeping coupling to Augur's internal data model minimal.

The database client provides a set of focused query methods that retrieve repository-related raw data, such as release information, contributor activity, pull request statistics, and issue interaction timestamps. Time-based filters are applied to limit queries to relevant observation windows, ensuring that calculated metrics reflect recent project activity. Retrieved data is post-processed within the microservice where necessary, for example to calculate metrics such as the bus factor or change request closure ratio.

Query results are serialized into `JSON` representations to ensure a uniform data format across the Health Assessor, regardless of whether metrics originate from Augur's REST API or direct database access. This uniform representation simplifies downstream processing and persistence within the SCA Tool database.

5.2.5 Metric Collection and Computation

This subsection outlines the details of the periodic metric retrieval within the Health Assessor. It explains how repository health metrics are fetched from Augur, how derived metrics are computed, and how both types are stored persistently for later consumption by the Monolith.

Metric Model and Classification

To ensure a consistent and extensible representation of repository health metrics, the Health Assessor defines a centralized metric model that captures both the

semantic meaning of a metric and its retrieval strategy. This model is implemented as an `enum` that serves as the central entity of all supported health metrics within the system.

Each of these `enums` is associated with three core attributes: an identifier used for database storage, a classification indicating whether the metric is fetched or computed, and, where applicable, a binding to the corresponding retrieval method of the Augur client. Metrics classified as `FETCHED` represent indicators that can be obtained directly from Augur, either via its REST API or through database access. Metrics classified as `COMPUTED`, in contrast, are later calculated by the Health Assessor based on previously retrieved data and therefore do not define a direct retrieval method.

For fetched metrics, the model establishes an explicit functional association between a metric and the Augur client method responsible for retrieving its value. This association is implemented using Java's `BiFunction` interface, which binds each metric to the corresponding retrieval function provided by the `AugurClient` (see subsection 5.2.4). This design allows the metric collection logic to invoke metric retrieval in a generic and uniform manner, without embedding metric-specific control flow in the collection process. Computed metrics are excluded from this mechanism and are handled separately during the metric calculation phase.

By centralizing metrics into the `enum` model, and having retrieval bindings within a single model, the Health Assessor achieves a clear separation between metric semantics and execution logic. This design simplifies extensibility, as new metrics can be introduced by adding a single definition to the metric model without requiring structural changes to the surrounding collection or persistence workflows.

Fetched Metric Retrieval

To retrieve metrics classified as `FETCHED` and persist them in the database, the `MetricsCollectorService` is employed. Its primary method, `fetchAllMetricsForAllRepositories`, is invoked periodically according to a configurable schedule. For each metric defined in the metric `enum` with the type `FETCHED`, the associated `BiFunction`, as described in the previous subsection, is executed to retrieve the corresponding metric value from Augur. The retrieved results are subsequently stored in the database (see subsection 5.2.1) in `JSON` format using an upsert strategy.

Computed Metric Derivation

Computed metrics are derived using a workflow that is largely analogous to the retrieval of fetched metrics, with key differences in the source and processing of the data. Metric computation is performed by the `MetricsCalculatorService`,

whose primary method, `calculateAllMetricsForAllRepositories`, is invoked periodically by a scheduler in the same manner as metric retrieval. The service iterates over all metric definitions and selects those classified as `COMPUTED`. For each such metric, the corresponding calculation method is executed. The resulting values are subsequently stored in the database using the same upsert strategy employed for fetched metrics.

To support these calculations, a dedicated utility class providing reusable `static` statistical methods was introduced, enabling consistent and centralized implementation of common aggregation and statistical operations.

5.3 Monolith and API

This section describes the modifications introduced in the Monolith to support the integration of the Health Assessor microservice, as well as the design and operation of the API based on an OpenAPI specification. It examines the relevant services implemented within the Monolith and outlines how the OpenAPI definition enables structured communication between system components.

5.3.1 API Specification and Code Generation

The API is formally specified using an OpenAPI definition, which serves as the single source of truth for the contract between the backend and the web-based user interface. Based on this specification, interfaces and data transfer objects are automatically generated for the Monolith, ensuring type-safe controller implementations and consistent request and response handling. In parallel, corresponding client-side types and service interfaces are generated for the frontend. This approach guaranties alignment between backend and frontend representations, reduces manual boilerplate code, and minimizes the risk of inconsistencies when evolving the API.

Health Assessor API Integration

To expose health assessment results to the user interface, the Monolith provides a consolidated API endpoint that supports bulk retrieval of CHAOSS-based health metrics. Instead of querying metrics on a per-repository basis, a single endpoint is defined that accepts a list of package identifiers and returns the corresponding aggregated health metrics for all associated repositories in one request. This design reduces network overhead and allows efficient retrieval of health data for projects with multiple dependencies.

The endpoint accepts a `JSON` request body containing an array of package URLs and responds with a structured object that maps each package to its aggregated

health metrics. The response model encapsulates all metrics, that were specified in the system (see subsection 5.2.5) thereby providing a comprehensive view of repository health within a single response payload.

As the user interface requires access to the complete set of health metrics in a single view (see section 5.4), this approach significantly reduces the number of API requests and therefore improves overall performance.

5.3.2 Monolith Services

Within the Monolith, health metric retrieval and aggregation are encapsulated in a dedicated service component that serves as the central access point for persisted metrics. This service abstracts the underlying database representation and provides a stable interface for the API layer.

The `ChaossMetricService` exposes a method for retrieving health metrics for multiple repositories in bulk. Its implementation accesses metric data stored in the SCA Tool database in a generic `JSON`-based format, allowing different metric types to be handled uniformly. Individual metric values are retrieved through a generic `getMetric` function, which resolves a metric by repository identifier and metric name and applies a type-specific mapping function. This design avoids metric-specific retrieval logic and enables concise, type-safe access to heterogeneous metric data. This design also allows to be extended safely and with minimal effort as additional health metrics are integrated.

For API consumption, the service aggregates all relevant metrics of a repository into a single OpenAPI-generated `ChaossHealthMetricsApi` data transfer object. Bulk retrieval is supported by iterating over multiple package identifiers and returning the aggregated health metrics in a single response, thereby aligning with the requirements of the user interface and minimizing API overhead.

5.4 User Interface Integration

The user interface of SCA Tool is web-based and implemented using a React, Next.js, and TypeScript technology stack. Access to the system requires user authentication. After logging in, users can initiate the Analyzer, Health Assessor, and Scanner workflow on a given codebase by providing the corresponding repository URL, as described in section 4.1. Upon completion, an overview of the analyzed project is presented, accompanied by a sidebar navigation that provides access to various views, such as the Software Bill of Materials (SBOM) and the Legal Notices Overview.

This chapter describes the integration of the Health Assessor into the existing user interface, explains its placement within the overall navigation structure,

and outlines the newly introduced components required to present health-related information to the user.

5.4.1 Placement in the User Interface

Similar to the SBOM and the Legal Notices Overview, a dedicated tab for the Health Assessor was added to the left-hand sidebar of the repository overview. This design decision was motivated by two primary factors. First, the volume and complexity of health-related data necessitate a dedicated view that provides sufficient space for meaningful presentation. Second, the health assessment represents a conceptually distinct aspect of the analysis and therefore cannot be logically integrated into any of the existing overview categories. By integrating the page in this manner, the Health Assessor becomes a first-class element of the repository overview while remaining consistent with the existing navigation and routing conventions of the SCA Tool user interface.

Integration

The Health Assessor view was integrated into the repository overview by adding a dedicated route within the existing Next.js–based routing structure and exposing it through a corresponding entry in the left-hand sidebar. The route is parameterized with the version unit identifier (`verUnitId`), which represents a concrete version of an analyzed project and serves as the contextual anchor for all subsequent data retrieval.

Upon navigation to the Health Assessor tab, the associated page component resolves the `verUnitId` and renders a dedicated Health Assessor component. This component uses the version unit to retrieve the set of packages associated with the analyzed project version and to derive the corresponding repository references. These package-level references form the basis for requesting aggregated health metrics from the backend.

Access to the Health Assessor view is controlled via feature flag, allowing the tab to be conditionally enabled without affecting the overall navigation structure. Through this integration, the Health Assessor is seamlessly embedded in the existing repository overview while maintaining a clear contextual link between project versions, packages, and their associated health metrics.

5.4.2 Health Metrics Data Grid

The core component of the Health Assessor user interface is a tabular data grid that provides a consolidated overview of health metrics for all packages associated with a specific package url (PURL). Implemented using the Material UI

`DataGrid`[1] component, it supports pagination, sorting, and custom cell rendering to ensure scalable and structured presentation of metric data.

Each row represents a package–repository pair and is populated from a mapping of `purls` to aggregated health metrics retrieved from the backend. Metric values are displayed using dedicated evaluation components that combine numerical values with qualitative health classifications based on predefined threshold configurations, enabling easy comparison and evaluation across repositories.

Several metrics are transformed prior to display to improve readability, such as converting time-based values into hours or days and expressing ratio-based metrics as percentages. Repository URLs are rendered as external links with abbreviated labels to maintain clarity while preserving direct access to the source repositories.

In addition to individual metrics, the data grid includes an aggregated risk indicator (see subsection 5.4.4) that summarizes overall repository health based on the distribution of metric evaluations. The grid also provides lightweight interaction by allowing repositories to be marked as completed via a checkbox column, visually de-emphasizing completed entries to improve usability in projects with many dependencies.

By combining metric visualization, qualitative assessment, and basic interaction within a single component, the data grid forms the central element of the Health Assessor tab and enables for efficient exploration of repository health across dependencies of a project.

5.4.3 Metric Retrieval from the Backend

Health metrics are retrieved using a dedicated React hook called `useHealthMetrics` that encapsulates the complete data fetching logic and exposes a minimal, reusable interface to consuming components. This hook is responsible for preparing the request data, invoking the backend bulk API (see section 5.3), and managing the associated loading and error states.

The hook accepts a list of package-related information as input and derives the relevant package identifiers (`purls`) using memoization via React's `useMemo`. This ensures that the list of identifiers is only recalculated when the underlying package data changes, thereby preventing unnecessary refetching caused by unrelated component updates. Packages without associated repository URLs are filtered out at this stage, as they cannot be resolved to health metrics.

Data fetching is performed inside a `useEffect` hook that is triggered whenever the derived set of `purls` changes. The effect issues a single bulk request to

[1] https://mui.com/x/react-data-grid/

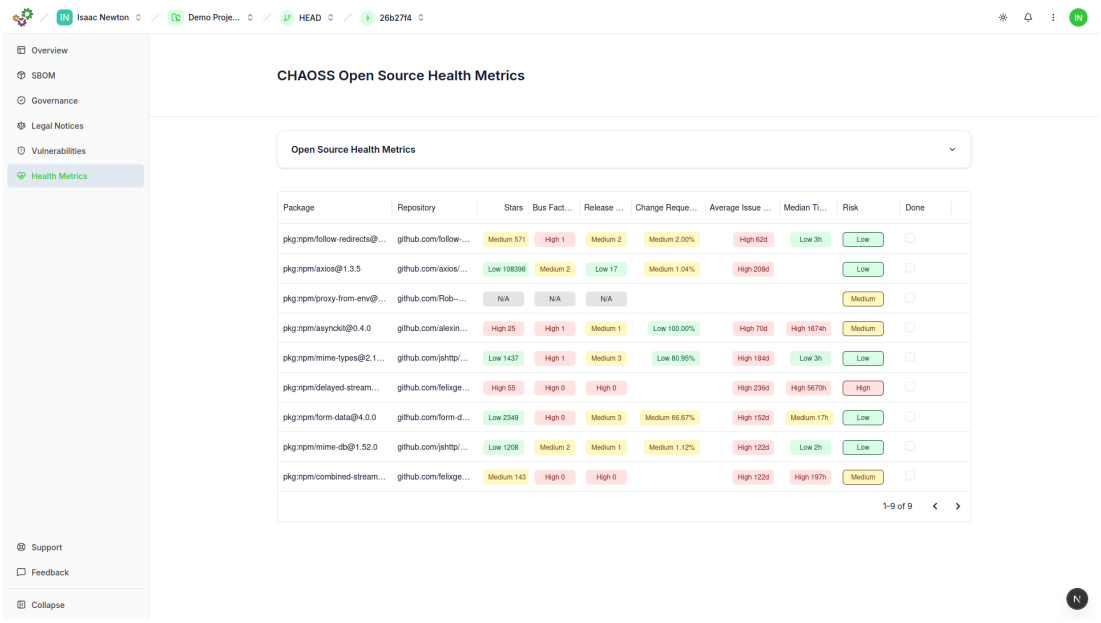

Figure 5.1: The Health Assessment in SCA Tool's UI

the backend health metrics endpoint, consolidating all required data retrieval into one network call (see section 5.3). The local component state is used to track the retrieved data, loading status, and potential errors, allowing the UI to reactively render loading indicators or error messages as needed. A cancellation flag is employed to prevent state updates after component unmounting, following recommended practices for handling asynchronous operations in React.

5.4.4 Health Evaluation Chip

To support the classification of package health, a dedicated wrapper component was implemented around the shadcn[2] `Badge` component. This component categorizes the assessed risk of a metric into three levels: low (green), medium (yellow), and high (red).

The resulting `HealthEvaluationChip` receives both the metric-specific threshold values and the evaluated metric value as input parameters and determines the corresponding risk category accordingly. In addition to the color-based representation, the textual classification (low, medium, or high) is displayed to ensure accessibility for users with color vision deficiencies. Metrics that are unavailable, either because they have not yet been collected or because the corresponding data has not been provided by Augur, are displayed in gray and labeled with the value `N/A`.

[2]https://ui.shadcn.com/

The component is designed to be reusable across the user interface. For example, it can be used in a purely visual mode, as within the integrated wiki (see subsection 5.4.5), where only the color indicator is shown while the textual classification is omitted.

5.4.5 Wiki

As the interpretation of health metrics is highly context-dependent and simplified classifications may be misleading when viewed in isolation, a lightweight informational wiki was introduced above the DataGrid (see subsection 5.4.2). This wiki provides users with explanatory context for each metric and guidance on how the displayed values should be interpreted. By placing this information directly within the Health Assessment view, users are supported in making informed judgments without requiring external documentation.

The wiki component was deliberately designed as a generic and reusable UI element that can be populated with any content and reused across different sections of SCA Tool. Its layout consists of a navigation menu on the left-hand side listing individual wiki sections and a scrollable content area on the right that displays the corresponding explanatory text. This structure enables efficient navigation within longer documentation while maintaining a compact visual footprint. The implementation is based on shadcn UI components, ensuring visual consistency with the rest of the application.

From an implementation perspective, the wiki is realized as a configurable `WikiCard` component that accepts three primary properties: an optional title, a list of navigation links, and the content to be displayed. Each navigation link is defined by a unique identifier and a display title. These identifiers are also assigned to the corresponding sections within the content, allowing the navigation menu to scroll the content area to the appropriate position. Scrolling behavior is implemented using a `ScrollArea` component in combination with a reference to the underlying viewport, enabling smooth, programmatic scrolling to individual sections upon user interaction.

This approach decouples structure and content, allowing the same wiki component to be reused with different link sets and content definitions, while maintaining consistent interaction behavior and visual appearance throughout the SCA Tool user interface.

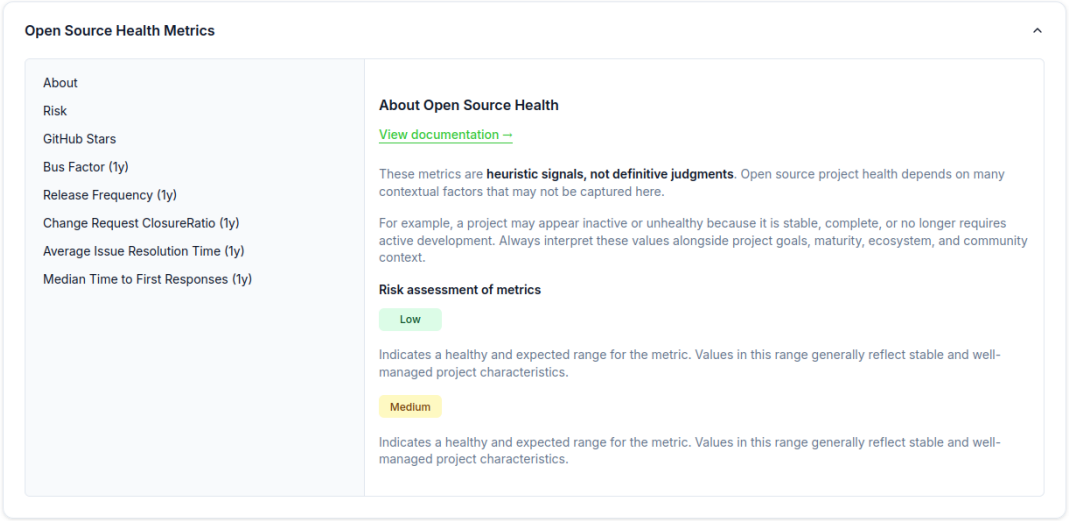

Figure 5.2: The foldable Wiki Component in SCA Tool's UI

6 Evaluation

This chapter assesses the requirements defined in chapter 3 with regard to their degree of fulfillment and overall completeness. It discusses the objectives achieved, highlights the strengths of the implemented solution, and identifies and evaluates the remaining challenges and limitations.

6.1 Requirements Evaluation

The implemented health assessment system is evaluated by systematically comparing the solution with the requirements defined in chapter 3. Each requirement is examined individually to determine whether it has been fully, partially, or not fulfilled. Functional requirements are evaluated based on the concrete implemented features and their visible behavior within the system. This includes reviewing the corresponding components, API endpoints, and user interface elements. Non-functional requirements are evaluated based on architectural design decisions, integration strategy, usability considerations, and maintainability aspects.

6.1.1 Functional Requirements

This subsection examines each functional requirement defined in section 3.1 and evaluates its degree of fulfillment according to the evaluation criteria outlined previously.

FR-01: The system shall evaluate SCA Tool–scanned packages using open-source health metrics defined by the CHAOSS foundation

Evaluation: Fulfilled

- Packages extracted by the Analyzer are propagated via RabbitMQ (see section 4.1, subsection 5.2.1).
- The Health Assessor retrieves the associated repositories and applies CHAOSS-defined metrics (see subsection 5.2.5).

- The calculated metrics are persisted and visualized in the frontend (see subsection 5.4.4).

The system therefore evaluates scanned packages using the selected CHAOSS metrics, and the requirement is considered fully satisfied.

FR-02: The system shall provide health metrics for extracted packages that are associated with a valid and accessible Git repository.

Evaluation: Fulfilled.

- Repository URLs are validated during processing; invalid, missing, or inaccessible repository references are filtered out and not further processed (see subsection 5.2.3).
- For all packages associated with a valid and accessible Git repository, health metrics are retrieved and evaluated (see subsection 6.1.1).

Consequently, health metrics are provided exclusively for packages that meet the validity and accessibility criteria, and the requirement is therefore considered fully satisfied.

FR-03: The system shall periodically update previously collected health metrics.

Evaluation: Fulfilled

- The Health Assessor performs scheduled retrieval of metrics from Augur at fixed intervals (see subsection 5.2.5).
- The calculation of derived (computed) metrics is likewise executed periodically via a scheduler (see subsection 5.2.5).

Through this recurring retrieval and recalculation process, previously collected metrics are continuously updated, thereby satisfying the requirement.

FR-04: The system shall persist collected health metrics in the SCA Tool database.

Evaluation: Fulfilled.

- The Health Assessor stores all fetched and computed metrics in dedicated database tables within the SCA Tool database schema (see subsection 5.2.1).
- Metrics are persisted in structured form, including repository association, metric name, JSON-based value representation, and retrieval timestamp.

All collected and calculated health metrics are therefore permanently stored in the SCA Tool database, ensuring traceability and subsequent retrieval. The requirement is considered fully satisfied.

FR-05: The system shall use a suitable tool to collect open-source project health data.

Evaluation: Partially fulfilled.

- The Health Assessor integrates the CHAOSS-based tool Augur for collecting repository-related data (see subsection 2.3.3 and subsection 4.3.1)
- Augur provides the raw data necessary to retrieve and calculate the defined health metrics.

Augur supplies all required data for the implemented metric set and is conceptually aligned with the objectives of this thesis. However, its architecture is primarily designed as a standalone system and offers limited external controllability and orchestration capabilities. This results in reduced flexibility when integrating and managing data collection within the SCA Tool ecosystem.

While Augur is technically suitable for collecting health-related data, these architectural constraints limit seamless integration. The requirement is therefore considered only partially fulfilled.

FR-06: The system shall support bulk retrieval of health metrics for multiple packages in a single request.

Evaluation: Fulfilled.

- The Monolith exposes a dedicated API endpoint that aggregates and returns health metrics for multiple packages within a single request (see section 5.3).
- The endpoint accepts a list of package identifiers and responds with a structured mapping of packages to their corresponding health metrics.

Through this consolidated API design, the system enables efficient retrieval of health metrics for projects containing multiple dependencies while reducing network overhead. The requirement is therefore considered fully satisfied.

FR-07: The system shall expose health metrics through a well-defined API that provides a stable interface for the SCA Tool frontend.

Evaluation: Fulfilled.

- The system defines an OpenAPI specification that formally describes the health metric endpoints and their request and response structures (see section 5.3).
- Based on this specification, strongly typed data models and interfaces are automatically generated for both the Spring Boot backend and the React-based frontend, ensuring consistent and type-safe integration.

By providing a formally specified and automatically generated API contract, the system enables seamless and reliable consumption of health metrics within the SCA Tool frontend. The requirement is therefore considered fully satisfied.

FR-08: The system shall allow users to mark individual packages as reviewed by the user within the user interface.

Evaluation: Partially fulfilled.

- The user interface provides a mechanism to mark packages as reviewed, making the review status explicitly visible within the Health Assessment view (see subsection 5.4.2).
- However, the review state is currently persisted only locally within the browser using local storage and is not stored centrally in the database on a per-user or per-organization basis.

While a functional mechanism for marking packages as reviewed exists, the lack of centralized persistence limits its practical applicability across sessions and users. As a result, the requirement is considered partially fulfilled.

FR-09: The system shall provide users with detailed explanations of collected health metrics and guidance on their interpretation.

Evaluation: Fulfilled.

- The user interface integrates a dedicated wiki section that provides detailed explanations of each implemented health metric, including contextual interpretation guidance and notes on potential misinterpretation (see subsection 5.4.5).
- The wiki is directly embedded within the Health Assessment view, ensuring that explanatory information is accessible alongside the metric visualization (see Figure 5.2).

By supplying contextual documentation within the application itself, the system supports informed interpretation of health metrics and reduces the risk of misleading conclusions. The requirement is therefore considered fully satisfied.

6.1.2 Non-Functional Requirements

This subsection examines each non-functional requirement defined in section 3.2 and evaluates its degree of fulfillment according to the evaluation criteria outlined previously.

NFR-01: The system shall be implemented as an independent microservice within the existing SCA Tool architecture.

Evaluation: Fulfilled.

- The Health Assessment System is implemented as a standalone Spring Boot application with its own lifecycle, configuration, and deployment process (see subsection 4.3.3).
- Communication with other SCA Tool components is performed via well-defined interfaces and messaging mechanisms, ensuring architectural separation (see subsection 4.2.6 and subsection 5.2.1).

The Health Assessor operates independently while integrating into the overall SCA Tool ecosystem through clearly defined interfaces. The requirement is therefore considered fully satisfied.

NFR-02: The system shall reuse the existing backend infrastructure of SCA Tool to the greatest extent possible.

Evaluation: Fulfilled.

- The Health Assessor reuses the existing SCA Tool database instance and integrates with the established Monolith services for API-based data provision (see subsection 4.2.1 and subsection 5.3.2).
- The OpenAPI-based API configuration is reused to ensure consistent interface definitions between backend and frontend (see section 5.3).
- Existing infrastructure components, including RabbitMQ for messaging and Redis for caching, are reused without introducing additional parallel instances (see subsection 4.2.4).

By using the established backend infrastructure and avoiding redundant components, the Health Assessor minimizes architectural complexity and ensures consistent integration within the SCA Tool ecosystem. The requirement is therefore considered fully satisfied.

NFR-03: The system shall reuse the existing frontend infrastructure of SCA Tool to the greatest extent possible.

Evaluation: Fulfilled.

- The Health Assessment UI components are implemented within the existing React-based frontend architecture and follow established routing and layout conventions (see section 5.4).
- Existing UI libraries and design components, including shadcn and MUI, are reused to ensure visual and technical consistency.

By building upon the established frontend structure and component ecosystem, the Health Assessment feature integrates seamlessly into the SCA Tool user interface without introducing parallel design systems or architectural deviations. The requirement is therefore considered fully satisfied.

NFR-04: The system shall be integrated into the existing SCA Tool user interface.

Evaluation: Fulfilled.

- The Health Assessment feature is accessible via the existing repository sidebar and follows the established navigation structure of SCA Tool (see section 5.4 and Figure 1).

The Health Assessment is therefore fully embedded within the existing UI structure without introducing separate navigation patterns. The requirement is considered fully satisfied.

NFR-05: The system's user interface shall be aligned with the existing visual design and styling conventions of SCA Tool.

Evaluation: Fulfilled.

- Existing style guides, theming mechanisms, and layout conventions were consistently applied to ensure visual homogeneity and a seamless user experience (see Figure 1).

By adhering to the established design system and component library, the Health Assessment feature maintains visual consistency with the overall SCA Tool interface. The requirement is therefore considered fully satisfied.

NFR-06: The system shall present collected health metrics in an informative and intuitive manner within the user interface.

Evaluation: Fulfilled.

- The user interface employs a structured DataGrid component present both individual metric values and their qualitative assessment, providing a concise and meanwhile informative overview (see Figure 1).

By combining tabular organization with intuitive visual cues, the system enables users to quickly interpret metric results while retaining access to detailed information. The requirement is therefore considered fully satisfied.

NFR-07: The system shall present assessed project health risks using a simple, intuitive color-coded scheme, indicating low risk in green, medium risk in yellow, and high risk in red.

Evaluation: Fulfilled.

- The user interface implements a dedicated HealthEvaluationChip component that visually encodes risk levels using the prescribed color scheme: green (low risk), yellow (medium risk), and red (high risk) (see Figure 1 and subsection 5.4.4).

The consistent use of a clearly distinguishable color scale allows users to quickly recognize risk levels and interpret them consistently across different metrics. The requirement is therefore considered fully satisfied.

NFR-08: The system shall avoid tight coupling between the Health Assessor and external data providers.

Evaluation: Not fulfilled.

- The Health Assessor directly accesses Augur's database schema via JDBC (see subsection 5.1.1), which creates structural dependencies on Augur's internal data model.
- Since Augur exposes only a limited subset of the required data through its official REST API (see subsection 4.3.1), certain metrics necessitate direct database queries, making stronger decoupling impractical.

Due to the reliance on Augur's internal schema for retrieving raw data, the Health Assessor remains tightly coupled to Augur's implementation. The requirement is therefore not fulfilled.

6.2 Limitations and Challenges

This section examines in greater detail those requirements identified in subsection 6.1.1 and subsection 6.1.2 as partially fulfilled or not fulfilled. It analyzes the underlying technical and architectural constraints and discusses the reasons that prevented full compliance.

6.2.1 Limitations of Augur as a Data Collection Tool

As discussed in subsection 6.1.1, Augur is only partially suitable as a data collection tool for the Health Assessor. While it provides access to relevant repository data, its architecture and intended usage introduce several limitations.

Augur is primarily designed as a standalone system with its own frontend and internal processing logic, rather than as a dedicated external data provider. Consequently, its public REST API exposes only a limited subset of functionality. For example, the addition of repositories requires the use of the sidecar component (see subsection 5.1.4), and only a small portion of the required CHAOSS metrics can be retrieved directly via the API. This necessitates supplementary mechanisms, including direct database access, which increases coupling (see subsection 6.1.2).

Another limitation arises from Augur's data collection process. Augur gathers repository information by interacting with Git hosting platforms, primarily GitHub. This process requires a large number of requests to the GitHub API, which enforces strict rate limits and provides higher request quotas only through paid tiers. As a result, the initial traversal and analysis of repositories may take a considerable amount of time, particularly for large projects. Consequently, there can be a noticeable delay before the Health Assessor is able to provide a complete health assessment for newly added repositories within SCA Tool.

Furthermore, external control over Augur's internal processes is not supported. Status monitoring, scan prioritization, or fine-grained orchestration of repository processing would require significant modifications to Augur's source code, leading to even tighter integration and maintenance overhead.

Overall, Augur represents a practical and capable solution for collecting repository data; however, due to its architectural constraints and limited external controllability, it does not fully meet the ideal requirements of a flexible, decoupled data provider for the Health Assessor.

6.2.2 Implementation Effort Constraints

As discussed in subsection 6.1.1, the review status of packages is currently persisted only in the browser's local storage rather than at user or organizational level within the database.

A fully persistent, user-aware solution would have required additional backend extensions, data model adjustments, and authorization logic to support multi-user state management. Given the scope and time constraints of this thesis, such an implementation would have introduced disproportionate development effort relative to its contribution to the core objective of health metric integration.

The chosen approach therefore represents a pragmatic trade-off between implementation complexity and functional benefit.

7 Future Work

This chapter outlines potential extensions and enhancements that could further improve the Health Assessor beyond the scope of this thesis. It discusses opportunities for functional expansion, architectural refinement, and improved usability that may be addressed in future work.

Additional CHAOSS Metrics

Within the scope of this thesis, the metrics defined in section 2.3 as part of the CHAOSS Project Health Starter Set were integrated into the Health Assessor. However, the CHAOSS framework defines a substantially broader range of metrics that could be incorporated in future extensions.

Due to the existing integration with both Augur's database and its REST API, the addition of further metrics can be considered comparatively low in implementation effort. The established architectural structure therefore provides a solid foundation for incremental expansion of the metric portfolio.

Extended Context Research for Metric Evaluation

As discussed in subsection 2.4.1, the interpretation of health metrics is strongly context-dependent. For this reason, future work could focus on incorporating additional contextual information into the Health Assessor in order to support more nuanced evaluations of repository health. Such contextual data may include time-series analyses of metric development, project age, contributor growth trends, or characteristics of the surrounding technology ecosystem. Integrating this information would allow users to assess not only isolated metric values but also their development over time and their relevance within a broader project context.

From a technical perspective, this data could either be derived from already aggregated information within Augur or obtained through direct integration with

external APIs, such as the GitHub API. Providing these contextual insights within the user interface would further enhance interpretability and improve the overall decision-support capabilities of the system.

Administrative Configuration in the User Interface

As described in subsection 5.4.4, the current interpretation of metric values in the user interface, specifically the color-coding of the Health Evaluation Chip as green, yellow, or red, is defined statically in the system's source code. Consequently, the threshold values that determine these classifications are fixed by the system provider.

A potential enhancement would be to allow these thresholds to be configurable at the organizational level within SCA Tool, for example through an administrative dashboard. Such a configuration mechanism would enable organizations to adjust the interpretation and weighing of individual metrics according to their specific requirements and risk preferences, thereby improving the flexibility and applicability of the health assessment.

8 Conclusion

As outlined in chapter 1, the concept of the Health Assessor and the evaluation of community health for open-source packages contained in SCA Tool–scanned projects originates from the intention to complement traditional software composition analysis with additional insights into project activity and sustainability. Beyond security and licensing considerations, the goal was to provide users with an overview of the development dynamics and community engagement surrounding a package, thereby supporting more informed decisions regarding its adoption or continued use.

Since the health assessment functionality within SCA Tool was conceived as a prototype and exploratory engineering effort, the objectives of this thesis can be considered largely fulfilled. The implemented solution successfully integrates a Health Assessor microservice into the existing SCA Tool architecture, collects and computes selected CHAOSS-based metrics, processes these metrics for presentation in the user interface, and enables users to derive meaningful insights about the health of dependent open-source projects. Most of the defined requirements were therefore achieved.

Nevertheless, certain limitations remain. Some are technical in nature, such as architectural constraints and integration dependencies, while others arise from the inherent challenges of interpreting socio-technical data. Metrics describing community activity and project dynamics cannot always provide definitive conclusions and must therefore be interpreted within their respective context.

Despite these limitations, the work demonstrates the value of incorporating community health indicators into software composition analysis. It provides a solid foundation for assessing the health of open-source components used within a project and enables the identification of potential risks or “bad smells” that may warrant further investigation. As organizations increasingly depend on open-source software, maintainers and developers require reliable information about the sustainability and activity of the projects they depend on. The Health Assessment functionality implemented in SCA Tool provides a solid foundation for further development and refinement of such capabilities.

Appendices

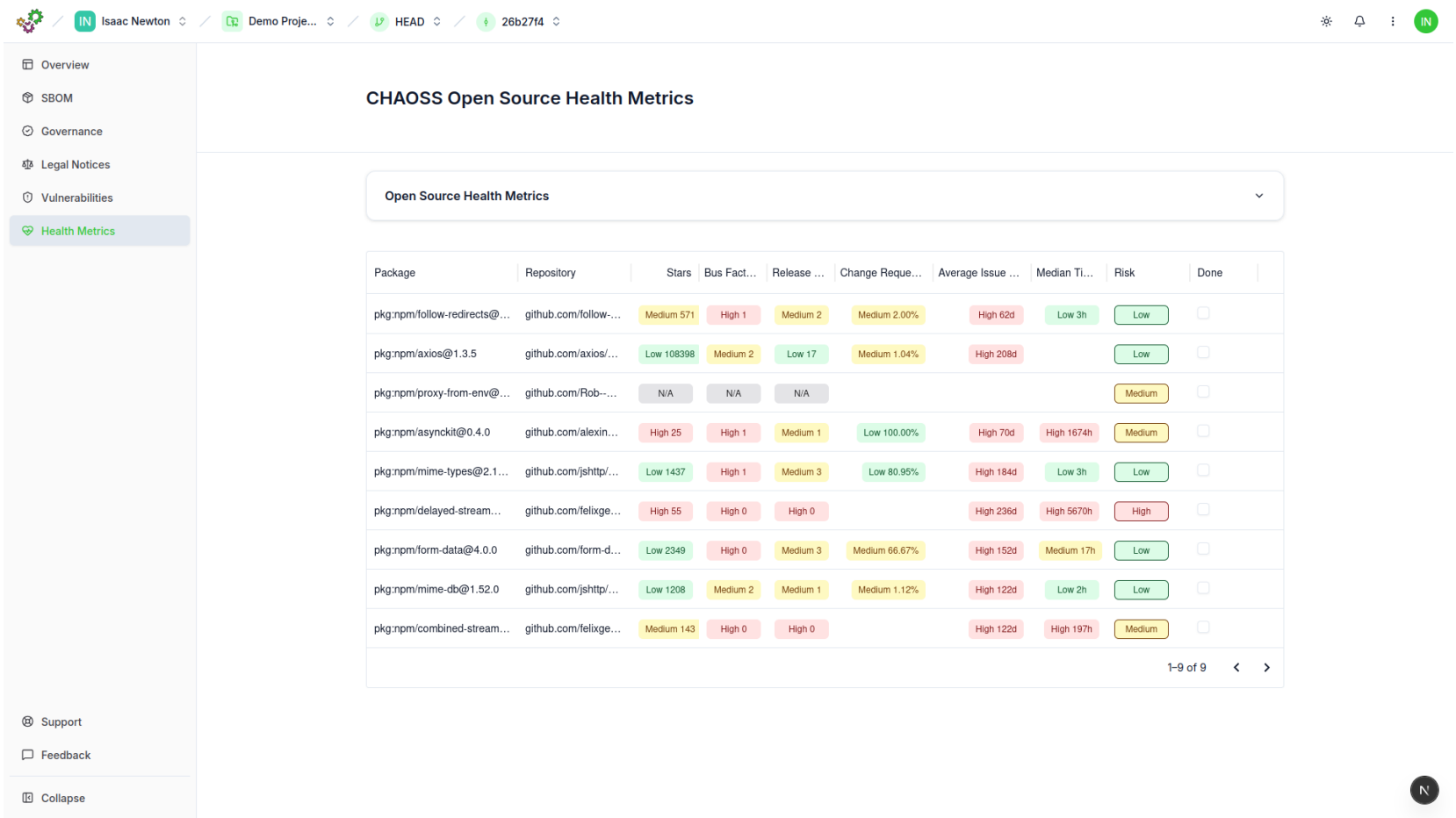

Figure 1: The Health Assessment in SCA Tool's UI

References

CHAOSS. (2024a, October). Metric: Time to first response. Retrieved November 28, 2025, from https://chaoss.community/kb/metric-time-to-first-response/

CHAOSS. (2024b, November). Metric: Change request closure ratio. Retrieved November 25, 2025, from https://chaoss.community/kb/metric-change-request-closure-ratio/

CHAOSS. (2024c, November). Metric: Contributor absence factor. Retrieved November 28, 2025, from https://chaoss.community/kb/metric-contributor-absence-factor/

CHAOSS. (2024d, November). Metric: Release frequency. Retrieved November 25, 2025, from https://chaoss.community/kb/metric-release-frequency/

CHAOSS. (2025a). Home. Retrieved November 24, 2025, from https://chaoss.community/

CHAOSS. (2025b). Software. Retrieved November 28, 2025, from https://chaoss.community/software/

Colt, J. (2023). An introduction to using metrics to assess the health and sustainability of library open source software projects [Repository: Code4Lib Journal]. *The Code4Lib Journal*, (57). Retrieved November 12, 2025, from https://journal.code4lib.org/articles/17514?utm_source=rss&utm_medium=rss&utm_campaign=an-introduction-to-using-metrics-to-assess-the-health-and-sustainability-of-library-open-source-software-projects

Dabbish, L., Stuart, C., Tsay, J., & Herbsleb, J. (2012). Social coding in GitHub: Transparency and collaboration in an open software repository. *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work*, 1277–1286. https://doi.org/10.1145/2145204.2145396

Goggins, S., Lumbard, K., & Germonprez, M. (2021). Open source community health: Analytical metrics and their corresponding narratives. *2021 IEEE/ACM 4th International Workshop on Software Health in Projects, Ecosystems and Communities (SoHeal)*, 25–33. https://doi.org/10.1109/SoHeal52568.2021.00010

Goggins, S. P., Germonprez, M., & Lumbard, K. (2021). Making open source project health transparent. *Computer*, *54*(8), 104–111. https://doi.org/10.1109/MC.2021.3084015

Nagle, F., Powell, K., Zitomer, R., & Wheeler, D. A. (2024, December). *Census III of Free and Open Source Software: Application Libraries* (tech. rep.). The Linux Foundation. https://doi.org/10.70828/JRPB8299

OpenSSF. (n.d). Openssf scorecard. Retrieved February 11, 2026, from https://scorecard.dev/

Qiu, H. S., Lieb, A., Chou, J., Carneal, M., Mok, J., Amspoker, E., Vasilescu, B., & Dabbish, L. (2023). Climate coach: A dashboard for open-source maintainers to overview community dynamics. *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. https://doi.org/10.1145/3544548.3581317

Schrock, S., Shaikh, A., laurentsimon, Arya, A., Kaul, R., AdamKorcz, Augustus, S., Wheeler, D. A., Lewandowski, K., olivekl, Vereshchagin, E., Gutierrez, G., asraa, Wang, Y., afmarcum, dlorenc, Magee, J., Costello, M., Engelen, A., … Shearin, A. (2026, February). Ossf/scorecard. Retrieved February 11, 2026, from https://github.com/ossf/scorecard