Hochschule Karlsruhe
University of
Applied Sciences

Fakultät für
Informatik und
Wirtschaftsinformatik

Master Thesis

Name:	Fabian Burth
Topic:	Design and Implementation of a Security and Compliance Data Lake
Place of work:	SAP SE, Walldorf
Supervisor:	Prof. Dr.-Ing. Vogelsang
Co-examiner:	Prof. Dr. Körner
Deadline:	16/02/2023

Karlsruhe, 17/08/2022

The Chairman of the examination committee

[illegible]

Prof. Dr. Heiko Körner

Statutory declaration

Abstract

The importance of software is constantly growing and so is its complexity. Thus, large-scale enterprise software systems are usually not developed completely from scratch. Commonly required functionality like logging or serialization is often provided by already existing *open source software (OSS)*. Hence, modern software systems are composed of several components. These components usually have individual version updates, vulnerabilities, and licenses. Version updates of a component may affect the compatibility with other components. Vulnerabilities in one specific component may compromise the security of the whole software system. Violating license agreements may lead to litigations due to copyright infringement. Furthermore, one must consider that each component itself may be composed of several other components.

So, maintaining and monitoring large-scale enterprise software systems is an important part of the *Application Lifecycle Management (ALM)* and poses a considerable challenge to software companies. The common way to tackle these risks nowadays is by incorporating *Software Composition Analysis (SCA)* tools into the application development process. These tools analyze applications and retrieve information like vulnerabilities, licenses, and *software bill of materials (SBOM)*. But this approach often still has its flaws. The information extracted by these tools is frequently treated like logs and hence of limited value for future usage. Additionally, in larger companies different development teams often come up with point-to-point solutions of integrating the tools tightly coupled to their CI/CD pipeline.

The main objective of this thesis is the design and prototypical implementation of a *Security and Compliance Data Lake (SCDL)*, which provides a standardized way of integrating even multiple different SCA tools loosely coupled to CI/CD pipelines and to store the extracted information. By offering an *Application Programming Interface (API)*, it then should enable consumers to query this information on different levels of aggregation to answer questions that might not even have been known at the time the SCA was performed. A recent and popular example for such a question is "Which components contain log4j?".

This *Security and Compliance Data Lake* will build upon the *Open Component Model (OCM)*, an open standard to describe the SBOM with so called *Component Descriptors*. These *Component Descriptors* also describe how to access sources and resources. It thereby provides an entry point for the execution of SCA tools.

Contents

List of Figures

List of Tables

Acronyms

PoC	proof of concept
SCDL	Security and Compliance Data Lake

Chapter 1

Introduction

"[T]he trust we place in our digital infrastructure should be proportional to how trustworthy and transparent that infrastructure is"
Executive Order on Improving the Nation's Cybersecurity [1]

1.1 Motivation

The introductory quote above initially sounds pretty intuitive and self evident. Probably everyone in the IT industry would be able to agree on this. Yet, the panic and public outcry unleashed in the software industry after the vulnerability in Log4j was discovered, a popular and widely used logging library, goes to show how far the statement strives from reality. The vulnerability is rated with the highest severity possible since it enables *Remote Code Execution (RCE)*, which in other words allows an attacker to run any code on the machine using Log4j [2]. After the discovery, IT specialists all over the world had to identify which of their applications and systems were using vulnerable versions of the library before they could even start dealing with the vulnerability. This increases the response time and subsequently the risk exposure significantly. But what is it, that makes this such a difficult task?

Modern software systems are composed of numerous software components, such as Log4j, and these components may themselves be composed of other software components and so on. This adds several layers of complexity obfuscating the dependencies of a software system. The manufacturing industry has been dealing with such issues in their supply chains for years. But these companies usually maintain close relationships and contracts with their suppliers, perhaps even exchanging *bills of materials (BOM)*. Thus, the companies can easily keep track of each individual part in their products by accumulating the BOMs of their suppliers and they will even be informed about problems affecting this parts by their suppliers. This is different from the software industry. Especially when it comes to *open source software (OSS)*, companies do not have a contract with the supplier. On the contrary, they frequently might not even know the maintainer of the software component. Subsequently, they also will not be informed about issues such as vulnerabilities with the components.

The software industry has come up with approaches to deal with this issue and implemented measures to proactively monitor applications and detect known vulnerabilities in its components. With growing complexity of software and changing development and deployment landscapes, these approaches are not sufficient and fail to answer questions such as which systems use vulnerable versions of Log4j. Why and how they fail in these situations will be further examined in the course of this thesis. But as "[s]oftware is eating the world"[3] and thus, as companies, devices of our everyday life, cars and even medical and military devices rely on software, as the impact of software on our privacy and also physical security increases, so does the responsibility of companies providing software and the gap between trust and transparency becomes less acceptable.

In an effort to address this issue, the US government has also published an Executive Order on Improving the Nation's Cybersecurity, which will require every company supplying software to the government to provide an *Software Bill of Materials (SBOM)* [1, 4]. As previously discussed, this is only possible if every company in the supply chain provides an SBOM for their software components. Therefore, the downstream impact of this Executive Order will most likely affect the entire industry.

1.2 Goals

The goals of this thesis aim to provide a research based practical contribution to closing the gap between the trust and transparency placed into our digital infrastructure [1].

To make this more concrete, the main goal of this thesis is to develop a *proof of concept (POC)* implementation of a *Security and Compliance Data Lake(SCDL)*. As the name already suggests, this is an application for storing information about software components. Such information might be about the occurrence of known vulnerabilities, as already indicated in the previous paragraphs, but also about licenses or dependencies. This kind of information about software components will be called metadata from here on.

Thus, the goal is to design and develop a central application for storing and querying software metadata, which is able to cope with the complexity and requirements of modern development and deployment landscapes. Therefore improving the transparency of the entire software supply chain and enabling companies to answer questions such as which applications, systems or even landscapes in a company contain a certain vulnerability.

Thereby, this thesis also contributes to the knowledge concerning metadata stores and respective API design by answering following research question:

> *What are the challenges arising during the development of a database application for the central metadata management of enterprise software systems and how may these challenges be solved?*

Furthermore, the application may support companies in fulfilling the requirements resulting from the recently announced Executive Order.

1.3 Scope

As the goal section already stated, this thesis is about designing and implementing a central application for software metadata management and its integration into modern development and deployment landscapes.

It is not about developing new ways of vulnerability detection. It is rather about monitoring systems and keeping track of already known vulnerabilities. This is just as important since a large amount of security incidents are caused by attackers exploiting such known vulnerabilities [5]. Therefore, neither does this thesis compete with current security testing techniques which are used to find vulnerabilities in applications [6], nor does it discuss these in detail. From the perspective of the central metadata store, each security testing technique applied may be viewed as a potential data source. But storing known vulnerabilities is still just one, although probably the most popular, use case of this central metadata store. A major design goal was to keep it extensible, so that all kinds of metadata may be stored.

1.4 Environment

This thesis is written in cooperation with SAP. As of today, with a total revenue of €27.34 billion, SAP is the third largest software company in the world after Microsoft and Oracle [7]. While also having gained some attention with *Business to Customer (B2C)* products like the *Corona-Warn-App*, its core products are *Business to Business (B2B)* enterprise software solutions. Originally, SAP grew around its *Enterprise Resource Planing (ERP)* system. Today, the company is also adopting *Internet* *of Things (IoT)* technologies and *Artificial Intelligence (AI)* to provide advanced analytics for its customers and to maximize the value of its software products [8]. Besides, SAP is also investing heavily to push its products and customers to the cloud. Therefore, the company maintains partnerships with Amazon, Microsoft, Google and Alibaba as infrastructure providers.

The department this thesis is written with is developing and maintaining the *SAP Gardener*. SAP Gardener is *SAP's own managed Kubernetes service* which enables SAP itself as well as SAP customers to ship their applications to all of these different infrastructures using a unified deployment underlay. Since SAP Gardener offers the possibility to configure practically every detail, neither SAP nor its customers need to rely on the managed Kubernetes service of each hyperscaler with their individual perks and restrictions, but can leverage the functionality of a fully configurable kubernetes cluster without actually having to fully configure it [9].

1.5 Structure of the Thesis

In order to achieve the goals laid out before, the thesis builds upon following structure:

Foundation: In this chapter, basic terminology and existing concepts in software metadata management are established.

State of the Art: In the previous motivation and goal sections, it is already mentioned that existing measures for monitoring software component metadata such as vulnerabilities are insufficient in some cases. Therefore, to further motivate this thesis, this chapter will briefly introduce the state of the art approach and point out its limitations. The identification of this problems is crucial for the successful design of the Security and Compliance Data Lake.

Context at SAP Gardener: As mentioned, this thesis is written with the SAP Gardener team. It is greatly inspired by the problems they are facing and by the approaches they already implemented to overcome limitations of the state of the art. Thus, this chapter introduces their proposed standard, the Open Component Model, and its capabilities. Based on that, their development and deployment landscape is explained. Finally, an architecture for integrating a central metadata store such as the one designed and developed in this work into their current environment and thereby overcoming all these limitations is presented.

System Design: The system design chapter covers the conception of the data model, the selection of a database, the overall application architecture and the design of the API. Thereby, it especially focuses on giving detailed information about the ideas and motives that lead to specific design decisions. Besides, alternatives that have been considered and the respective reasons to not follow through with them are an essential part of this section.

Results: Finally, the actual functionality of the application is showcased and the design decisions, the implementation as well as the knowledge gained through this work is evaluated and the research question is revisited.

Chapter 2

Foundations

This chapter provides the basic knowledge necessary to understand the following chapters as well as other research and literature regarding metadata management. It therefore explains several important concepts and abbreviations around *Software Identification*, *Open Source*, *Open Source Licensing* as and *Vulnerability Management*. Furthermore *Software Bill of Materials* and existing *Software Bill of Material Standards* are introduced. This is important to understand the requirements of the Executive Order mentioned in the introduction and to put everything into context.

2.1 Software Identification

Uniquely identifying a piece of software is a frequent concern when managing applications on enterprise scale. Thus, a standardized naming convention would be required. Unfortunately, pretty much every packet manager and tool uses its own convention.

Package URL (purl)

Package URL, or purl for short, is one of the most widely adopted attempts to standardize those existing approaches. As described in the projects GitHub Repository, it therefore specifies an URL string, a purl, which should be able "to identify and locate a software package in a mostly universal and uniform way across programming languages, package managers, packaging conventions, tools, APIs and databases" [10]. This Package URL consists of seven components [10]:

```
scheme:type/namespace/name@version?qualifiers#subpath
```

Listing 2.1: Package URL

```
pkg:docker/cassandra@sha256:244fd47e07d1004f0aed9c
pkg:github/package-url/purl-spec@244fd47e07d1004f0aed9c
pkg:deb/debian/curl@7.50.3-1?arch=i386&distro=jessie
pkg:npm/foobar@12.3.1
```

Listing 2.2: Package URL Examples

Each component is separated by a different specific character to allow for unambiguous parsing. The `scheme` (Required) is the constant value "pkg" which may be officially registered as an URL scheme in the future. `type` (Required) refers to a package protocol such as maven or npm. `namespace` (Optional) may be some type-specific prefix such as a Maven groupid, a Docker image owner or a Github user or organization. The `name` (Required) and `version` (Optional) are the name and version of the software. `qualifiers` (Optional) is also type-specific and may be used to provide extra qualifying data such as an OS, architecture or a distribution. With the `subpath` (Optional) one may specify a subpath within a package, relative to the package root [10].

Another convention to uniquely identify packages is the *Common Package Enumeration (CPE)* format which is primarily used in the context of vulnerability management ecosystem. Therefore, the standard will be discussed in the following section.

2.2 Open Source Software and Licensing

Open Source Software is widely used in modern software development. As to what Open Source Software actually is, the *Open Source Initiative* defined a set of rules, the *Open Source Definition* specifying distribution terms that the license of a software must comply with. Without going into detail and examining all of these terms, this definition generally ensures that such software "can be freely accessed, used, changed, and shared (in modified or unmodified form) by anyone" [11].

This description on its own may give the impression that there is no need to keep track of what *OSS Licenses* are used within an enterprise application. But there are still some limitation that *OSS Licenses* can put to the usage, especially the distribution of the software. Specifically, the so called *copyleft* principle and corresponding licenses might pose a challenge to some companies. While OSS generally may be used for commercial purposes, this principle dictates that if a company or individual distributes the software or a derivative, it has to be done under the same license it has been received under [11].

In the context of this work, OSS will be used within other OSS components, so *copyleft* is not an issue. But anyway, there is still a risk involved in using OSS. A company might distribute initial software versions under an OSS License until there is some kind of customer lock-in and then switch to another more restrictive license for further versions. Also, since there might not exists precedent cases regarding the legal interpretation of some specific OSS licenses, there is a risk of expensive law suits.

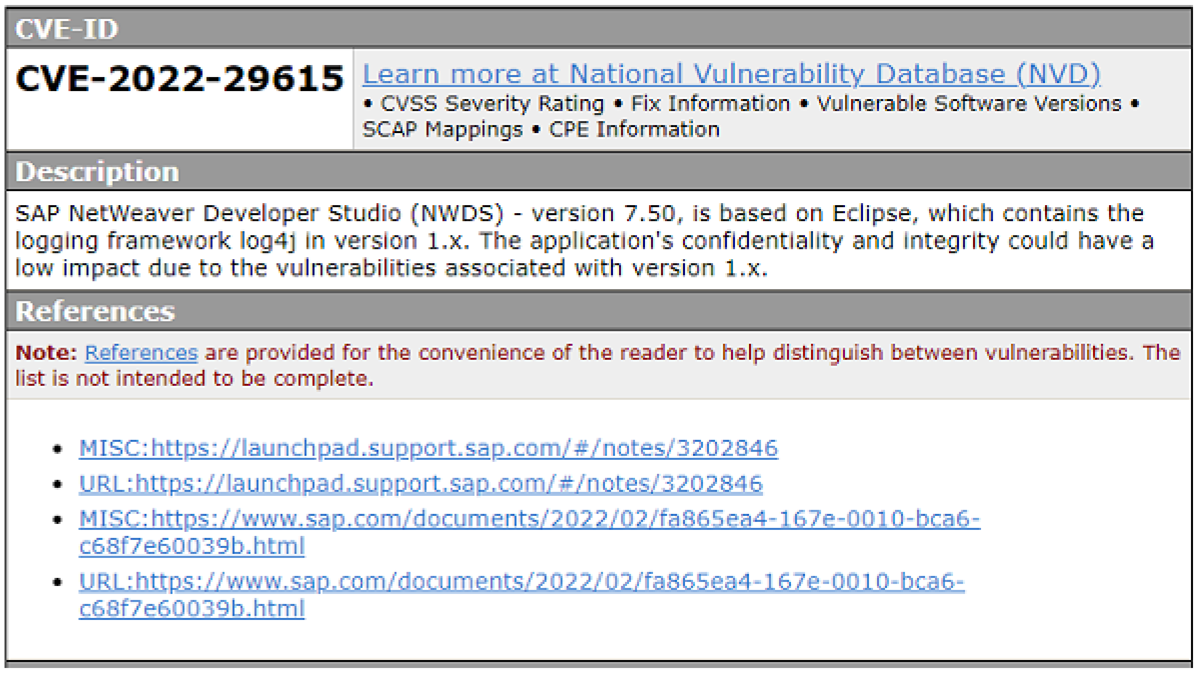
2.3 Vulnerability Management

Due to the widespread use of OSS, vulnerability management is an increasingly important topic. Since many organizations use the same software components, vulnerabilities often become publicly known. Also, the open source nature allows attackers to get precise information about the vulnerability itself. Thus, tracking publicly known vulnerabilities in an organizations software products is a crucial capability to keep them secure.

Vulnerability
According to the National Institute of Standards and Technology (NIST) a vulnerability is “A weakness in the computational logic (e.g., code) found in software and hardware components that, when exploited, results in a negative impact to confidentiality, integrity, or availability. Mitigation of the vulnerabilities in this context typically involves coding changes, but could also include specification changes or even specification deprecations (e.g., removal of affected protocols or functionality in their entirety).”[12]

Common Vulnerabilities and Exposures (CVE)
CVE was created by MITRE in 1999. Initially, the acronym was meant to stand for *Common Vulnerability Enumeration*. As described by the authors of the original whitepaper, many security tools and advisories used their own vulnerability identifiers. In order to integrate several of these, one had to manually compare and eventually relate the vulnerabilities to each other. Thus, a common naming convention and common enumeration of vulnerabilities was needed [13].

To solve this issue, the *CVE Program* assigns unique identifiers, so called *CVE IDs*, to each vulnerability. All the identified vulnerabilities are then maintained as *CVE Records* in the *CVE List*. A minimal *CVE Record* consists of a *CVE ID*, a description of the vulnerability and at least one public reference. It does not include technical data, information about risks, impacts or fixes. An example for such a CVE Record is shown in figure 2.1 below.



CVE-ID	
CVE-2022-29615	Learn more at National Vulnerability Database (NVD) • CVSS Severity Rating • Fix Information • Vulnerable Software Versions • SCAP Mappings • CPE Information

Description

SAP NetWeaver Developer Studio (NWDS) - version 7.50, is based on Eclipse, which contains the logging framework log4j in version 1.x. The application's confidentiality and integrity could have a low impact due to the vulnerabilities associated with version 1.x.

References

Note: References are provided for the convenience of the reader to help distinguish between vulnerabilities. The list is not intended to be complete.

- MISC:https://launchpad.support.sap.com/#/notes/3202846
- URL:https://launchpad.support.sap.com/#/notes/3202846
- MISC:https://www.sap.com/documents/2022/02/fa865ea4-167e-0010-bca6-c68f7e60039b.html
- URL:https://www.sap.com/documents/2022/02/fa865ea4-167e-0010-bca6-c68f7e60039b.html

Figure 2.1: CVE Record
Source: Based on [14]

In order to make sure that all vulnerabilities listed in the *CVE List* are unique and maintained properly, *CVE IDs* can only be assigned and *CVE Records* can only be published by MITRE and several partner organizations, so called *CVE Numbering Authorities (CNA)*. Thus, to add a new vulnerability to the *CVE List*, the discoverer has to report it to a CNA [15].

Common Vulnerability Scoring System (CVSS)

CVSS is an open framework for rating vulnerabilities. It is owned and managed by the non-profit organization *Forum of Incident Response and Security Teams (FIRST)*. The framework captures the main characteristics of a vulnerability to produce a numerical score between 0.0 and 10.0 reflecting its severity.

Therefore CVSS is composed of three metric groups: Base, Temporal, and Environmental. The *Base Score* considers the intrinsic characteristics of a vulnerability that are constant over time and assumes the worst case impact across different environments. These characteristics take into account exploitability metrics like attack complexity but also impact metrics like confidentiality impact. The *Base Score* is typically calculated by the organization maintaining the vulnerable product or by third party analysts on their behalf. The *Temporal Score* considers characteristics that may change over time but not across environments. Such a characteristic would be the maturity of exploit code. The *Environmental Score* considers the relevance of a vulnerability in a specific environment. *Temporal* and *Environmental Scores* are typically calculated by the consumers of the components to adjust the vulnerability rating to their organizations use case and environment. These Scores can then be used for internal risk management [16].

Common Weakness Enumeration (CWE)

CWE is a community-developed list of software and hardware weaknesses maintained by MITRE. Each weakness in the least is assigned a *CWE ID*. The list represents weaknesses on different levels of abstraction. This is conceptually shown by figure 2.2.

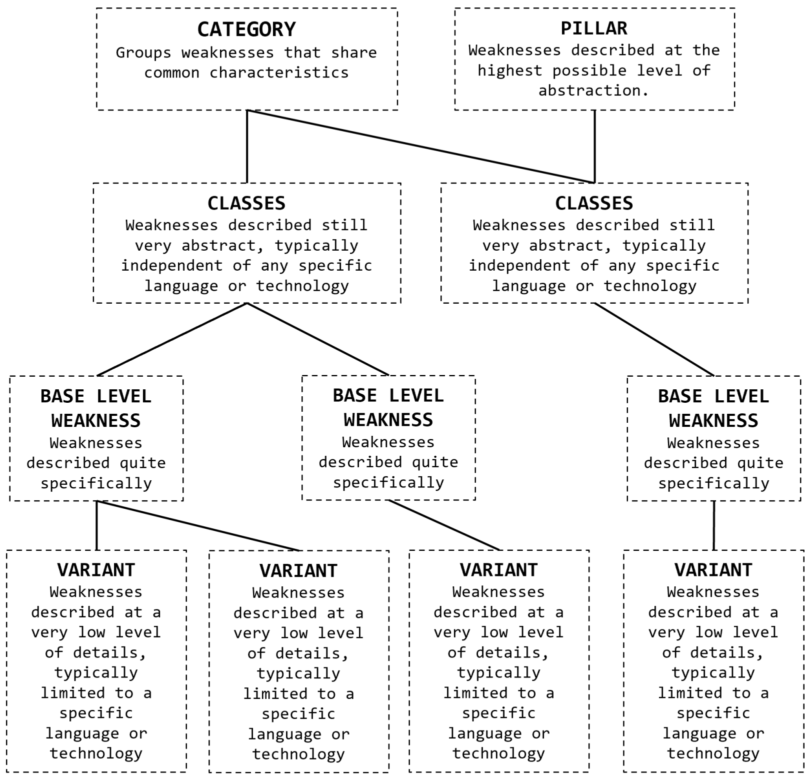

CATEGORY
Groups weaknesses that share common characteristics
PILLAR
Weaknesses described at the highest possible level of abstraction.
CLASSES
Weaknesses described still very abstract, typically independent of any specific language or technology
CLASSES
Weaknesses described still very abstract, typically independent of any specific language or technology
BASE LEVEL WEAKNESS
Weaknesses described quite specifically
BASE LEVEL WEAKNESS
Weaknesses described quite specifically
BASE LEVEL WEAKNESS
Weaknesses described quite specifically
VARIANT
Weaknesses described at a very low level of details, typically limited to a specific language or technology
VARIANT
Weaknesses described at a very low level of details, typically limited to a specific language or technology
VARIANT
Weaknesses described at a very low level of details, typically limited to a specific language or technology
VARIANT
Weaknesses described at a very low level of details, typically limited to a specific language or technology

Figure 2.2: CWE Hierarchy
Source: Own Representation

So there exist relationships between elements on different levels of abstraction. As shown *Classes* are usually member of a *Category* and might also be the child, hence a more concrete description, of a *Pillar*. Additionally to what is shown in the figure, these relationships may also skip levels of hierarchy. Thus, a *Base Level Weakness* may be a direct member of a *Category* or a child of a *Pillar*. An example for a Base Level Weakness is*CWE*-478*: Missing Default Case in Switch Statement* which is a member of *Bad Coding Practices* and a child of the class *Incomplete Comparison with Missing Factors*, which is a child of the pillar *Incorrect Comparison* [17].

Common Platform Enumeration (CPE): The CPE specification originally created by MITRE and now maintained by NIST provides a naming scheme for IT assets such as software. It may be used to uniquely determine a specific software and its version. This way a CPE enables cross referencing to other sources of information. The commonly used CPE naming scheme is structured as follows:

```
cpe:2.3: part : vendor : product : version : update : edition :
   language : sw_edition : target_sw : taget_hw : other
```

Listing 2.3: CPE Formatted String Binding

Thereby `part` may be *a* for applications, *o* for operating systems, and *h* for hardware devices. `edition` is a legacy attribute in the current version of the specification and may be omitted where not required for backward compatibility. The attributes after `edition` were newly introduced in this version and are referred to as *extended attributes*. `sw_edition` should characterize a particular market or class of users a product is tailored to (e.g. online), `target_sw` a software computing environment (e.g. linux), `target_hw` the instruction set architecture (e.g. x86), and `language` the language supported in the user interface [18].

National Vulnerability Database (NVD)

The NVD is a database of vulnerabilities owned and maintained by NIST. In the paragraph about CVE, it was mentioned that the *CVE Records* do not contain technical data, information about risks and impact, or fixes. The NVD feeds from the *CVE List* and uses the information provided in the *CVE Records* to perform further analysis. As a result, a NVD entry exists for each *CVE ID* and provides a *CVSS Base Score*, a *CWE ID* and a *CPE ID* [12].

Thus, NVD combines all the aforementioned standards and concepts to provide thorough and concise human and machine-readable information about vulnerabilities. *CPE IDs* identifying a particular software version in use may be queried against a NVD API to automatically check for known vulnerabilities. The *CVSS Base Score* is a valuable foundation for internal risk assessment and the *CWE ID* helps to quickly understand the type of a vulnerability.

2.4 Software Bill of Materials

A *Software Bill of Materials* is an inventory of the components used in a software. It ideally contains all direct and transitive components and their dependencies, so it is in other words pretty much the dependency graph of a software [19, 4].

As consequence to an *Executive Order on Improving the Nation's Cybersecurity*, the *National Telecommunications and Information Administration (NTIA)* published a document describing the minimum requirements for SBOMs [1, 4]). According to this document, these are:

Data Fields (Metadata)	Baseline information about each component: Supplier, Component Name, Version of the Component, Other Unique Identifiers, Dependency Relationship, Author of SBOM Data, Timestamp of SBOM creation
Automation Support	Automatic generation and machine-readability to allow for scaling across the software ecosystem.
Practices and Processes	Implementation of policies, contracts and arrangements to maintain SBOMs.

Table 2.1: Minimum Elements of a SBOM
Source: [4]

The goal of the *Data fields* is to sufficiently identify the components to track them through the supply chain and map them to other data sources, such as vulnerability and license databases. The *Automation Support* provides the ability to scale across the software ecosystem. The *Practices and Processes* ensure the maintenance by integration into the ALM. SBOMs thereby increase software transparency, providing those who produce, purchase and operate software the means to perform proper risk assessments [4].

Due to this Executive Order, SBOMs are now required for all U.S. federal software procurements. This does not only affect direct software vendors of the U.S. government [1]. As a consequence to this Executive Order, every organization that is downstream from the U.S. government in the supply chain may be required to provide SBOMs for its products. Thus, this will be a crucial capability for most software vendors.

There are three data formats mentioned in the minimum elements document which are interoperable, able to fulfill the requirements and either human- and machine-readable. Those are the *Software Package Data eXchange (SPDX)*, *CycloneDX* and *Software Identification (SWID)* tags [4]. SAP uses yet another SBOM format, called Open Component Model (OCM), which does not fulfill the minimum requirements. The OCM will be discussed further in one of the following sections. SPDX is the most mature standard. It has laid out a lot of groundwork for the more recent CycloneDX. Thus, to get an better and more concrete understanding of SBOMs, SPDX will be examined in more detail.

SPDX License List and License Identifiers

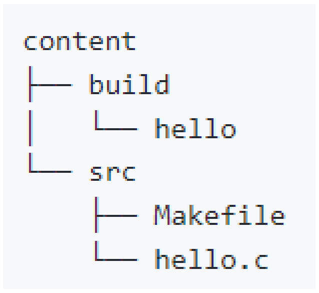
SPDX is an initiative founded in 2010 and hosted at *The Linux Foundation*. In 2021 the SPDX specification even became an ISO standard [20]. The initiative focuses on solving challenges regarding the licenses and copyrights associated with software packages. SPDX therefore assembles licenses and exceptions commonly found in OSS in the *SPDX License List*. More precisely, this list includes a standardized short identifier, the full name, the license text, and a canonical permanent URL for each license and exception. By incorporating this *SPDX License Identifiers* in source on file level, one enables automation of concise license detection, even if just parts of an OSS project are used. Furthermore, SPDX provides *Matching Guidelines* to ensure that e.g. a "BSD 3-clause" license in a LICENSE file of an OSS project with different capitalization or usage of white space than the master license text included in the *SPDX License List* is still identified as "BSD 3-clause" license.

SPDX Documents

At the heart of the SPDX initiative are the *SPDX Documents* which leverage the *SPDX License List* and *SPDX License Identifiers* to describe the licensing of a set of associated files, referred to as *Package* in the context of SPDX. A *SPDX Document* provides means to describe information about the document creation, the package as a whole, individual files, snippets of code within an individual file and other licenses that are not contained in the *SPDX License List* but are still relevant for the package, relationships between *SPDX Documents*, and annotations, which in a way are comments within an *SPDX Document*. The concept of relationships is a rather new addition to the specification. It is particularly useful if one has an SPDX Document describing a binary. Explicitly capturing relationships like "generated from" these source files and "dynamically linking" these libraries allows for a complete licensing picture.

These documents may be represented in one of the following five file format: tag/value (.spdx), JSON (.spdx.json), YAML (.spdx.yaml), RDF/xml (.spdx.rdf), and spreadsheets (.xls) [21, 22].

To give a more concrete idea of the basic concepts of *SPDX Documents*, an example from the SPDX GitHub repository will be briefly examined [23]. Therefore figure 2.3 below shows the directory structure of a “Hello World” project in C.



```
content
├── build
│   └── hello
└── src
    ├── Makefile
    └── hello.c
```

Figure 2.3: C Project Directory Structure
Source: [23]

Listing 2.4 shows a corresponding *SPDX Document*. Some tag:value pairs which are less relevant for the overall understanding are deliberately omitted to contain the length of the example.

```
SPDXVersion: SPDX-2.2
DataLicense: CC0-1.0
SPDXID: SPDXRef-DOCUMENT
DocumentName: hello
DocumentNamespace: https://swinslow.net/spdx-examples/example1/hello-v3
Creator: Person: Steve Winslow (steve@swinslow.net)
Created: 2021-08-26T01:46:00Z

##### Package: hello
PackageName: hello
SPDXID: SPDXRef-Package-hello
PackageDownloadLocation: git+https://github.com/swinslow/spdx-examples.git#example1/content
PackageLicenseConcluded: GPL-3.0-or-later
PackageLicenseInfoFromFiles: GPL-3.0-or-later
PackageLicenseDeclared: GPL-3.0-or-later
PackageCopyrightText: NOASSERTION

FileName: /build/hello
SPDXID: SPDXRef-hello-binary
FileType: BINARY
LicenseConcluded: GPL-3.0-or-later
LicenseInfoInFile: NOASSERTION
FileCopyrightText: NOASSERTION

FileName: /src/Makefile
SPDXID: SPDXRef-Makefile
FileType: SOURCE
LicenseConcluded: GPL-3.0-or-later
LicenseInfoInFile: GPL-3.0-or-later
FileCopyrightText: NOASSERTION

FileName: /src/hello.c
SPDXID: SPDXRef-hello-src
FileType: SOURCE
LicenseConcluded: GPL-3.0-or-later
LicenseInfoInFile: GPL-3.0-or-later
FileCopyrightText: Copyright Contributors to the spdx-examples project.

Relationship: SPDXRef-hello-binary GENERATED_FROM SPDXRef-hello-src
```

```
Relationship: SPDXRef-hello-binary GENERATED_FROM SPDXRef-Makefile
Relationship: SPDXRef-Makefile BUILD_TOOL_OF SPDXRef-Package-hello
```

Listing 2.4: SPDX Document

Most of the tag:value pairs are self-explanatory, but some might require some explanation. The *Concluded License* is the license the SPDX file creator has concluded as the governing license of a package or a file. *License Information from Files* contains a list of all licenses found in a package and the *Declared License* is the license declared by the authors of the package [22]. Additionally, listing 2.4 illustrates how the concept of relationships may be used.

It is also worth mentioning that the concept of *Packages* in SPDX as a set of associated files is really rather loose. Thus, describing the project in figure 2.3 as two separate packages, one for source and one for binary, optionally in the same or also in two separate *SPDX Documents* would be completely conform with the specification as well.

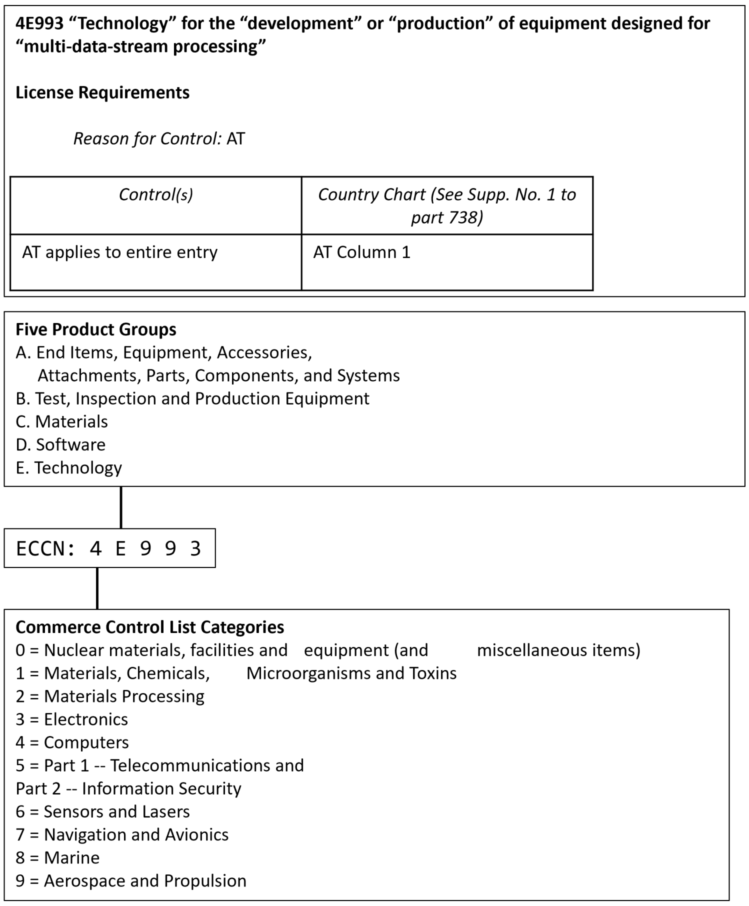
Since SPDX has been around for so long and is an accepted ISO standard, there exists a lot of useful tooling. It is therefore quite easy to automate tasks like producing, consuming, transforming and validating *SPDX Documents* [21].

2.5 Regulations

Vulnerabilities and licenses are not the only risks associated with enterprise software. Companies also need to consider governmental regulations since violations may lead to fines that have critical impact on the business.

Export Control Classification Number (ECCN)

The ECCN is a 5 character alpha-numeric designation used to determine whether an so called dual-use item needs an export license from the U.S. Department of Commerce in order to legally export it. Dual-use items are items that may be used for civil as well as military purposes. The ECCN gives some information about the product, as shown in figure 2.4 below.



4E993 "Technology" for the "development" or "production" of equipment designed for "multi-data-stream processing"

License Requirements

Reason for Control: AT

Control(s)	*Country Chart (See Supp. No. 1 to part 738)*
AT applies to entire entry	AT Column 1

Five Product Groups
A. End Items, Equipment, Accessories, Attachments, Parts, Components, and Systems
B. Test, Inspection and Production Equipment
C. Materials
D. Software
E. Technology

ECCN: 4 E 9 9 3

Commerce Control List Categories
0 = Nuclear materials, facilities and equipment (and miscellaneous items)
1 = Materials, Chemicals, Microorganisms and Toxins
2 = Materials Processing
3 = Electronics
4 = Computers
5 = Part 1 -- Telecommunications and Part 2 -- Information Security
6 = Sensors and Lasers
7 = Navigation and Avionics
8 = Marine
9 = Aerospace and Propulsion

Figure 2.4: ECCN Structure
Source: Based on [24]

All ECCNs are listed in the *Commerce Control List (CCL)*. The entries in the list contain information about why an item might be under export control regulations. In the top of figure 2.4 is a snippet of such an entry. The reason for the export regulations of products with the classification number 4E993 is AT, which is the abbreviation for Anti-Terrorism [24].

Chapter 3

State of the Art

This chapter gives an overview of the state of the art approach of integrating security and compliance measures into the *software development life cycle (SDLC)*. That includes a brief introduction of practices such as *Continuous Integration and Continuous Delivery (CI/CD)* and *DevOps*. In the end, the limitations of this current approach are discussed, thereby further motivating this thesis.

3.1 Software Development Life Cycle

Since the emerging of cloud technologies, the software industry, especially also the major software companies like Microsoft, Amazon and SAP shifted their business model from *software as a product (SaaP)* to *software as a service (SaaS)*. This gave companies the opportunity to frequently release updates without adhering to a rigid distribution cycle. So they could react to feedback much faster and improve their software continuously [25, 26].

To keep up with this new pace, the SDLC had to be adjusted as well. So developers started to *continuously integrate (CI)* their code after they conducted some changes while at the same time automatically testing the software. Thus, always keeping it up to date and in a shippable state.

As an extension of that idea, the software is also automatically and *continuously delivered (CD)* to testing or even production environments, accelerating the process even more [27].

A quick side note - also after reviewing some literature, there seems to be some disagreement whether to distinguish continuous delivery and continuous deployment [27, 25]. The articles and papers that do differentiate define continuous delivery as automatically delivering to production-like environments for evaluation and testing. But there are still some manual steps necessary to actually deploy into production or customer environments [27, 28, 29].

DevOps, a combination of "development" and "operations", is a practice which also resulted from this change of the SDLC. Since the concept of CI/CD merges areas of development and operations, the corresponding roles were combined to reduce communication overhead and misunderstandings [29]. DevOps is usually reliable for the setup and maintenance of the *CI/CD Pipeline*. Hence, they automate the steps required for the continuous integration and delivery, such as building and testing the software after a developer integrated his code changes and moving it to the next step after the previous tests and checks are passed.

3.2 Integrating Security and Compliance Measures

In practice, in order to conduct CI/CD, DevOps usually sets up a so called *CI/CD Pipeline* for the project. This includes a set of tools to automate build, test and deployment of the software. The most popular and widely used example is Jenkins [30]. But essentially, CI/CD pipelines are just simple stage-gate systems. Thus, the process is divided into a number of stages. Between each stage is a quality gate, such as a set of automated tests. The software has to pass this quality gate in order to being able to move to the next stage [31].

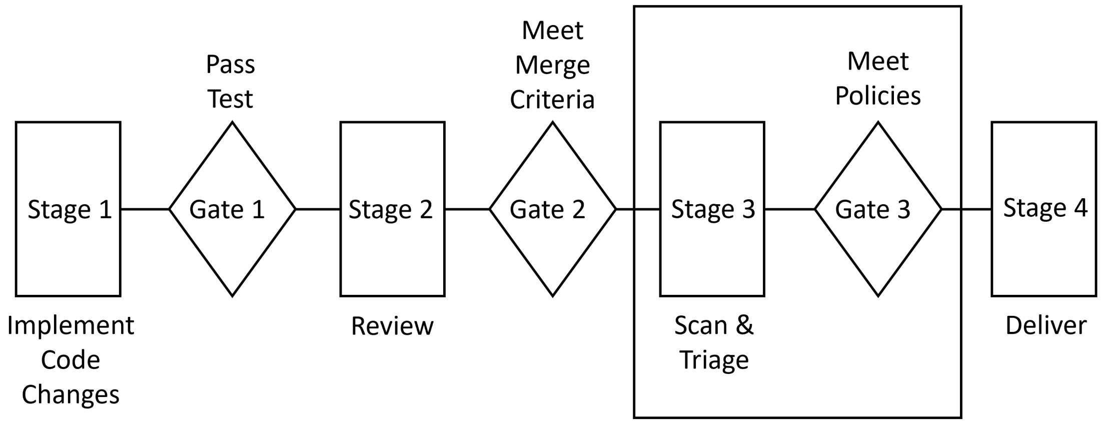


Figure 3.1: CI/CD Pipeline as Stage-Gate System
Source: Based on [31]

Figure 3.1 is an abstract technology agnostic illustration of a CI/CD pipeline as a stage-gate system. This also shows the currently established state of the art approach to reduce software supply chain risks such as problematic licenses or vulnerabilities in dependencies. Security and compliance scans are conducted as part of the CI/CD pipeline. The results of these scans may also be triaged before they are finally checked against a defined set of policies, forming another quality gate *before delivery*.

Triage is a term derived from medicine where it describes the classification and prioritization of patients if there are not enough resources to treat them all immediately. In software testing or scanning, it refers correspondingly to the process of (re-)rating and (re-)classifying issues found during the scan in the context of its occurrence within the particular project. Common issues include problematic licenses or vulnerabilities. For the latter, the triage may be done based on the CVSS introduced in the previous chapter. Policies may then prevent shipping of software that contains vulnerabilities with a CVSS that exceeds a certain threshold or specific blacklisted licenses.

Unfortunately, there is not a lot of academic literature on this specific topic to back up the claim that this is the state of the art approach. Therefore, for the purpose of this work, the offerings of the major vendors of corresponding tools, hence *Software Composition Analysis (SCA)* tools were deemed as a suitable source. SCA tools usually scan source code repositories or binary files, analyze the dependencies and match them against vulnerability or license databases.

According to a a market research of Forrester from 2021, based on a combination of market presence, current offering and strategy, these are Mend (formerly known as WhiteSource), Synopsys, Sonatype and Snyk [32]. All of these tools advertise their smooth integration into the CI/CD pipeline, to be more precise into the version control systems and build tools, as another quality gate [33, 34, 35, 36]. This should in general be a suitable representation of what the industry is requesting, although it is to mention that most of these vendors also offer other integration options. Another popular one for the SCA tools that scan binary files are the artifact repositories.

Additionally, it is undeniable that it makes a lot of sense to integrate these scans as another quality gate as part of the CI/CD pipelines into the repositories. At this point, all the files of a project are easy accessible for these tools. Besides, it is a commonly known principle in process management, that a defect becomes more expensive the later it is found.

3.3 Producing Software Bill of Materials

As a solution to increasing the supply chain transparency, the US government decided that analogous to other industries their software vendors have to provide SBOMs. Subsequently, the NTIA conducted research and published several documents on the topic. One of those being the minimum elements for SBOMs discussed in the foundations chapter. As mentioned there, one of the minimum requirements is automation support. Therefore, the NTIA conducted a survey of existing SBOM formats and standards considering how to integrate them into the SDLC [37].

In this document the NTIA also suggests to leverage existing tools such as version control systems, build tools, code scanners and binary analysis tools to generate SBOMs [37]. Thus, the tools already integrated into the CI/CD pipelines as established in the previous paragraphs. Since these SCA tools obtain some of the most important SBOM information like dependencies and licenses anyway, they can also provide these results in a specific SBOM format. Therefore, all of the previously mentioned SCA tool vendors quickly adjusted and do provide such options now.

3.4 Limitations

This approach is intuitive and might be sufficient for the scope of a single development team which deploys its software to a single infrastructure. But the limitations really start to show if this scope is exceeded.

While the quality gate approach is not wrong, it is insufficient. Once the software passed the corresponding quality gate, there are no more scans. But new vulnerabilities in one of the dependencies may still be discovered after this point in time. These would then go undetected until the respective quality gate is reached again with the next version of the software. Besides, the main software that is being developed might depend on additional third party software at runtime. Consequently, this third party software has to be delivered alongside the main software. But the third party software might never pass any of the companies CI/CD pipelines and thus never pass the quality gates. Therefore, it would never be scanned and its vulnerabilities would again go undetected. Furthermore, with the quality gate approach, the scans are frequently also treated as such. Thus, developers hope for their code to pass the scan and discard the results of it does. Finally, with this approach, each development team has to configure, integrate and maintain this scanning tools on their own in each of their pipelines. In larger companies, this often also leads multiple opinionated CI/CD pipelines with additional point to point integration for such compliance tools and individual reporting dashboards as an attempt to overcome the existing limitations of this approach.

To solve these issues, the *compliance scans have to be decoupled from the CI/CD pipeline.* A quite popular solution to this is to integrate the them with the artifact repositories. Thereby, even vulnerabilities in components that do not pass an internal CI/CD pipeline are detected. But of course this only works with the binary scanners, which usually provide less precise results. Besides, this might also get tedious and end up in multiple point to point solutions in cases where the artifacts are distributed over several different artifact repositories. Another possible option is based on the production of SBOMs during the compliance scans. When initially passing the quality gate, an SBOM can be generated, which may be used as access point to conduct further compliance scans. The tools of Mend and Synopsys already provide such an option. The problem here is, that the components in the generated SBOMs usually have tool specific identifiers. Therefore, each tool can only conduct scans based on their own SBOM. So both of this approaches are not ideal, and even if they were, there would still be a problem.

One might imagine a situation where the artifacts, which comprise the final product and are built from different internal as well as third party repositories, are stored in multiple different artifact repositories. From these repositories the artifacts are deployed into multiple different production environments. But naturally, there are different versions of the final product which are comprised of different versions of the artifacts. Given a scan detects a vulnerability in a specific version of an artifact in the artifact repository, how would one go about telling which environments contain this particular version of the artifact? Of course, the information about which product version contains what version of the artifacts is stored somewhere, the SBOMs for example. And the information where this product version is deployed is most certainly also stored somewhere else. But ultimately, there is no easy way to answer this question without skimming through this different data sources. This concludes the final and central issue.

Software metadata is distributed over the entire software development life cycle. Thus, there is a need for a central application for storing and querying software metadata.

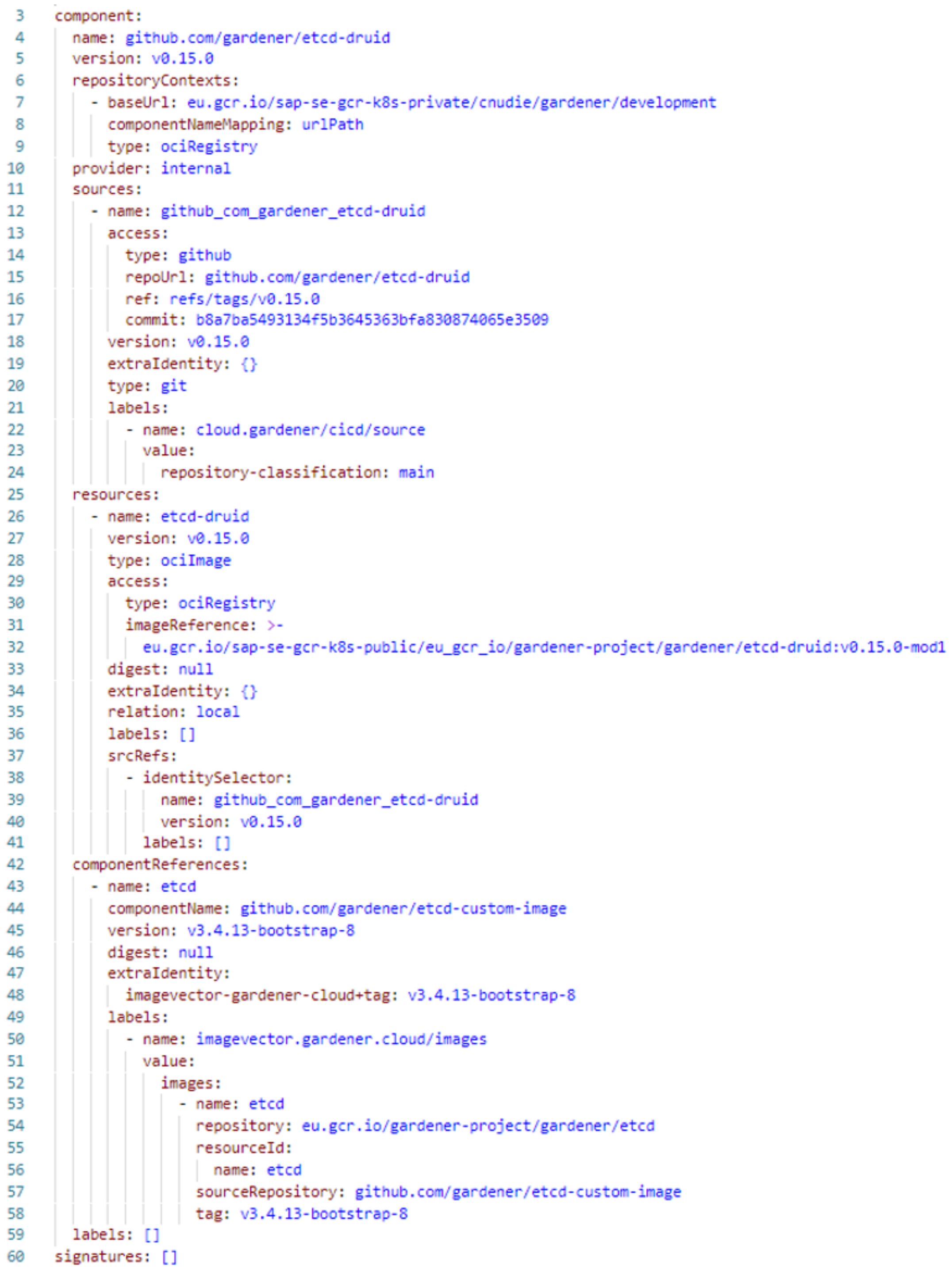
Chapter 4

Context at SAP Gardener

As mentioned in the introduction, this work is written in cooperation with SAP. In fact, the limitations mentioned in the previous chapter are deducted from the limitations the SAP Gardener team struggles with themselves. Subsequently, this chapter introduces the *Open Component Model (OCM)*, SAP Gardeners proposed standard to decouple the compliance scans from the CI/CD pipeline. Based on this knowledge, an overview of the development and deployment landscape of SAP Gardener is given. Finally, the suggested integration of the Security and Compliance Data Lake, as a central application for storing and querying software metadata, with the existing standard and landscape is presented. Thereby, this chapter provides a reference architecture on how to overcome the major limitations of the current state of the art approach.

4.1 Open Component Model

The OCM is an SBOM format created and used by SAP Gardener. It does not fulfill the minimum requirements as defined by the NTIA. But this is due to the fact that the OCM has a different focus than SPDX or CycloneDX. While those two were deliberately designed to be a bill of materials, thoroughly listing the inventory of a software, the OCM was specifically developed to decouple CI from CD and thereby overcome related limitations such as the ones mentioned in the previous chapter. Following is a technical explanation of the OCM deducted from the specification [38] and an internal presentation [39]. Even though this explanation focuses on understanding the rationale behind the design decisions of the proposed standard rather than technical completeness, it may still be a little hard to grasp at times. This is due to the abstract nature of the OCM. Therefore, emphasis and examples are used where possible. Also, while the textual description really explains the OCM as the abstract model that it is, figure 4.1 below shows an actual *Component Descriptor*, the serialization format of the OCM. This may also support in understanding the abstract concepts.



```yaml
component:
  name: github.com/gardener/etcd-druid
  version: v0.15.0
  repositoryContexts:
    - baseUrl: eu.gcr.io/sap-se-gcr-k8s-private/cnudie/gardener/development
      componentNameMapping: urlPath
      type: ociRegistry
  provider: internal
  sources:
    - name: github_com_gardener_etcd-druid
      access:
        type: github
        repoUrl: github.com/gardener/etcd-druid
        ref: refs/tags/v0.15.0
        commit: b8a7ba5493134f5b3645363bfa830874065e3509
      version: v0.15.0
      extraIdentity: {}
      type: git
      labels:
        - name: cloud.gardener/cicd/source
          value:
            repository-classification: main
  resources:
    - name: etcd-druid
      version: v0.15.0
      type: ociImage
      access:
        type: ociRegistry
        imageReference: >-
          eu.gcr.io/sap-se-gcr-k8s-public/eu_gcr_io/gardener-project/gardener/etcd-druid:v0.15.0-mod1
      digest: null
      extraIdentity: {}
      relation: local
      labels: []
      srcRefs:
        - identitySelector:
            name: github_com_gardener_etcd-druid
            version: v0.15.0
          labels: []
  componentReferences:
    - name: etcd
      componentName: github.com/gardener/etcd-custom-image
      version: v3.4.13-bootstrap-8
      digest: null
      extraIdentity:
        imagevector-gardener-cloud+tag: v3.4.13-bootstrap-8
      labels:
        - name: imagevector.gardener.cloud/images
          value:
            images:
              - name: etcd
                repository: eu.gcr.io/gardener-project/gardener/etcd
                resourceId:
                  name: etcd
                sourceRepository: github.com/gardener/etcd-custom-image
                tag: v3.4.13-bootstrap-8
  labels: []
signatures: []
```

Figure 4.1: Component Descriptor
Source: Based on [38]

In the context of the OCM, a *Component* is a software intended for a purpose which is identified by a *globally unique name*. This definition is still pretty vague. But this is on purpose and it will become clear why throughout this section.
A *Component Version* is identified by the *globally unique name* of the corresponding *Component* and a *version*. A *Component Version* may contain *Component References* and *Artifact References*. Furthermore, it has a *Repository Context*, a *Labels* and a *Provider* property. *Repository Context* provides a formal description of the repositories that store a representation of this *Component Version* and how to access them. *Labels* is really just a container for additional metadata. It therefore is an array of objects with the properties *name* and *value*. The *name* is a string while *value* may be a string, array or map, arbitrarily nested. *Provider* specifies the company or organization providing the *Component Version*.
A *Component Reference* is a reference to another *Component Version*. This initially sounds simple but there is actually a pitfall. A *Component Reference* is not identified by the *Identity* of another *Component Version*, thus by the *globally unique name* and the *version*. Each *Component Reference* has a *Component Version-local Identity* which additionally to the *globally unique Identity* of the referenced *Component Version* contains a *name* and an *extra identity*. This is due to the fact that a *Component Reference* does not have a strict predefined semantic. An example might help to understand this. Imagine one *Component Version* describing a version of a web server and another *Component Version* describing a version of an entire landscape of a REST application (as already mentioned, the definition of a *Component* is very loose). Now the REST application *Component Version* may use the web server *Component Version* twice, once as a classic HTTP server and once as a HTTP load balancer. Therefore, one *Component Reference* could have the *name* "http server" and the other "load balancer". Due to this definition of *Component Reference Identity*, the OCM provides the semantic capabilities to express such a situation. If the *name* is also not sufficient to uniquely identify a *Component Reference* within a *Component Version*, perhaps because the same *Component Version* is used as a HTTP server twice, the *extra identity* may be used to provide further distinction. This is a flat map of key-value-pairs. Besides, every *Component References* may also have a container for additional metadata, thus a *Labels* property.
Artifact is used as an umbrella term for *Sources* and *Resources*. *Sources* are usually the input for the build process of *Resources*, typically some source code. Subsequently, *Resources* are usually built from *Sources* and are capable of doing something. Thus, *Resources* are typically executables or OCI Images. *Artifact References* have a *Component Version-local Identity*, similar to *Component References*. There is one

significant different between the two which may easily lead to confusion. In the *Component Reference Identity*, the *version* is part of the referenced *Component Versions Identity.* In the *Artifact Reference*, the *version* is only part of the *Artifact Reference Identity.* What this means in practice is, if the *version* within a *Component Reference* changes, it is actually referencing a different *Component Version.* On the other hand, if the *version* within a *Artifact Reference* changes, the referenced *Artifact*, so the executeable or OCI Images, may still be exactly the same. Reciprocal, even if the *version* is the same in two *Artifact References*, the referenced *Artifacts* may have different versions. So there is no concrete coupling between the physical *Artifact*, so the executeable or the OCI Image, and the *Artifact Reference.* The *version* here is just another property to semantically distinct *Artifact References* within a *Component Version*, like *name* and *extra identity.* As a consequence, a *Artifact Reference* with the *name* "web server and *version* "v1.2.8" could reference an nginx v1.1.3 executeable and another *Artifact Reference* within the same *Component Version* with the *name* "web server" and *version* "v2.0.0" could reference an apache v2.4.54 executeable. This is an extreme case, supposed to show that there actually is now real coupling. In practice, the *name* and *version* usually do correspond to the underlying *Artifact.* And to do further distinctions in cases where a *Component Version* references the same physical *Artifact* twice, as an example twice nginx v1.1.3, once as an executeable for ARM platforms and once for x86-based platforms, the *extra identity* is used. The actual reference or rather coupling to the executeable or OCI Image is provided and *access* properties of the *Artifact Reference* .The *access* property, which is a map of key-value-pairs which has a required *type* property, provides a formal description of how and where to access the *Artifact.* Thereby, the *type* defines the access method, usually by specifying the repository type. The other key-values pairs in the map then depend on that type and may reference a particular commit by specifying the repository URL and a commit hash. Furthermore, all *Artifact References* have a *type* and *Labels* property, and specifically *Resource References* additionally have a *digest*, *relation* and *srcRefs* property. As already mentioned, a *Resource* may be an executable or an OCI Image. This can be expressed by the *type.* In the case of *Sources*, the *type* usually refers to the kind of source code management system. The *digest* is an object defining a *hash algorithm*, a *normalization algorithm* and the *hash* of the *Resource.* The *relation* property may have the value "local" or "external". The value "local" means that the *Resource* is derived from *Sources* contained in this *Component Version.* These *Sources* may then be referenced in the *srcRefs* property. This is an array of objects with a map type property called *IdentitySelector*, specifying the *Identity of the corresponding Resource Reference*, and

Labels.

As stressed several times throughout the last paragraphs, both *Component References* as well as *Artifact References* have *Component Version-local Identities.* This means, while the *Identity* of *Components* and respectively *Component Versions* contains a *globally unique name*, which makes the whole *Identity* globally unique and therefore *Component Versions* globally uniquely identifiable, the *Identity* of *Component and Artifact References*, thus the combination of *name*, *component name*, *version* and extra identity or just *name*, *version* and possibly *extra identity* respectively, is only guaranteed to be unique within the scope of a *Component Version.* As a consequence, to globally uniquely address the abstract concepts of *Component and Artifact Reference*, the combination of *Component Version Identity* and *Component or Artifact Reference Identity* is necessary. Thus, the combination of the globally unique *component name* and the *component version* plus respective local *Component or Artifact Reference Identity.*
As mentioned and shown in the beginning of this section, to *serialize* this abstract concept, OCM defines a format called *Component Descriptor.* A *Component Descriptor* thereby is the serialized form of a *Component Version.* It may be expressed in YAML as well as JSON. Thus, this is *machine readable* representation of a *Component Version.* The explanation also sporadically mentioned some abstract data types, mainly to make the concepts more tangible. In cases the data type was omitted, one may assume it is a string. Nevertheless, some of these strings have to adhere to specific patterns. Such information was omitted on purpose to avoid clutter. To obtain this information and get a technically complete specification of the standard, refer to the official documentation [38].

4.2 Capabilities of the Open Component Model

After this abstract description, this brief section revisions the major concepts of and points out the most important capabilities enabled by the Open Component Model.

Components and *Component Versions* respectively are an abstract concept, defined in a way that allows to describe anything, from single resources to entire application landscapes. Due to the loose semantic of the *Component References*, *Component Versions* may even be used to form arbitrary groups of other *Component Versions.* Thus, a *Component Version* named "SAP Gardener CI/CD" could be used to group all *Component Versions* used by the SAP Gardener CI/CD team. But especially, *Component Version* are capable of describing configurations of deployment environments. The *Component References* and *Artifact References* are then spanning a graph over all *Component Versions* and thereby indirectly over all *Artifacts* deployed in a deployment environment. By traversing this graph, it is therefore possible to get all *Resources* deployed in the environment. Since both, *Resources* as well as *Sources*, through the *access* property, provide a machine readable way to automatically find and download them, compliance scans with binary and even source code scanners may be conducted asynchronously, independent of any CI/CD pipeline.

There are some further aspects to the Open Component Model, that are not as important for this work, but are interesting nonetheless and therefore shall still be briefly discussed here. Besides enabling asynchronous compliance scans, the aforementioned concept allows for an entire holistic deployment automation based on a *Component Version* and thereby for a complete decoupling of CI from CD. A fact that was not explicitly mentioned before, while almost all properties of a *Component Descriptor* are immutable in the context of a *Component Version*, the *access* properties are exchangeable. This is especially important in complex development and deployment landscapes as it provides additional flexibility. Due to national restrictions, one might be obligated to deploy resources from artifact repositories located in the respective country. Hence, different *Component Repositories* may store *Component Descriptors* for a specific *Component Version* with different *access* properties. Thus, the *access* properties may be exchanged transparently without affecting the deployment automation.

While the Open Component Model thereby solves almost all limitations of the current state of the art approach, it does not provide the means to easily answer where Log4j is deployed. *Thus, there is still a need to store this information in a queryable manner alongside other possibly relevant metadata.*

4.3 Development and Deployment Landscape at SAP Gardener

So SAP Gardener is SAP‘s own managed Kubernetes service which enables to easily setup a kubernetes cluster in the cloud of one of the hyperscalers as well as on premise. In order to provide this service, the team has to maintain a sophisticated development and deployment landscape, which leverages the capabilities of the OCM. Therefore, this section provides a concrete example how to decouple CI from CD using the OCM.

A representation of the CI part of the development landscape is illustrated in figure 4.2 below.

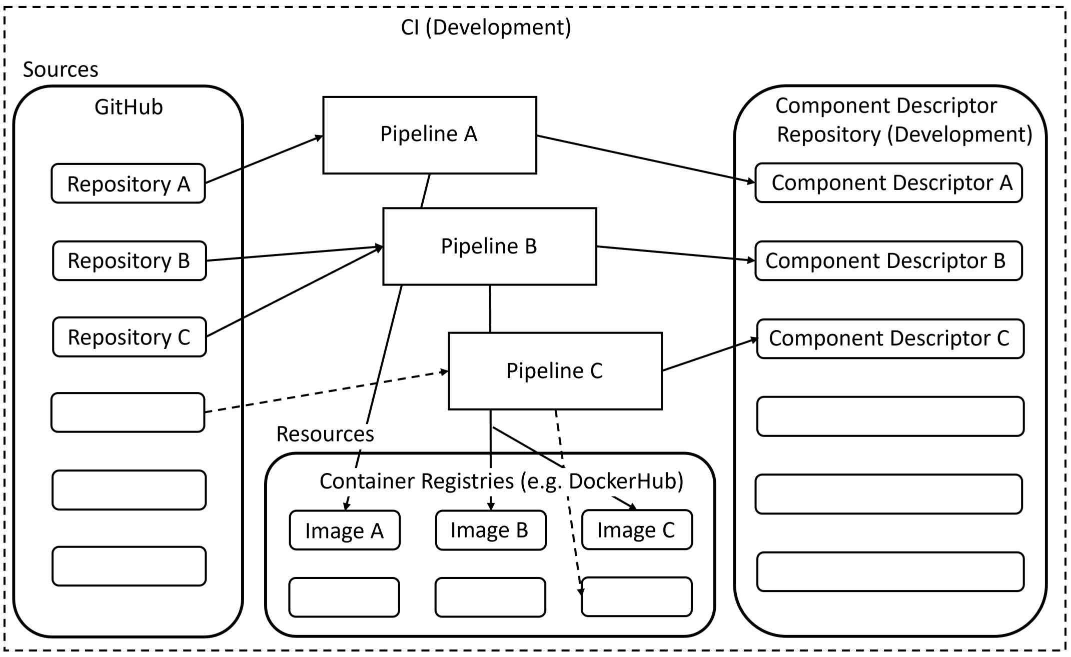


Figure 4.2: Gardener CI
Source: Based on [39]

As shown, there are several sources organized in git repositories hosted on GitHub. Furthermore, there are pipelines for creating the Component Descriptors. These pipelines may build OCI Images from the sources, as indicated by the unbroken lines to Pipeline A and Pipeline B. Instead of actually building, these pipelines may also just use references to existing open source OCI Images and optionally their corresponding sources, as indicated by the dashed arrows. In fact, more than half of the OCI Images powering SAP Gardener are only referenced and not built within the pipeline. Finally, as a result of the pipeline processing, the Component Descriptors are published into the Developments Component Descriptor Repository.

The next illustration in figure 4.3 is a representation of how this CI system is connected to the CD system.

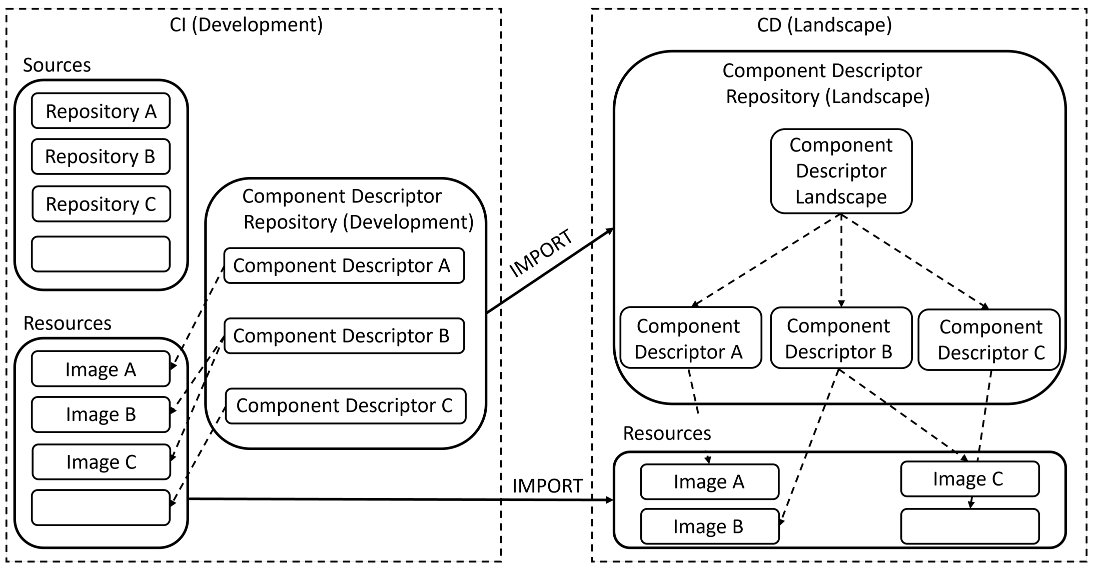


Figure 4.3: Gardener CD
Source: Based on [39]

The left side, CI (Development) is a condensed view of figure 4.2. The dashed arrows represent references. Since the references to the sources stay exactly the same, these dashed arrows are omitted to avoid even more clutter.

So whenever a new Component Version is released through the CI system, the CD system discovers and imports the corresponding Component Descriptors together with the referenced resources into Component Descriptor and Artifact Repositories of the respective landscape. Thereby, while essentially still referencing the same OCI Images, the access property of these Resource References of the replicated Component Descriptors is adjusted to the OCI Images in the landscapes Artifact Repository. Also, as shown in the illustration, in SAP Gardener, an additional Component Descriptor for a Landscape Component gets automatically created. It references all the Component Versions in a given snapshot of SAP Gardener. Every time a new Component Version is released in the CI system, a new Component Version of this Landscape Component is created.

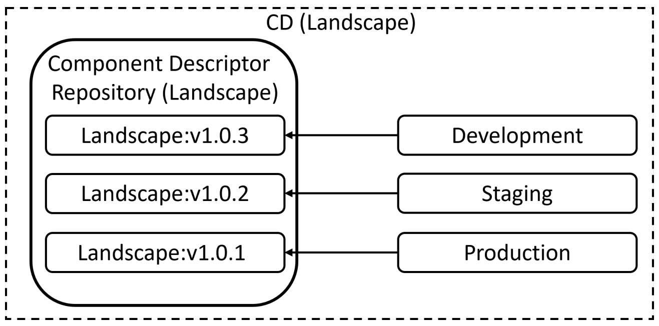


Figure 4.4: Gardener Deployments
Source: Based on [39]

Finally, figure 4.4 indicates how this setup and particularly the Landscape Component is leveraged to actually deploy an instance of the SAP Gardener. Therefore, the Component References of a Landscape Component Version are traversed and the referenced resources deployed. As usual, the Production deployment is based on an older Landscape Component Version than the Development deployment.

4.4 Integration of the Security and Compliance Data Lake into the SAP Gardener Landscape

As already indicated in the previous chapters, although the Security and Compliance Data Lake is an application for storing all kinds of metadata, the initial primary use case is storing the data resulting from compliance scans, thus dependencies, vulnerabilities and licenses. Below figure 4.5 shows the architecture designed for the integration into this complex environment of SAP Gardener.

The *Data Collection Service* is the central component of this architecture. Its job is to fetch all the information that shall be stored in the *Data Lake.* In practice, this would be done based on policies, requiring to do such a fetching once a day or based on some kind of event. In order to actually fetch the information, the Data Collection Service sends a request to the *Access Service* (1).

The Access Service is a transparency layer already built into the *Component Repositories* as part of the OCM. Thus, if the Data Collection Service requests a set of Component Versions and their referenced Artifacts, the Access Service first fetches the corresponding Component Descriptors from the Component Repository (2). After receiving this information, the Access Service evaluates the access property in the referenced Sources and Resources and sends respective requests to the *Source* and *Resource Repositories* to fetch the Artifacts (3). Finally, the Access Service may return all the requested information to the Data Collection Service.

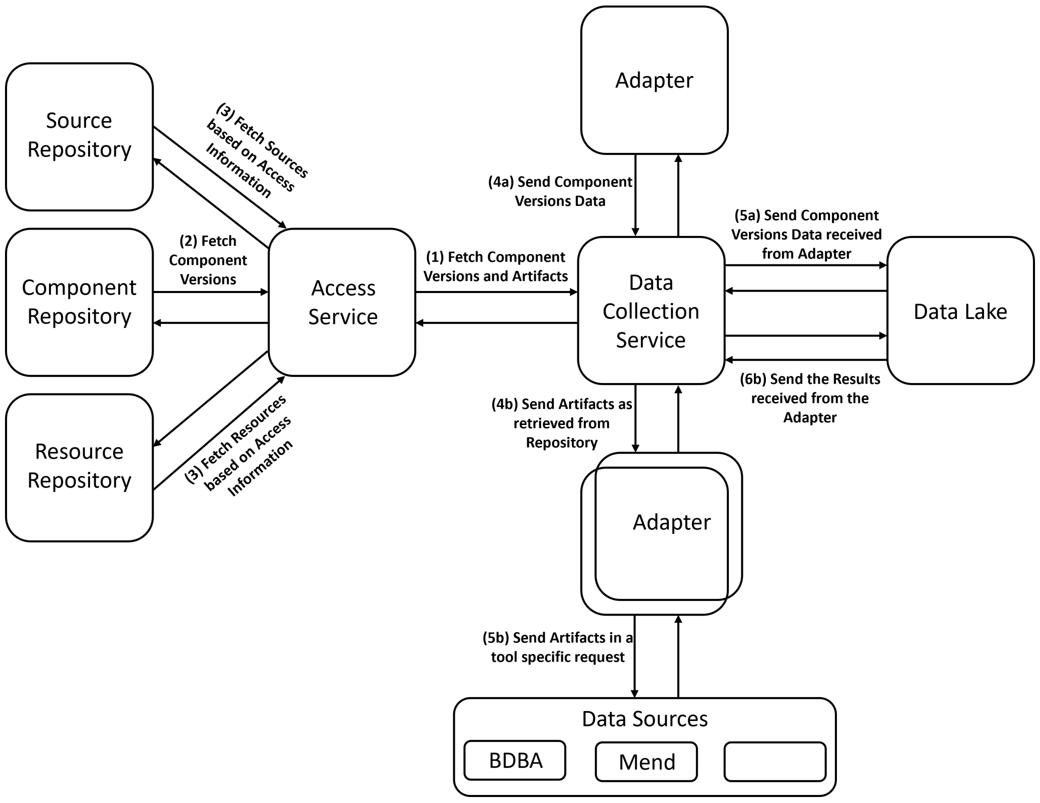

Source Repository
Component Repository
Resource Repository
Access Service
Data Collection Service
Adapter
Data Lake
Adapter
Data Sources
BDBA
Mend
(3) Fetch Sources based on Access Information
(2) Fetch Component Versions
(3) Fetch Resources based on Access Information
(1) Fetch Component Versions and Artifacts
(4a) Send Component Versions Data
(5a) Send Component Versions Data received from Adapter
(6b) Send the Results received from the Adapter
(4b) Send Artifacts as retrieved from Repository
(5b) Send Artifacts in a tool specific request

Figure 4.5: Data Lake Integration
Source: Own Representation

As shown in the figure, based on this information, there are two requests flows that may be triggered by the Data Collection Service (4a) and (4b). There is no required order to this, in practice, they may even be executed in parallel. The request flow illustrated above the Data Collection Service (4a) sends the Component Versions data, so the Component Descriptors, to an *Adapter* which adjusts and returns this data in the format required for the consumption by the Data Lake.

The request flow below the Data Collection Service (4b) distributes the Artifacts to a set of Adapters. Here, the Adapters initially wrap the Artifacts into the format required by a specific Data Source. In this case, there would be an Adapter for each compliance scanner. Upon receipt of the scan results, these Adapters also adjust the data to the format required for consumption by the Data Lake, before returning it to the Data Collection Service.

Finally, the Data Collection Service may send all the information, thus the information acquired from the Component Versions about Components, Resources, Sources and their relationships (5a) as well as the information from the scanning tools about dependencies, vulnerabilities and licenses within these Artifacts (6b) to the Data Lake.

There were no communication protocols mentioned so far. This is due to the facts, that it does not really matter on the conceptual level and that the final implementation of this integration architecture is out of the scope of this work. But in practice, the communication between Data Collection Service and Access Service, between Adapters and Data Sources and between Data Collection Service and Data Lake will be over HTTP. The Adapters however will most likely not be implemented as discrete services but as components within Data Collection Service and therefore communicate over shared memory. Since it is very likely that additional Data Sources shall be added later on, the important part on the conceptual level is to still logically separate the components. Thus, there should still be well defined interfaces between the Adapters and the Data Collection Service.

Chapter 5

System Design

This section describes the design of the *Security and Compliance Data Lake*. It covers the conception of the data model, the selection of a database and the design of the API. Thereby, it especially discusses alternatives and focuses on giving detailed information about the ideas and motives that led to specific design decisions.

5.1 Requirements

Before actually going into the details of the systems design, the requirements have to be specified, since they are at the core of every design decision.

5.1.1 Functional Requirements

The below table 5.1 provides a condensed list of the functional requirements for the Security and Compliance Data Lake. Thereby, every requirement is described by a short and precise but abstract statement of what functionality the system must have and an additional explanation which also includes an example. Additionally, there is a column categorizing the requirements as priority 1 or 2.

Priority 1 is functionality deemed necessary for a central metadata store which should solve the limitations and problems identified in the previous chapters. Furthermore, priority 1 functionality is usually functionality that has to be considered in the design process and otherwise cannot be easily added without foundational remodeling.

Priority 2 functionality describes convenience features which are less urgent and may easily be added later on.

Requirements		
Ref.#	**Functionality**	**Prio.**
R.1	The SCDL shall be able to consume and store metadata from multiple different data sources. The SCDL shall be able to work with any kind of metadata about software components. Therefore, it has to be able to handle multiple different scanning tools as well as other kinds of data sources like build tools. As an example, it may have to consume data from BDBA, Mend but also Jenkins. Thus, it has to be considered that besides vulnerabilities and licenses, a variety of other metadata types may need to be added in the future.	1
R.2	The SCDL shall store the metadata from different data sources without aggregation[1]. Different tools that generally serve the same purpose may provide similar information. As an example, BDBA and Mend are both SCA tools and therefore provide overlapping results. To ensure that no data is lost, this information shall not be combined and aggregated before storing.	1
R.3	The SCDL shall provide the metadata from different data sources with aggregation[1]. As mentioned before, to ensure no data is lost, the data from different data sources shall be stored without aggregation. Anyway, to be consumed by a user, this data shall be aggregated. As an example, when querying all packages contained in a specific resource, the result returned by the SCDL shall not contain the same package twice in different representations, if it was identified by BDBA and by Mend. Instead, it shall contain an aggregated representation of the package. Thus, some kind of aggregation layer is needed which provides transparency regarding the data sources.	1
	Continued on next page	

[1] *aggregation* in this context means to merge the data about a package of e.g. a BDBA scan and a Mend scan to a single package entity instance

Requirements		
Ref.#	**Functionality**	**Prio.**
R.4	The SCDL shall provide a level of aggregation[2] to group sources and resources. As pointed out before, one problem also with SBOMs is the disconnection of the artifact metadata and the deployment information. To bridge this gap, an additional aggregation level for grouping artifacts is necessary. As an example, this additional aggregation level shall enable to group all resources contained in a specific deployment.	1
R.5	The SCDL shall enable users to query the metadata on different levels of aggregation[2] As an example, a user shall be able to query for all vulnerabilities in a specific resource, thus query on the aggregate level of resources. But a user shall also be able to query for all vulnerabilities in an entire specific deployment, thus querying on the aggregate level of deployments (querying on this level of aggregation enables to answer where Log4J is deployed).	1
R.6	The SCDL shall enable users to perform assessments. The relevance of specific pieces of information such as vulnerabilities or licenses depends on the usage context. As an example, while the internal usage of an altered OSS with a copyleft license is lawful, the distribution is not. Therefore, a possibility has to be provided to assess such pieces of information in the context of their occurrence.	2

Continued on next page

[2] *aggregation* in this context refers to the "whole/part" semantic of the word [40]. Thus, since resources and sources are comprised of packages, they are both aggregations of packages. On a model level, the same applies for the relationships between packages and vulnerabilities or licenses as well as between entire deployments and the deployed resources.

Requirements		
Ref.#	**Functionality**	**Prio.**
R.7	The SCDL shall provide common data aggregation and filter functions for the queries. As an example, a user shall be able to filter for the vulnerability with the highest CVSS within a resource or shall be able to get the count of vulnerabilities within a resource.	2
R.8	The SCDL shall enable users to query the metadata in the common SBOM formats. In order to be able to fulfill governmental requirements of the executive order mentioned in the Software Bill of Materials section, the SCDL has to provide a way to to query the metadata in the common SBOM formats. As an example, a user shall be able to query the SPDX document for a specific resource.	2

Table 5.1: Requirements

So, by fulfilling this functional requirements, the Security and Compliance Data Lake may actually serve as a central application for storing and querying software metadata. Thereby, solving the problem of metadata being distributed throughout the development life cycle and bridging the gap between artifact metadata and deployment information.

5.1.2 Non-functional Requirements

Since this shall be a prototypical implementation, there is a strong focus on fulfilling the functional requirements. Thus, no concrete limits regarding performance or scalability such as a maximum response time of 5 seconds or support for up to 1000 concurrent users are set here. Considering the novelty of the topic, there is very few reference data and therefore, such specifications would be premature. However, for a central metadata store which may prospectively power dashboard web applications for monitoring purposes, scalability and performance definitely have to be considered in design decisions already.

5.2 Data Model

The basic entities relevant in the software supply chain are artifacts, thus sources and resources, and the packages comprising these artifacts. The definition of sources and resources is still the same as introduced in the previous chapter. Resources are capable of doing something and are usually executables or OCI Images. Sources are the code the resources are built from. Compliance scanners usually scan entire source code repositories or binaries. Through different methodologies, these tools detect the packages contained in these scanned artifacts on a best effort basis. In the context of this work, a package is defined as functional unit contained in artifacts, whereby it is usually a collection of files forming a library which is imported in the source code. By subsequently matching these packages against different databases such as the NVD, introduced in section 2.3 "Vulnerability Management", known vulnerabilities and licenses are identified. To give a better idea of these results, figure A.1 in the appendix shows a snippet returned from the API of BDBA. The results on their own are useful already and provide interesting data about the above mentioned entities. But it is still loose metadata that lacks context information such as which deployments contain the corresponding entities. Therefore, an additional entity type to conduct further grouping is required. The OCM already introduced such an entity type, the component.

To conclude this, from a high level perspective, the important entities are *components*, *sources*, *resources*, *packages* and the information attached to these entities such as vulnerabilities and licenses. To generalize this and abstract away from specific data sources, the entity type representing these types of metadata is called *info snippet*. So these entity types are the basic building blocks for the data model. From here on, it is getting rather complex and abstract. To still keep the explanations tangible, below figure 5.1 already shows an *entity-relationship model (ERM)* describing the final and universal data model. This may be used as a reference point throughout the following paragraphs, discussing the design decisions leading up to the specific entities, relationships and cardinalities.

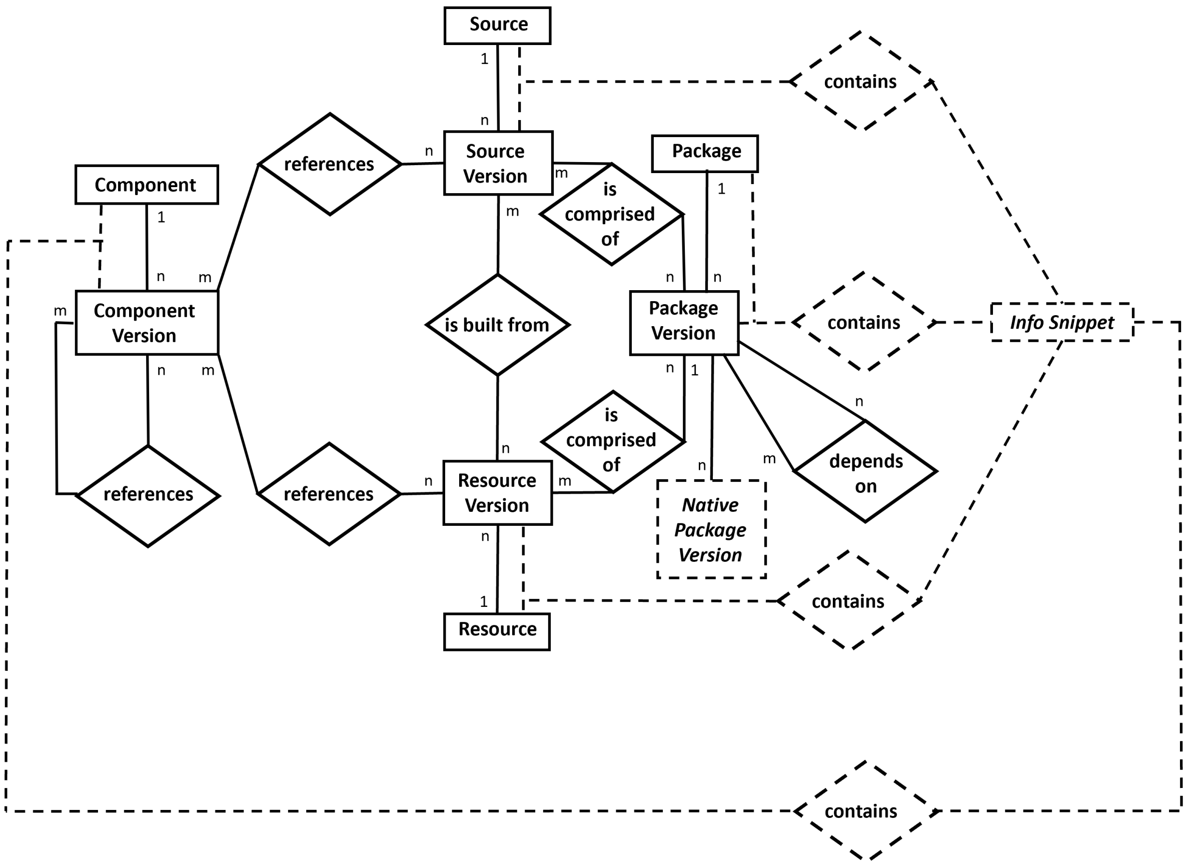


Figure 5.1: Data Model
Source: Own Representation

Even though the motivation behind every element may not be obvious immediately, the model as a whole should feel quite familiar and intuitive by now. An important notice at this point, the data model is inspired by the OCM. As mentioned above, especially the entity type *component* is lend from the OCM. Since this thesis is written within SAP Gardener, a seamless integration of the this Security and Compliance Data Lake such as described in the previous chapter is of course also a requirement, even though it is not explicit listed. But still, the Security and Compliance Data Lake is designed independent of the OCM. Thus, in theory it is entirely possible to use a different kind of component model. As an example, if one is able to express the concept of *components* and *artifacts* with the means of the SPDX standard, one could use SPDX instead of OCM to provide this structural information. Or, since SPDX is not optimal for this purpose, one could create and use an own component model, as long as it has the means to express *components* and *artifacts* (There is generally no necessity to distinct between *sources* and *resources*. *Sources* could be treated as *resources*, at the only cost of losing the connecting "is built from"-information between the two entities.).

5.2.1 Universal Data Model

Contrary to common ERMs, the one in figure 5.1 does not have any properties. There are two major reasons for this. Firstly, the just mentioned independence of a specific component model would hardly be possible if the data model would define fixed predefined properties for each entity type. Secondly, the different scanning tools provide a wide range of information about packages and other data sources but scanning tools may also be added. It is therefore practically impossible to foresee what properties may be needed. Besides, these may vary depending on the user of the Security and Compliance Data Lake.

Another special feature of above ERM are the entity types and relationships illustrated with dash lines. These represent classes of entity types and relationships. Since the whole set of data sources cannot be known upfront, the whole set of potentially required *Info Snippets* cannot as well. As already mentioned in several examples before, when adding a build tool as data source, an entity type *Build Information* may be needed. Also, the relationships of different *Info Snippet* entity types may vary. While a *Vulnerability* and a *License* is usually *contained* in multiple *Package Versions* leading to a (n:m)-relationship, a *Build Information* is usually associated to one *Resource Version* leading to a (1:n)-relationship. But generally, *Info Snippets* could be associated to any other entity type in the data model with any cardinality. This kind of flexibility is necessary to enable R.1 (consume and store metadata from multiple different data sources). The *Native Package Version* correspondingly illustrates the representation of a package, native to a concrete data source. Thus, instances of *Native Package Versions* may be *BDBA Package*, *Mend Package* or even *Jenkins Package*. So if all three data sources provide information about the exact same *Package Version*, each representation may be stored without a need to merge their properties before storing. Thereby, this enables R.2 (store metadata from different data sources without aggregation). Then, a set of properties commonly provided by all of the data sources may be aggregated on *Package Version* level, thereby also enabling R.5 (provide the metadata from different data sources with aggregation). So, all these *Native Package Versions* representing the exact same package are related to the same *Package Version* on the model level. As the different data sources may use different identifiers for the packages, the merging process cannot be triggered automatically. Hence, until a human defines that the *BDBA Package*, the *Mend Package* and the *Jenkins Package* are actually representations of the same package, no merging is done and each is related to a different *Package Version*.

So after explaining the special features of above ERM, the common entity types and relationships may be discussed. The basic entity type *component* is broken down into two distinct entity types, *Component* and *Component Version*. As immediately noticeable, this distinction is done for each of the basic entity types. *Component* is a purely abstract entity type. It merely groups all the versions of the same component together. Thereby, the *Component* may provide information about the semantics of this grouping such as whether this *Component* describes a specific deployment or whether it describes all software used by a department. Thus, information that is identical for all versions of this component and would have to be stored redundantly for each *Component Version* otherwise. Naturally, there are multiple *Component Versions* of each *Component*. Therefore, the (1:n)-cardinality here is self-explaining.

As established by the previously described grouping semantic of *components*, a *Component Version* may reference multiple other *Component Versions*. For example, a *Component Version* describing a specific version of a deployment may reference multiple other *Component Versions* such as *Component Versions* describing specific versions of a web server, a service and a database. Reciprocal, a *Component Version* may of course be referenced by multiple *Component Versions*. For example, a *Component Version* describing a web server may be referenced by several *Component Versions* describing different versions of the same deployment or entirely different deployments. Thus, this is a recursive (n:m)-relationship. There may also be a need to store additional occurrence specific metadata as properties of the *references*. Considering the above example, such occurrence specific metadata may provide information about the usage of the web server within the deployment, hence whether it is used as a HTTP server or as a load balancer. Together, these model elements fulfill requirement R.4 (provide an aggregation level to group sources and resources).

Furthermore, a *Component Version* may also reference multiple *Source Versions* and *Resource Versions*. As an example, the *Resource Versions* comprising the web server and the *Source Versions* from which the respective *Resource Versions* were built. As before, with the recursive relationship of *Component Versions*, the *Source Versions* and *Resource Versions* may of course also be referenced by multiple *Component Versions*, resulting in a (n:m)-relationship. Again, there may be a need to store additional occurrence specific metadata as properties of the *references*. Specifically, these *references* may be used to store *triage* information. As this *reference* describes the usage context of the *Artifact*, one may for example decide whether a copyleft license is or is not acceptable here. The relationship between *Source* and *Source Version* as well as between *Resource* and *Resource Version* is similar to the relationship between *Component* and *Component Versions*. But the abstract *Source* and *Resource* entity types actually do have a concrete purpose but only preventing redundant storage of certain properties. In this case, these abstract entities may have properties to store *triage policies*. As an example, one may store that a specific vulnerability may be ignored for the usage of *Resource Versions* v1.0.0 to v1.2.3 of a respective *Resource* within *Component Versions* v1.4.2 to v1.4.12 of a specific *Component*. The *references* and the abstract *Source* and *Resource* entities thereby enable the fulfillment of requirement R.6 (enable users to perform assessments).

Resource Versions may also reference the *Source Versions* they are built from. A *Resource Version* may be built from multiple *Source Versions* and a *Source Version* may be used to build multiple *Resource Versions*. This also results in a (n:m)-relationship.

Since *Artifacts* are comprised of *Package Versions* and the same *Package Version* may occur in multiple *Artifacts*, both *Source Version* as well as *Resource Version* have a (n:m)-relationship to *Package Version*. Again, there may be a need to store additional occurrence specific metadata as properties of the *is comprised of* relationship.

The relationship between *Package* and *Package Version* is again similar to the relationship between *Component* and *Component Versions*. But as already explained *packages* are broken down even further into three different entity types and thereby three aggregate levels. Since several data sources may provide different representations, thus different *Native Package Versions*, of the same *Package Version*, the cardinality of this relationship is (1:n). *Package Versions* frequently *depend on* other *Package Versions* and so on. This may lead to quite long chains of dependencies. This has to be kept in mind as these transitive dependencies are also relevant when trying to answer the question, whether a certain *Artifact* or *Component* contains a specific *Package* such as Log4J, and its corresponding vulnerabilities.

Finally there is the *Info Snippet* class of entity types. As explained above, different *Info Snippet* entity types may have relationships to different entity types with different cardinalities.

5.2.2 Insights into the Development Process

At this point, some insight into the development process may be beneficial to understand this design decisions. The scope of this central data store was initially much narrower. The first PoC was actually strictly bound to the OCM. Therefore, the data model predefined the properties of *component* and *artifact* entity types. As it was bound to the OCM, there was no issue in doing so. But the data model also predefined the properties of the *package* entity types. In fact, the third aggregate level for *packages*, *Native Package Version* did not exist at all. Instead, the *Package Version* entity type had a set of properties that was hoped to be common and harmonizable throughout all prospective data sources. At this time, the application was also tailored to only having scanning tools as data sources. Therefore, to define this set of properties, the API documentations of different scanning tools were analyzed for the common and most important properties, especially the ones of BDBA and Mend [41]. Additionally, to get a better understanding, both scanners were used on some artifacts to get some sample data. Then, the provided attributes were narrowed down and some interviews with developers were conducted. A huge effort was made here, as this was such an important decision. Also, instead of having the *Info Snippet* class of entity types, there was only a *Vulnerability* and a *License* entity type whose set of properties was defined in the same process. Besides the fact that there was already a substantial amount of disagreement between different developers which properties were actually required, by the time the PoC was finished, several new use cases were discovered that required additional properties and even entire additional entities to provide other information than about vulnerabilities or licenses.

This led to a change in perspective, interpreting the task of defining the right entity types with the right properties rather as a task to make the entire application extensible regarding the respective entity types and properties. But this flexibility and extensibility comes at the cost of a highly increased overall complexity. Apart from remodeling the data model, it also required a completely new architecture and completely different implementation. Thus, only after that, the scope widened drastically, also allowing other component models. Consequently, as a kind of disclaimer, the design process presented here is not exactly as chronological as it may initially seem, since it hides a complete development iteration leading up to the final concepts.

5.2.3 Application of the Data Model

Although a great effort was made to make the data model as tangible as possible, it may still be hard to grasp due to its abstract nature, omitting properties and introducing classes of entity types. Therefore, this section discusses how this data model could be applied. As reference component model, the OCM is used and as reference data sources, only BDBA is used. This thereby also reveals a problem that has to be faced when applying this data model to a real world use case. The figure 5.2 below shows the corresponding data model instance.

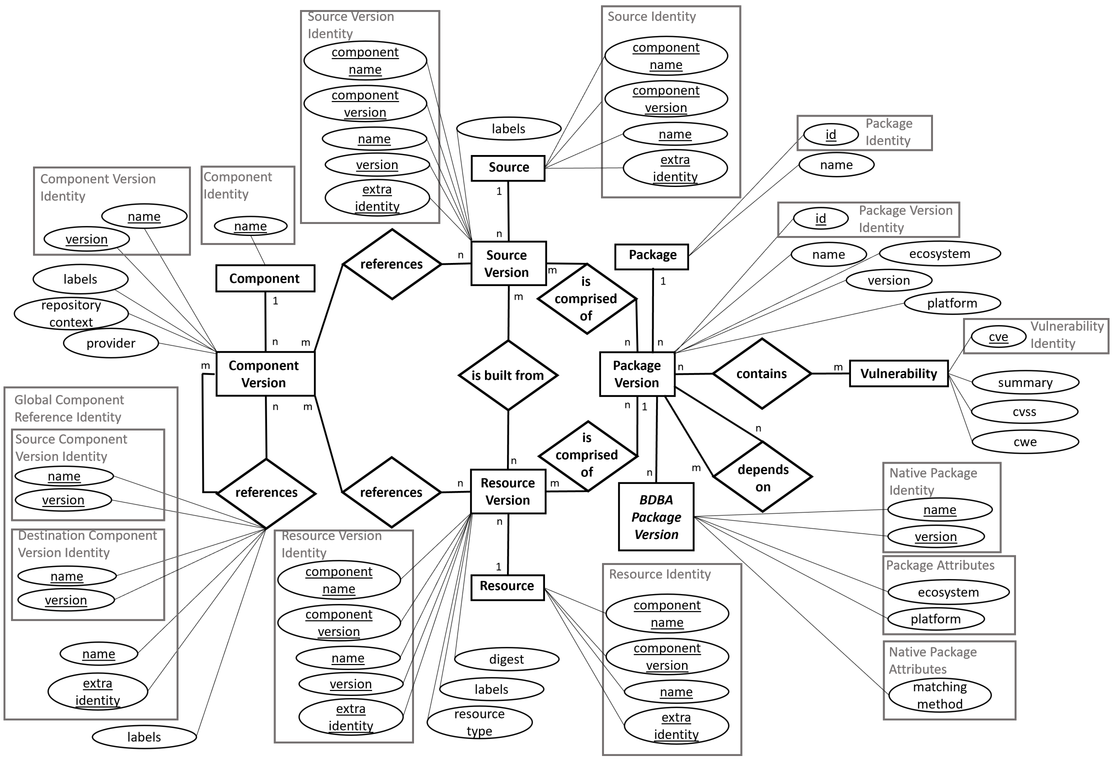


Figure 5.2: Example Data Model Instance
Source: Own Representation

The figure is very crowded. But this is necessary, as the purpose of this figure is to provide a concrete and thorough example of how the abstract universal data model may be applied to specific component models and tools.

The important aspect to point out here is the relationship between *Component Version* and *Source Version* and between *Component Version* and *Resource Version*. Although depicted as a relationship with (n:m)-cardinatility in compliance with the universal data model, due to the *Component Version-Local Identity* of *Source Version* and *Resource Version*, these can actually only be (1:n)-relationships. There were detailed explanations about this in section 4.1 "Open Component Model". *Source Version* and *Resource Version* were referred to as *Source Reference* and *Resource Reference* because in the context of OCM, instead of actually representing the technical artifact, they only reference the technical artifact through their *access* property. The SAP Gardener team chose this initially rather confusing specification of *Source Versions* and *Resource Versions* with *Local Identities* since in practice, it is difficult to reliably determine whether two referenced technical artifacts are actually the same technical artifact.

The following sections further explain the issue and discuss different approaches of dealing with this artifact identity problem. They thereby point out the use cases and limitations of each approach. The terms *Resource Version* and *Source Version* are from here on used as in the context of OCM, thus these terms are interchangeable with *Resource Reference* and *Source Reference*.

As the common software developer immediate attempts to translate this data model into tables, a corresponding exemplary relational model is shown in the appendix A.2.

Uniform Resource Identifiers

The initial and most obvious approach is to treat technical artifacts just as components. Thus, assign a globally unique name and a version to each technical artifact. But the reason this works for components is that these are purely abstract or logical entities. Thus, there is no digital or rather technical twin such as source code or a binary that corresponds to a specific component. If there is, who guarantees that two *Resource Versions* with the same name and the same version actually point to the same binary? Or reciprocal, that two *Resource Versions* pointing to the same binary actually have the same name and version? Where and how would one look up this globally unique name of a *Resource Version* in the first place?

A common approach in this situations are URIs. As introduced in section 2.1 "Software Identification", by providing a specification of how this URI is composed, standards such as purl provide a way to create theoretically reproducible identifiers. Theoretically, because in practice composing this URIs still has to be done by people. Consequently, there is always room for interpretation and human error. Imagine a *Resource Version* in two different *Component Versions* referencing artifacts in two different repositories, for example `github.com/example/nginx` with tag `1.0.3` and `github.com/example/nginx-webserver` also with tag `1.0.3`. At this point, it is difficult to determine, whether these are technically the same artifact and should consequently have the same URI. Thus, this approach is unreliable and impractical.

Content-Addressable Uniform Resource Identifiers

In order to guarantee that two *Resource Versions* with the same URI actually point to the same binary and that two *Resource Versions* pointing to the same binary actually have the same URI, this URI has to be content-addressable. Therefore, a coupling of identity and location is necessary.

The straight forward approach to do this in practice is to use the URLs provided by the repositories. But as the example above shows, this approach is not really flexible, as the URLs have to be immutable.

The OCM specifies that the access property is variable. Still in section 4.3 "Development and Deployment Landscape at SAP Gardener", it is explained that images, thus technical artifacts, and component descriptors are copied from the development landscape into other landscapes and that the access property changes in this process. So the development landscape may serve as single source of truth and the respective artifact URLs of the access property may be used as this content-addressable URI. Even though the URLs in the access property of the component descriptors in other landscapes may differ from the URI globally identifying the artifact, as this URI is used to copy the artifact to the location referenced by the URL in the new access property, it is the same per construction. So this approach is generally reliable and practical.

SAP Gardener did not consider this approach anyway, as it is more complex and globally unique identifiers for artifacts were not relevant for the original scope of the OCM, which was deployment automation.

Digests as Global Identifiers

So, another option to determine whether two artifacts are technically the same, even though they are located in different repositories is through calculating a suitable *normalized digest*.

A *digest* refers to a short, fixed-length string calculated through a hash function [42]. There are several features in which artifacts may vary but that are irrelevant for their comparison. For example, the exact same source code file may provide different digests when hashed with the exact same hash method, depending on whether it is hashed on a windows or a linux machine. This is due to the different line endings, thus "Carriage Return and Line Feed" on windows and "Line Feed" on linux. In order to calculate a normalized digest, the input data for the hash function is prepared so that such irrelevant features do not have an impact on the digest. So the input data is normalized.

This approach is generally reliable and clean, but there are again several problems. When looking at figure 5.2 above, there is already a digest property available for *Resource Versions*. As described in section 4.1 "Open Component Model", this digest property specifies a hash function, a normalization function and the corresponding digest value. Although the primary reason for this is that different kinds of resources, for example executeables and OCI Images, may need different kinds of normalization functions, this also means that the same artifact may have different digest values in different *Component Versions*.

Consequently, the digests need to be calculated in a standardized way, independent of the OCM. But even then, there is still another issue regarding the data model. There is no way to decide whether artifacts with different normalized digests may be different versions of the same artifact. Thus, the additional aggregate level, *Source* and *Resource*, required for triage policies is practically impossible to determine.

Local Identifiers

This is the approach used in the OCM. This concept was explained in detail in section 4.1 "Open Component Model". The detailed examination of the general artifact identity problem also showed implicitly why differentiating between the terms *Resource*, *Resource Version* or *Source*, *Source Version* and *Resource Reference* and *Source Reference* is so difficult from a modeling perspective.

After all, the most important aspect to point out here, is that although it is not intuitive, only having local identifiers for artifacts is still quite functional regarding the goals which shall be achieved by the data model.

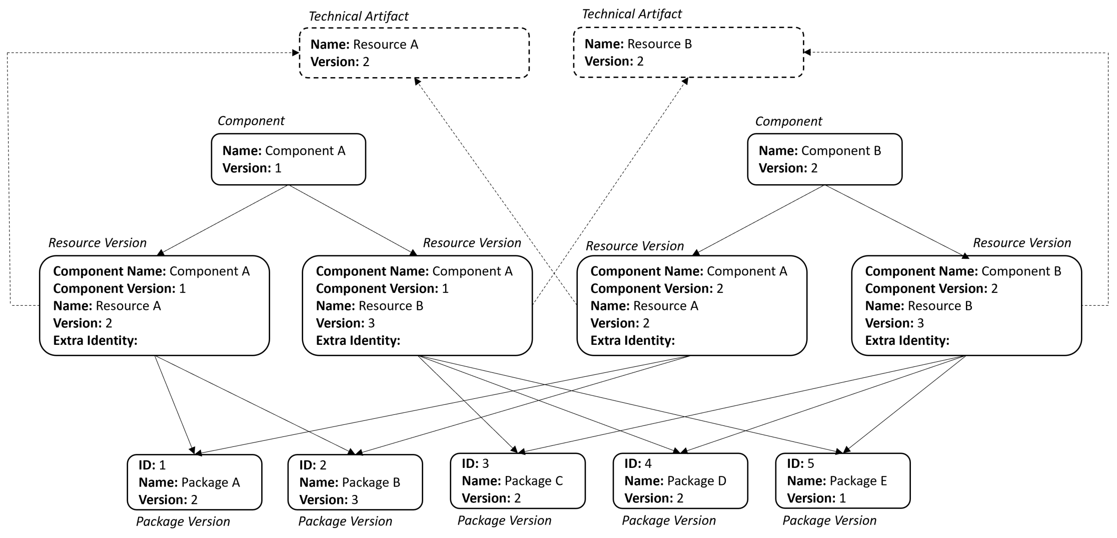


Figure 5.3: Local Artifact Identities
Source: Own Representation

Figure 5.3 shows two components that reference the exact same technical artifacts. But since the *Identity* of *Resource Versions* is local, the component name and version are part of the *Resource Versions Identity*, as also shown in figure 5.2. Thus, it appears as they are referencing different artifacts. But as the packages comprising the *Resource Versions* are identified by scanning the technical artifact accessed through the access property, they are the same for the *Resource Versions* which practically reference the same technical artifact.

So, for answering questions such as which deployment contains Log4J and its corresponding vulnerabilities, the local artifact identities do not make a difference. As in practice, it may also be assumed that within a *Component*, *Resource Versions* with the same name but different version numbers are in fact pointing to different versions of the same technical artifact, the *Resource* aggregate level is also kind of possible. But its scope and the scope of *triage policies* within these entities respectively is of course limited to a certain *Component*. Furthermore, attaching *Build Information* and other *Info Snippets* to artifacts is difficult with this approach.

5.3 Database

After the application context and data model are defined, a suitable database has to be selected. Therefore, this section analyzes the most relevant database technologies regarding their applicability as a central data store based on the previously defined data model.

5.3.1 Requirements for the Database

The most important aspect regarding the suitability of a database technology is the purpose of the data store and the respective kind of usage. Several relevant factors depend on this information. Is read or write performance more important? What may be acceptable delays when reading or writing data? What kind of queries may be used most frequently? Will the data be used for statistical analysis and require a lot of aggregation over dimensions such as time periods or locations? Are the entities highly inter-connected and require traversing relationships efficiently? How many users may need to access the data concurrently? May the database need to be distributed?

Write Performance: The data may be provided by all sorts of data sources. As already described in the reference architecture in section 4.4 "Integration of the Security and Compliance Data Lake into the SAP Gardener Landscape", the data may be fetched based on policies. Considering scanning tools, such policies may require to scan all Component Versions or respectively the referenced technical artifacts once a day or based on particular events, such as upon creation of a new Component Version. Depending on the number and size of artifacts, the scans themselves take something between several minutes up to several hours. As the information in the database may be updated concurrently with the scanning process, so for example every time the scanning tool has finished scanning a particular artifact, the results may be fetched, the *write performance may be quite low.* Theoretically, the database could correspondingly to the scanning tools take up to several minutes to write or update the information about an artifact.

Read Performance: As already pointed out in section 5.1.2 "Non-functional Requirements", the Security and Compliance Data Lake may prospectively be used as backend for dashboard web applications. Of course, these dashboards are for technical professionals. Thus, the response time does not necessarily need to abide to common UX design requirements. But still, common queries should ideally not take more than several seconds. So *read performance should be high.*
Queries: The most popular example for a query is "Which deployments contain Log4J?" or more precisely in terms of the previously introduced data model "Which Component Versions contain a vulnerable Log4J Package Version?" or even "Which Component Versions contain the Log4J Vulnerability?". Considering the exemplary data model in figure 5.2, to resolve this query, one has to find the *Vulnerability* with *CVE CVE-2021-44228* [2], find all *Package Versions* that contain this *Vulnerability* and also all *Package Versions* that directly on transitively depend on one of those *Package Versions*, find all *Source* and *Resource Versions* that contain one of the respective *Package Versions* and finally find all *Component Versions* that reference one of the respective *Artifacts*. Thus, resolving this query requires traversing a lot of relationships. This is prospectively also the most popular kind of query in general. Queries including aggregations may be something like "What Vulnerability contained in a Deployment X has the highest CVSS?". Therefore, even the queries including aggregation require a traversal of relationships to retrieve the subset of data the aggregation function has to be applied to.
Distribution: The database is the central metadata store of a company, used to monitor the application landscape and increase the transparency of the infrastructure. Therefore, the number of concurrent users will not be exceedingly high. Besides, the whole application and thus, also the underlying database is not business critical for a company. So database distribution to increase scalability in terms of parallel queries or to increase availability and fault tolerance are not a primary concern for the Security and Compliance Data Lake.

From here on, this rough overview provides a good idea of the most important properties to consider when selecting a database technology for the Security and Compliance Data Lake.

5.3.2 Relational Databases

The first database technology analyzed are *relational databases*. They are by far the most popular and widely used database technology. This is represented in Stack Overflow's 2021 developer survey [43]. There, the top 3 most used databases are

MySQL, PostgreSQL and SQLite, which are all relational databases. Among the technologies discussed here, it also has been around for the longest as the original paper introducing the relational model for databases by Edgar Codd was published in 1970 [44]. Therefore, the technology itself is very mature and there is a lot of know-how in the industry. Thus, relational databases are worth considering for every enterprise project.

Theoretical Foundations

A quick repetition of the foundations of relational databases. The name "relational database" stems from the mathematical definition of *relation*:

> *Given sets S1, S2,..., Sn (not necessarily distinct), R is a relation on these n sets if it is a set of n-tuples, the first component of which is drawn from S1, the second component from S2, and so on. More concisely, R is a subset of the Cartesian product S1 × S2 x . . . × Sn.* [45]

Thus, a mathematical relation is an unordered set of tuples of the same type. Mapping this to the *relational model*, a table represents a relation, each row represents a tuple of this relation, the ordering of the rows is irrelevant and as per definition of sets, all rows are distinct from one another [45].

These relations are then used to represent objects of a certain problem domain, therefore *entities* and their *relationships.* Each object type has a distinct type identifier, which becomes the name of the relation. Every instance of an object type must have an instance identifier, which uniquely identifies the entity among all the other entities of the same type. This identifier is commonly referred to as *primary key* [45].

Relationships between entities are mapped to relations by using the primary key of a related entity as a reference. In the scope of the referencing relation, the primary key of the referenced relation is commonly referred to as *foreign key* [45].

Furthermore, the relations are usually normalized, which leads to a low level of redundancy and reduces the risk for anomalies during data manipulation, thereby enabling the adherence to the ACID criteria [45].

In order to being able to consider and compare performance, a basic understanding of how the data is stored and accessed with each database technology is required. For relational databases, these basics are explained excellently and with attention to detail by Ramez Elmasri and Shamkant Navathe in "Fundamentals of Database

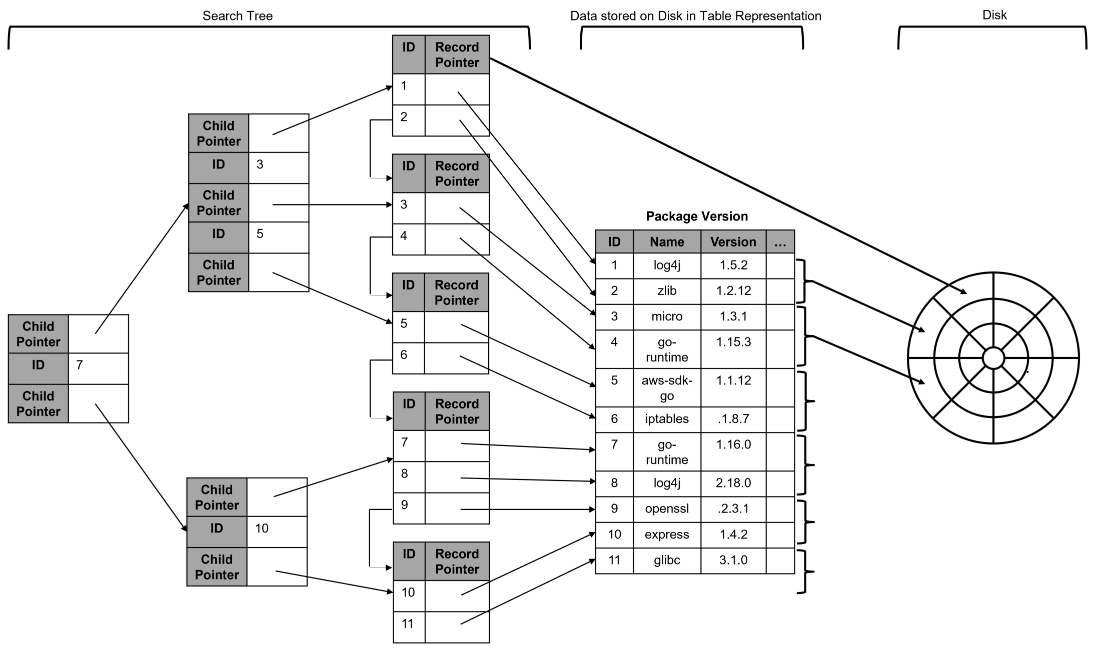
Systems" [46]. Based on this book, the most important principles which are also necessary for further understanding are introduced.

The data stored on disk is organized as *files* of *records*. Generally, a record is a collection of data values that can be interpreted as facts about entities, their attributes, and their relationships. In the context of relational databases, a record usually refers to a tuple of a relation, or respectively to a row in a table. Thus, the data is stored in a 1-dimensional format, listing all rows of a table.

There are several techniques, determining how the file records are physically placed on the disk, and hence how the records can be accessed. These techniques are commonly referred to as *file organizations* or *primary file organizations*. A *heap file* or *unordered file*, as the name suggests, places the records on disk in no particular order by appending new records at the end of the file. Opposed to that, a *sorted file* or *sequential file* keeps the records ordered by the value of a particular field called the *sort key*. Correspondingly, a *hashed file* uses a hash function applied to a particular field called the *hash key* to determine a record's placement on disk. *B/B^+-trees* use tree structures. *Secondary file organizations* enable efficient access to file records based on *alternate fields* than those that have been used for primary file organization.

The implications are quite intuitive. In the case of heap files, without additional indexing, thus secondary file organizations, a *linear search* looking through each record has to be conducted when searching based on a specific condition. Even with file organizations and indexing, this is still the case for conditions only considering non-key fields [46].

As common relational database technologies such as InnoDB, the storage engine behind MySQL and MariaDB, use B^+-trees for both, primary as well as secondary file organization, this is discussed in further detail [47]. Theoretically, hash key file organizations, or rather hash indexes, are capable of providing even more efficient look ups, but due to certain limitations and implications in the context of relational databases, these are not suitable for such an application and therefore not further considered.



ID	Name	Version	...
1	log4j	1.5.2	
2	zlib	1.2.12	
3	micro	1.3.1	
4	go-runtime	1.15.3	
5	aws-sdk-go	1.1.12	
6	iptables	.1.8.7	
7	go-runtime	1.16.0	
8	log4j	2.18.0	
9	openssl	.2.3.1	
10	express	1.4.2	
11	glibc	3.1.0	

Figure 5.4: B$^+$-tree as 4-way Search Tree
Source: Based on [46]

So, figure 5.4 provides a holistic view onto the search trees and how they are used in database systems. There are of course some simplifications and assumptions.

On the right of the figure is a representation of a *magnetic disk*, which is still most commonly used as storage for large databases. The concentric circles are called tracks. These tracks are further divided into *sectors*, represented by the intersections of the concentric and straight lines. This division leads to equal-sized *disk blocks*[3] which are also commonly called *pages*. The pages are the areas the arrows are pointing to in the figure. With InnoDB, the block size may be configured between 4KB and 64KB.

Data in secondary storage such as a disk cannot be processed directly by the *central processing unit (CPU)*. First, it must be copied into a primary storage such as main memory which is usually *dynamic random access memory (DRAM)*. The units in which this data is transferred between disk and main memory are the just introduced pages. Thus, to read data contained in a certain page, the whole page is copied into the main memory. And to change that data, this data is edited and then the whole page is written back, or in other words copied, to the disk. As accessing pages on the disk is in the order of milliseconds while accessing RAM is in the order of nanoseconds, this is a major bottleneck and therefore, the goal is to *minimize the*

[3]One may wonder that these blocks are not equal sized in the figure. The outer ones are actually much larger than the inner ones. In practice, there are different types of sector organization. The type shown in the picture maintains a fixed angle. To provide equal-sized disk blocks nonetheless, the outer sectors usually have a lower record density [46]

number of block transfers. So this enables to understand the necessity of *multilevel indexes* or *search trees.*

By only looking at figure 5.4 without the background knowledge just provided, there would be no apparent reason to construct and store such a sophisticated search tree. It would be more efficient to just store the index as a sorted table. Then, when accessing the database with said index, load the whole table into main memory and conduct a binary search. Thus, the only reason to use such search trees is when the index itself outgrows the page size and therefore has to be stored in multiple pages. So in practice, each node of a search tree is stored in a separate page as indicated by the arrow from the upper index leaf node. By traversing this search tree, the number of necessary page accesses may be reduced significantly. The index size with a search tree of order 4, so a node can at most have 4 children, is probably the most obvious simplification in above figure. The node and leaf indexes are so small that they could easily be stored together in a single index table.

The example shows the search tree as used for secondary file organization. So the records could generally be stored as a heap file, with no ordering at all. If used for primary file organization, instead of having pointers to the actual records, the leaves would directly store the records enforcing a respective order.

To traverse such a search tree, for example to find the record with ID 4, one starts at the root node. This is the node illustrated on the far left. As $4 < 7$, one follows the pointer above the 7 to the corresponding child node. In practice, this is a pointer to the page of the child node and consequently leads to copying this page into main memory. Then, this process repeats, as $3 < 4 < 5$, one follows the pointer between 3 and 5 to the next child node. In this example, this is already a leaf node, containing the pointer to the actual record with ID 4. So here, 4 pages have to be loaded, 3 index pages and the page containing the actual record.

So up until now, instead of B-tree or B$^+$-tree, the term search tree was used. This is because B/B$^+$-trees are essentially search trees which have to abide to an extra set of constraints. To be more concrete, a *B-tree of order m* is a tree which satisfies the following properties [48].

1. *Every node has at most m children.*
2. *Every node, except for the root and the leaves, has at least* $\lceil m/2 \rceil$ *children*
3. *The root has at least 2 children (unless it is a leaf).*
4. *All leaves appear on the same level*
5. *A nonleaf node with k children contains k-1 keys*

The goals of *balancing*, which is another term for all leaf nodes being on the same level, is to make the search speed uniform. Thus, the average time to find any random key is roughly the same. Furthermore, these constraints ensure that the nodes stay relatively full and do not end up empty if there are many deletions, thereby preventing the waste of storage space and an unnecessary high number of levels.

Figure 5.4 shows a B$^+$-tree. A B$^+$-tree is actually a variation of a B-tree which stores record pointers only at the leaf nodes. Consequently, the leaf nodes have an entry for every ID leading to some IDs – specifically 7, 3, 5 and 10 – being contained twice in the search tree. On the contrary, in a B-tree every ID is only present once at some level in the tree and the record pointer is stored directly alongside the child pointers. Thus, in above figure, the root node would additionally have a pointer to the record with ID 7. But of course, the whole tree structure would be different, as several leaf nodes could be omitted. Furthermore, B$^+$-trees usually link the leaf nodes to provide ordered access. This is indicated by the the arrows between the leaf nodes. In practice, this linking is also done with an additional pointer [46].

Finally, the *upper bound on running time* for searches with B-trees is examined. Initially, it is important to understand, the leaves carry essentially no information searching wise. Thus, for these considerations, leaves may just be regarded as terminal nodes. As explained in Donald Knuth's "Art of Computer Programming - Sorting and Searching" [48], suppose there are N keys, and the $N+1$ leaves appear on level l (In the book the relation that a B-tree with N keys always has $N+1$ leaves is just given. For a further explanation on why this is always the case, refer to appendix A.3). Then, as per constraint, the number of nodes on levels 1,2,3, ... is at least 2, $2\lceil m/2 \rceil$, $2\lceil m/2 \rceil^2$, ..., hence

$$N + 1 \geq 2\lceil m/2 \rceil^{l} - 1$$

Solving the equation for l

$$l \leq 1 + log_{\lceil m/2 \rceil}(\frac{N+1}{2})$$

Furthermore, on every level at most m keys have to be searched. As the elements in the nodes of a B-tree are sorted, binary search may be used. Therefore, the maximum number of look ups s per level is $log_2(m)$. Consequently, the total number of look ups is

$$s \leq \lceil (1 + log_{\lceil m/2 \rceil}(\frac{N+1}{2})) \rceil * log_2(m)$$

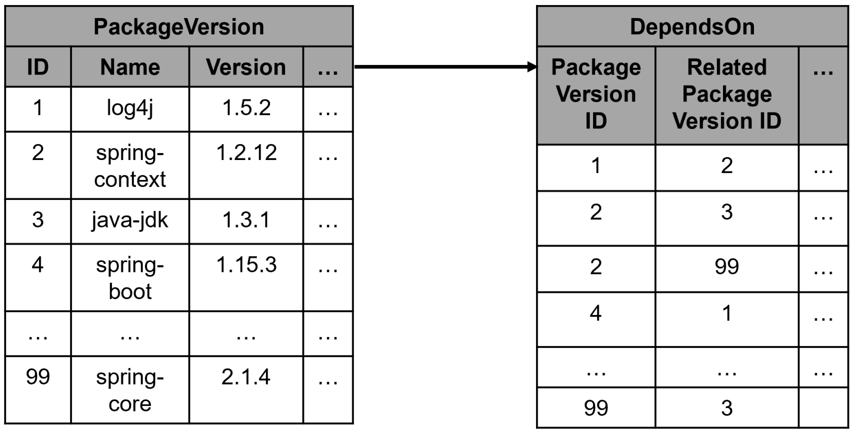
Considering Big O notation, constants may be ignored. Also m is definitely smaller than and independent of N. This leads to a worst case time complexity for searching a B-tree of

$$O(logN)$$

The estimation is also true for B$^+$-trees.[4]

Suitability for the Central Metadata Store

To evaluate and properly illustrate the suitability of the database technology regarding the aforementioned relevant properties, a representative example based on the established data model is used. The complete data model is quite complex and its actual instantiated form, thus the final entities, relationships and especially the properties, depends on the use case. Therefore, to keep the considerations general and clear, the representative example only considers the *Package Version* entity type. Since *Package Versions* may be contained in other *Package Versions*, this example allows for considering arbitrarily complex relationship chains.



PackageVersion			
ID	Name	Version	...
1	log4j	1.5.2	...
2	spring-context	1.2.12	...
3	java-jdk	1.3.1	...
4	spring-boot	1.15.3	...
...	...	...	...
99	spring-core	2.1.4	...

DependsOn		
Package Version ID	Related Package Version ID	...
1	2	...
2	3	...
2	99	...
4	1	...
...	...	...
99	3	

Figure 5.5: Package Versions and Dependencies
Source: Based on [49]

Figure 5.5 shows the *Package Version* entity type and the corresponding *depends on* relationship type mapped to tables and filled with some example entities. As

[4]Although this section covers several details, it is still just a very brief introduction into relational databases, the internals of database systems and B-trees, only covering the topics in the depth necessary for understanding the further sections. Aspects such as transactions and ACID criteria, how the records are actually placed on the disk, how variable-length fields are handled or what happens when records exceed page boundaries have been omitted. Regarding B/B$^+$-trees, the algorithms used for insertion or deletion and their respective bounds on running time are not discussed either. For further and more detailed information on these topics, refer to the respective sources "The Relational Model for Database Management" [45], "Fundamentals of Database Systems" [46] and "The Art of Computer Programming - Searching and Sorting" [48].

indicated by the ordering of the tables, the example also assumes there are indexes on *ID* and *PackageVersionID*. Furthermore, it may also be assumed, that there is a composite index on *Name* and *Version*.

For simplicity reasons, initially pretend there are no transitive dependencies. In this case, to answer the question "Which Package Versions does the spring-context:1.2.12 Package Version depend on?" in a relational database, the following SQL query would have to be used:

```
SELECT p1.Name, p1.Version
  FROM PackageVersion p1
  JOIN DependsOn
    ON DependsOn.RelatedPackageVersionID = p1.ID
  JOIN PackageVersion p2
    ON DependsOn.PackageVersionID = p2.ID
 WHERE p2.Name = "spring-context" AND p2.Version = "1.2.12"
```

Listing 5.1: Package Version Dependencies

Based on the sample data in figure 5.5, this returns *java-jdk 1.3.1* and *spring-core 2.1.4* [49]. Although slightly hard to read, the query is still relatively simple and not particularly computationally expensive, because it constraints the number of rows under consideration by applying the filter `WHERE p2.Name = "spring-context" AND p2.Version = "1.2.12"`. Based on the indexes, this initial look up of this record has a time complexity of $O(log(n))$ with n being the number of rows in the *PackageVersion* table. The consecutive join then has a time complexity of $O(m * log(n))$ with m being the number of records found during the previous look up and n being the number of rows in the *DepensOn* table. Each additional join in the query adds a $O(m * log(n))$. This assumes the so called nested loop join algorithm is used. This is usually the most efficient join algorithm in such scenarios, where indexes on the join condition exist and it is a small number of rows that have to be joined. Relational databases automatically estimate and choose the best strategy [50]. Thus, depending on the number of joins, the general time complexity for such a relationship traversal query may be estimated with

$$O(log(n)) + O(m * log(n)) + ... + O(m * log(n))$$

Now, to answer the reciprocal question "Which Package Versions depend on the spring-context:1.2.12 Package Version?" in a relational database, the following SQL query would have to be used:

```
SELECT p1.Name, p1.Version
  FROM PackageVersion p1
  JOIN DependsOn
    ON DependsOn.PackageVersionID = p1.ID
  JOIN PackageVersion p2
    ON DependsOn.RelatedPackageVersionID = p2.ID
 WHERE p2.Name = "spring-context" AND p2.Version = "1.2.12"
```

Listing 5.2: Package Version Reciprocal Dependencies

Based on the sample data in figure 5.5, this returns *log4j 1.5.2* [49]. Although this query looks very similar to the one before, it is computationally more expensive, because there is no index on *RelatedPackageVersionID* and consequently, the whole table has to be considered with a complexity of $O(n)$. In practice, an additional index could be added without problems, as the write performance which would suffer from maintaining an additional index, does not matter too much in this use case. Then this query would also have a complexity of $O(n)$

Finally, the transitive dependencies have to be considered. To traverse these transitive dependencies in SQL, recursive joins have to be used. Therefore, joining a table with itself. But as the dependency chains may be of arbitrary length, a simple recursive query is not sufficient. Nowadays, most of the popular relational database provide a SQL feature, the WITH clause or rather *Common Table Expressions (CTE)*, to support recursive queries [51, 52, 53]. CTEs are auxiliary statements which can be thought of as defining temporary tables existing for just one query [52]. So to actually answer the question "Which Package Versions does the spring-context:1.2.12 Package Version depend on?" correctly, thus also including the transitive dependencies, the following SQL query would have to be used:

```
WITH RECURSIVE cte(PackageVersionID) AS (
  -- Anchor member.
  SELECT d.RelatedPackageVersionID
    FROM DependsOn d
    JOIN PackageVersion p
      ON d.PackageVersionID = p.ID
   WHERE p.Name = 'spring-context' AND p.Version = '1.2.12'
  UNION ALL
  -- Recursive member.
  SELECT d.RelatedPackageVersionID
    FROM DependsOn d
    JOIN cte
      ON d.PackageVersionID = cte.PackageVersionID
)

SELECT p.Name, p.Version
```

```
  FROM PackageVersion p
  JOIN cte
    ON cte.PackageVersionID = p.ID
```

Listing 5.3: Package Version Dependencies (including transitive)

Based on the sample data in figure 5.5, this returns *java-jdk 1.3.1* twice (as union all does not remove duplicates) and *spring-core 2.1.4*. And correspondingly to answer reciprocal question "Which Package Versions depend on the spring-context:1.2.12 Package Version?", the following SQL query would have to be used:

```
WITH RECURSIVE cte(PackageVersionID) AS (
  -- Anchor member.
  SELECT d.PackageVersionID
    FROM DependsOn d
    JOIN PackageVersion p
      ON d.RelatedPackageVersionID = p.ID
   WHERE p.Name = 'spring-context' AND p.Version = '1.2.12'
  UNION ALL
  -- Recursive member.
  SELECT d.PackageVersionID
    FROM DependsOn d
    JOIN cte
      ON d.RelatedPackageVersionID = cte.PackageVersionID
)

SELECT p.Name, p.Version
  FROM PackageVersion p
  JOIN cte
    ON cte.PackageVersionID = p.ID
```

Listing 5.4: Package Version Reciprocal Dependencies (including transitive)

Based on the sample data in figure 5.5, this returns *log4j 1.5.2* and *spring-boot 1.15.3*. The queries are tested with a PostgreSQL 15 database. The respective DDL statements to set up the sample data are included in appendix A.4.

Besides the computational complexity and the general difficulty of the queries, another aspect to consider are the *schemas* required by relational databases. This may lead to a lot of null values in a table in cases where data sources do not provide the same amount of information for entities of the same type. Depending on how the data is stored on disk, this may lead to a waste of storage space. But most importantly, by *enforcing a schema on database level*, the flexibility to store further raw data provided by the data sources is lost. Furthermore, this makes it difficult to create or adjust schemata at application runtime. This is an issue for a central metadata store, considering that the properties and even several entities depend on the specific component model, data sources and also the use case.

So, as a conclusion for this section, a quick summary of the most important aspects when considering a relational database for the use case. (n:m)-relationships lead to large tables in relational databases. To be able to efficiently traverse these relationships and run queries with a reasonable performance, it is important to define *indexes* on the columns used in the respective join conditions. Relational databases that support recursive CTEs are able to *traverse hierarchies of arbitrary depth* without additional logic in the client side code. The *queries to traverse several relationships become quite large and may be hard to read.* The necessity for a schema on database level leads to a considerable loss of flexibility.

5.3.3 Document Databases

Document Databases have grown tremendously in popularity throughout the last several years. Especially MongoDB which is ranked the top 4 most used database in Stack Overflow's 2021 developer survey has found widespread use as a primary database for applications and therefore, as a substitute for relational databases [43].

Theoretical Foundation

Document databases are a type of NoSQL database which is commonly interpreted as an acronym for "Not only SQL". It is thereby somewhat of an extension of the most simplistic kind of databases, the *key-value databases.* These enable very fast look ups at the cost of only allowing users to address records by their key. The value is usually *opaque* to the database. Correspondingly, document databases also do not require a schema for the values they store, but they store the values in a standard format such as XML, PDF or most frequently JSON [54]. Hence, the values are not opaque and the database may still parse the data and create indexes on non-key fields even though being schema less.

Generally, the primary reason for using document databases, or rather any NoSQL databases, are the horizontal scalability issues of relational databases [54]. While vertical scalability refers to providing the database machine with more compute resources, thus, CPU and RAM, horizontal scalability refers to distributing the database over multiple machines. The important questions at this point are, where do the scalability issues come from? And also, how are they overcome by document databases?

De-normalization is one of the key features of document databases. Thus, while relational databases map entities and relationships to multiple tables connected by foreign keys, document databases allow storing entities and relationships as nested structures through their schema less nature. These nested structures are often referred to as *documents* or also more generally *aggregates*, a term from Domain-Driven Design describing a collection of related objects that is treated as a unit. Documents are grouped to *collections*, which is kind of the equivalent to a table in relational databases [55]. A common example is shown by figure 5.6 and listing 5.6.

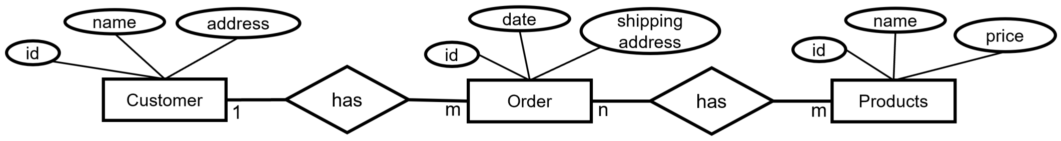


Figure 5.6: Sales Data Model
Source: Based on [55]

```
{
  "id": 1,
  "name": "Martin",
  "address":
    {"city": "Berlin"}
  "orders": [
    {"id": 99,
     "date": "04.01.2023",
     "shippingAddress": {"city": "Berlin"}
     "products": [
       {"id": 42,
        "name": "The Hitchhiker's Guide to the Galaxy",
        "price": 7.50 },
       {"id": 12,
        "name": "The Art of Computer Programming",
        "price": 120.00 }
      ]
    }
  ]
}
...
```

Listing 5.5: JSON Document

This solves the so called *impedance mismatch*, the difference between the relational model and in-memory data structures. Hence, with relational databases developers frequently have to translate the nested in-memory data structures to a relational representation to persist them. Of course, there are object-relational mapping frameworks such as Hibernate, but these often lead to performance issues [55]. By allowing to store objects as JSON objects, document databases such as MongoDB enable developers to store and retrieve their *JavaScript* objects or *Python* dictionaries as they are, without any further conversion. This is definitely one of the major reasons for their success, also in use cases where scalability is not a primary concern.

Furthermore, through this de-normalization, document databases *remove the necessity of joining a lot of data.* On one hand, this makes querying objects such as customer with all its orders easier for developers, as join queries are comparably difficult to write and understand. On the other hand, this increases the performance and scalability of this queries, as joins add quite some computational complexity which even depends on the number of stored objects. Thus, the performance of joins becomes worse as the database grows.

Besides, this kind of data organization enables efficient *sharding.* Sharding is a technique where different parts of the data are put onto different severs. Tables, or in the context of document databases rather collections, are split up into smaller shards which may be placed on multiple servers. A common sharding technique is to hash a key, also called *shard key* in MongoDB [56] or *partition key* in DynamoDB [57]. The respective hash value determines where, so on which server, to place the shard. This hashing provides an even distribution. But there are also other techniques such as ranged sharding. Thereby, the shards are distributed by the key value itself. This may be useful, when the records have some sort of order and it is common to access multiple sequential records. Or the key may be a post code and the distribution is based on the sever location to minimize latency [55]. This whole technique is enabled through the nesting, as documents are collection of related objects that are treated as units. Therefore sharding, or generally any form of partitioning, is rather difficult with relational databases. Since the data is scattered over multiple tables and the query language SQL puts no restrictions on how to query and connect this data, such partitioning would regularly require to send requests over the network to collect all the information [55].

So de-normalization obviously has numerous advantages. And this is also where most getting started guides of document databases such as MongoDB stop [58]. But naturally, it introduces the issues which were solved by normalization in the first place. Considering the example above, there is a (n:m)-relationship between orders and products. As the product is nested in the order which is again nested within the customer, updating a particular product while maintaining data integrity requires searching through all customers and through all orders within those customers, to update every single occurrence. Of course, in some cases write performance may not be relevant. But the nesting also affects queries. Imagine the above example describes the database of a retailer and this retailer wants to analyze its "Hitchhiker's Guide to the Galaxy" product sales over the last year. Again, to do this, the database has to search through all orders within all customers. There is the option to create indexes on nested fields which works pretty similar to relational database indexes to increase this performance. The problem that remains in this case is that document databases still have to retrieve the entire documents that contain the respective product from disk, thus, every customer record with all orders and all products. This leads to enormous RAM usage and may also slow down the performance significantly [59].

Generally, there are several restrictions which may serve as an orientation whether to nest or embed documents or whether to normalize and link them.

If the application frequently has to *access the embedded document independently of the embedding document(s)* and the *documents are quite large*, it may be best to normalize and link the document rather than embedding due to the otherwise excessive RAM usage [59]. In above example, orders may be queried independently of customers. So it may be better to normalize this and make the orders reference the respective customer.

If documents grow, the database may have to relocate it on disk to an area with more space. In the above example, as a customer may regularly order products from the retailer, the order array within the customer document would grow and potentially require relocation. Such a relocation decreases performance significantly [59]. Moreover, document databases may even have a *hard limit for the document size.* For MongoDB, this limit is 16MB [60]. So, this is another argument to normalize the orders.

Considering all these limitations, many-to-many relationships are especially problematic to de-normalize. Therefore, MongoDB even suggests to *model complex many-to-many relationships in normalized form* [61]. For simple many-to-many relationships, where the references may rarely be updated, it is also a possibility to embed the references instead of creating the equivalent of a join table.

As especially MongoDB and its query language provide a lot of the functionality also provided by SQL, the final question may be, why not store the relational model in a document database? The primary concern therefore may be data integrity. Although MongoDB provides ACID conform transactions, there is no way to enforce referential integrity on database level. Without a database schema, there is no way to tell the database about foreign keys and corresponding constraints.

Furthermore, several operations such as joining are usually more efficient with a relational database. The syntax to perform the join is also simpler with SQL. To make this more tangible, below listing shows the syntax to perform the logical equivalent to a left outer join on orders with products. The example thereby assumes the data is stored according to a relational model as shown in A.5.

```
db.orders.aggregate([
{ "$match": { "_id": 1 } },
{ "$lookup": {
  "from": "relations",
  "let": { "orderId": "$_id" },
  "pipeline": [{"$match":{"$expr":{"$eq":["$order","$$orderId"]}}},
  { "$lookup": {
    "from": "products",
    "let": { "productId": "$product" },
    "pipeline": [{"$match":{"$expr":{"$eq":["$_id","$$productId"]}}
   }],
    "as": "products" }
  }],
  "as": "order_products" }
},
{ "$project": {
  "_id": 1,
  "date": 1,
  "shippingAddress": 1,
  "products": "$order_products.products" }
}
])
```

Listing 5.6: JSON Document

Without going into too much detail about this query, it is obvious that it is rather complex and resembles a query execution plan. Thereby, the output of each stage ($match, $lookup, $lookup, $project) is the input of the consecutive stage [62]. This drastically limits the automatic optimization capabilities. So MongoDB will generally always perform a nested loop join even without indexes when a relational database would probably perform a merge or hash join.

So document databases and de-normalization work especially well, in cases where the application only requires a limited set of specific repetitive access patterns. But in order to actually leverage the advantages, these access patterns have to be identified and the de-normalization has to be done accordingly. This requires a very good understanding of the application already in the data modeling phase.[5]

Suitability for the Central Metadata Store

In contrast to the corresponding section about the suitability of relational databases, this one is kept short. Based on the background knowledge provided in the theoretical foundations, it is apparent that the advantages of document databases may barely be leveraged for this use case. Extensive examples on how to implement the central metadata store with the capabilities of a document database are therefore omitted as well.

The data model is based on (n:m)-relationships. Also, the database's primary purpose is to perform analysis about different entities. Thus, "Which Component Versions contain a vulnerable Log4J Package Version?" with Package Version being the access point. Or reciprocal, "Which Package Versions are contained in a particular Component Version?" with Component Version being the access point. So, there is not a limited set of specific repetitive access patterns as every entity may need be used as access point to answer common questions. Therefore, the *central metadata store cannot really be optimized through de-normalization.*

Consequently, at least concerning the major entities, the data model would have to be stored as a relational model. In this case, the document database would have the same scalability issues as relational databases and thereby mitigate a lot of the general advantages of document databases. Relational databases are practically designed with the primary purpose of handling relational models. Consequently, the query language, SQL, as well as the general data processing is better adjusted to such use cases.

But, being *schema less* on database level may be an advantage. Although there is no way to enforce referential integrity on database level, enforcing a schema on application level allows for easier creation and adjustments at application runtime.

5.3.4 Graph Databases

The last database technology analyzed are *graph databases*. They are also a type of NoSQL database. Although especially Neo4j is doing a lot of advertising, graph databases are still the least known database technology, as also represented by their

[5]Again, although this section covers several details about document databases, it is still just a very brief introduction to provide the foundations necessary for understanding the further sections. Especially the topics of transactions and ACID criteria as well as the CAP theorem and the general treatment of consistency in document databases have been omitted.

absence in Stack Overflow's 2021 developer survey [43].

Theoretical Foundation

Again, first a quick repetition of the foundations. The mathematical definition of a graph is the following:

> *A graph is a pair of sets $G = (V, E)$. In an undirected graph, the elements of E are 2-element subsets $\{v, w\}$ of V. In a directed graph, the elements of E are 2-element tuples (v, w), thus ordered pairs, of V.* [63]

In this definition, V refers to *vertices*, or rather *nodes*, and E refers to *edges*. Quite frequently, two nodes v and w connected through an edge (v, w) are elements of different sets, thus, $v \epsilon S1$ and $w \epsilon S2$ with $V = S1 \cup S2$. So, E is a subset of the Cartesian product of $S1xS2$. Looking back at the definition of relations in section 5.3.2 "Theoretical Foundations" of relational databases, this is also the definition of a relation.

Thus, E is a *relation* and V is the union of the *source set* and *destination set*. Now, this is generally well known from functions, which are defined by a *domain* $S1$, a *codomain* $S2$ and a *mapping* f, which assigns elements of $S1$ to elements of $S2$. Hence, drawing a function in a coordinate system is essentially the visualization of a graph, where commonly the *x-axis* denotes the *source set* and the *y-axis* denotes the *destination set* and the dots, or rather line, for most common functions denotes the *relation*.

In practice with business data, the source set and the destination set are usually sets of discrete and finite elements and the relationship between them is rather artificial and application specific. A primitive example, $S1$ may be the set of all package names, $S2$ may be the set of all version numbers and the relation E (v, w) may be package versions. Thereby, the source set and destination set are rather irrelevant. Besides, one could argue that for the scope of an application, the union of all v in E is equal to the source set and the union of all w in E is equal to the destination set.

log4j	1.5.2
log4j	1.3.1
spring-context	1.2.12
java-jdk	1.3.1

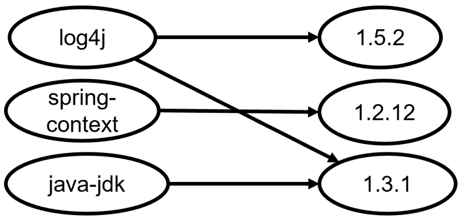


Figure 5.7: Graph Representations
Source: Own Representation

So, for the scope of an application, the table and the vertices and edges in figure 5.7 may be interpreted as two different representation of the same graph. This comparison of relations and graphs may be a slightly abstract and the example is oversimplified, but its purpose is to show how these two concepts and their common representations are tightly coupled.

The first part approached the topic from a mathematical perspective. From here on, this shifts to the actual technologies. Although the previous example may indicate this, in graph databases the graph is not used as a different representation for classical relations, hence to represent the relation of the properties of an entity. It is rather used to represent the relationships between the different entities. Thus, even though figure 5.7 is theoretically accurate, the following figure 5.8 is more suitable from a database technology perspective.

PackageVersion		
ID	**Name**	**Version**
1	log4j	1.5.2
2	spring-context	1.2.12
3	java-jdk	1.3.1

DependsOn		
Package Version ID	**Related Package Version ID**	**Risk Factor**
1	3	0
2	1	8
2	3	1

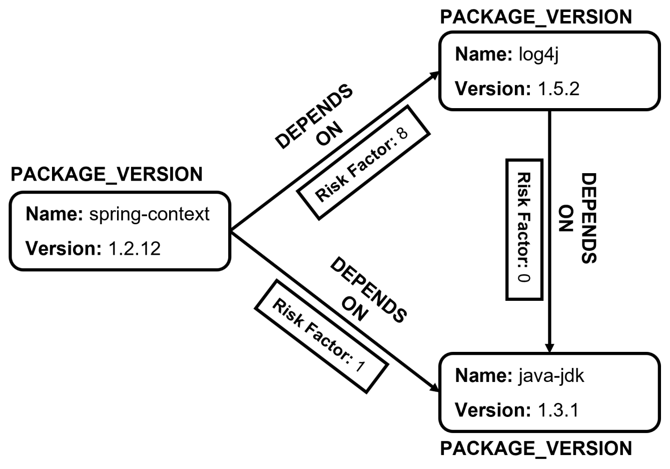


Figure 5.8: Graph Database Representation
Source: Own Representation

This graph model is called *labeled property graph.* As defined in the "Graph Databases - New Opportunities for Connected Data" written by engineers of the Neo4j database, a labeled property graph has the following characteristics [49]:

1. *It contains nodes and relationships.*
2. *Nodes contain properties (key-value-pairs).*
3. *Nodes can be labeled with one or more labels.*
4. *Relationships are named and directed, and always have a start and end node.*

5. *Relationships can also contain properties.*

This model is generally quite intuitive. But obviously, even in this more complex example, with the suitable interpretation of the table (thus, instead of interpreting the relation itself, the foreign key relationships are considered), it still expresses the same graph as the nodes and relationships. And, as discussed previously, document databases are able to express this graph as well, either also through a normalized model or through nested documents.

So the key difference of graph databases is not that they store graphs in general. The key difference is that their file organization and data processing is optimized to work with graph data.

As shown in figure 5.8 on the bottom left, table representations and consequently relational databases use an index to link entities, or in this context rather nodes. To increase the performance of reciprocal queries, an additional index may be created on *Related Package Version ID*. These *indexes are global*, so as the graph grows, so do the indexes.

In contrast, in most graph databases with native processing capabilities each node maintains direct references to link nodes. Thus, each node maintains kind of a *local index* whose size and consequently performance do not depend on the total size of the graph. This concept of linking adjacent nodes is commonly referred to as *index-free adjacency*.[6] For further assumptions and explanations, the architecture of the Neo4j database is considered [49].

So, when querying "Which Package Versions does the spring-context:1.2.12 Package Version depend on?", the initial look up works quite similar as in relational databases and therefore still has a complexity of $O(log(n))$. But finding the related packages is $O(1)$ instead of $O(log(n))$. Since nodes maintain references to nodes with incoming as well as to nodes with outgoing relationships, the reciprocal query "Which Package Versions depend on the spring-context:1.2.12 Package Version?" may be answered with the exact same complexity [49].

So given these complexities, in theory graph databases scale and perform better than other database technologies when the application involves a lot of graph traversals.[7]

[6]There has been a lot of discussion whether index-free adjacency is a requirement for graph databases [64]. The current consensus seems to be that it is not, since especially ArangoDB developed another approach to the problem which allows similar complexities as index-free adjacency for traversing graphs [65].

[7]As before, this theoretical foundation chapter is obviously not complete. Especially the file organization which enables the fast traversals are an interesting topic. For a good and thorough explanation of a implementation, refer to "Graph Databases - New Opportunities for Connected

Suitability for the Central Metadata Store

This suitability section takes a similar approach as the first one evaluating the suitability of the relational database. Thus, examining the same representative *Package Version* example. This allows for a direct comparison afterwards. Below figure 5.9 therefore illustrates the equivalent example from 5.3.2 "Theoretical Foundations" of relational databases as a classical graph.

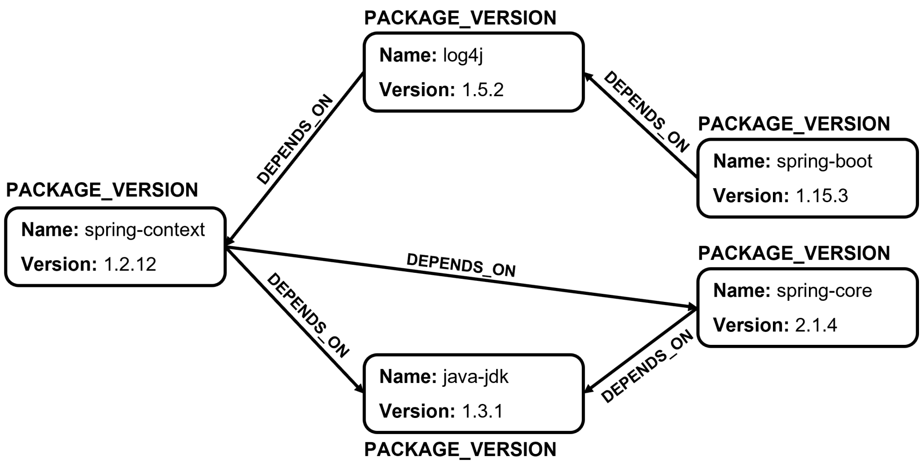


Figure 5.9: Package Versions and Dependencies as Graph
Source: Own Representation

Neo4j defines its own SQL inspired language called *cypher*. To answer the respective question "Which Package Version does the spring-context:1.2.12 Package Version depend on?" in Neo4j, the following cypher query would have to be used:

```
MATCH (p1:PACKAGE_VERSION {Name:"spring-context",Version:"1.2.12"})
MATCH (p2:PACKAGE_VERSION)
WHERE (p1)-[:DEPENDS_ON*1..]->(p2)
RETURN (p2)
```

Listing 5.7: Package Version Dependencies (including transitive)

The first line in listing 5.7 matches the node with the *label* `PACKAGE_VERSION` and the key-value-pairs `Name: "spring-context"` and `Version: "1.2.12"` and assigns the resulting nodes, or in this example rather node as there is just a single one that fulfills this pattern, to the variable `p1`. The second and third line match all nodes with the *label* `PACKAGE_VERSION` that `p1` depends on. Thus, the `WHERE` uses a path as a predicate. Thereby, `DEPENDS_ON` specifies the *label* of the relationship and the part after the asterisk specifies the minimum and maximum amounts of hops between the nodes. In this case, the minimum is 1 because otherwise it would consider itself

Data" [49].

and as there is no explicit maximum given, it defaults to infinite, to consider all transitive dependencies. The nodes that fulfill this criteria are assigned to `p2` and returned, in this case, *java-jdk 1.3.1* and *spring-core 2.1.4*.

Assuming there is a composite index on *Name* and *Version*, the initial look up of the node has a time complexity of $O(log(n))$ with n being the number of all nodes with the label *PACKAGE_VERSION* [49]. The consecutive look ups then have a time complexity of $O(m)$ with m being the number of nodes found during the previous look up. Thus, depending on the number of hops, the general time complexity for such a relationship traversal query may be estimated as

$$O(log(n)) + O(m) + ... + O(m)$$

So theoretically, the graph database performs and scales significantly better than the relational database for this kind of queries. Besides, the query is definitely simpler to read and understand than the SQL query involving Common Table Expressions. Although, SQL it is arguably still easier to learn the Common Table Expression than learning an entire new query language.

Now just for completion, to answer the reciprocal question "Which Package Versions depend on the spring-context:1.2.12 Package Version?", the following cypher query would have to be used:

```
MATCH (p1:PACKAGE_VERSION {Name:"spring-context",Version:"1.2.12"})
MATCH (p2:PACKAGE_VERSION)
WHERE (p1)<-[:DEPENDS_ON*1..]-(p2)
RETURN (p2)
```

Listing 5.8: Package Version Reciprocal Dependencies (including transitive)

The only difference in this query is the reversed direction of the relationship in line 3. In this case, it return *log4j 1.5.2* and *spring-boot 1.15.3*. The respective cypher statements to set up the sample data are included in appendix A.6.

Besides the efficient traversal of relationship with rather simple queries, there are some other aspects to consider with graph databases. Generally, most graph databases support transactions with *ACID compliance* [66, 67]. Also, there are some mechanisms to support *referential integrity*, but those do not work exactly as in relational databases. So, in graph databases there is no way to create a relationship to or from a node that does not exist. Furthermore, nodes cannot be deleted as long as relationships to or from that node exist. But compared to relational databases, there is no option to cascade a delete [68].

Regarding *schemas*, or the general storage model, different graph databases vary. Neo4j, for example, only supports key-value properties on nodes or relationships.

This is considered as single model graph database. In contrast, in ArangoDB every node is a JSON document, which is considered as multi model graph database [69]. Therefore, Neo4j does not really have a classical schema. But is still offers several types of constraints such as *unique node property constraints*, *node property existence constraints* and *relationship property existence constraints* to enforce a specific data model [70]. Similar to MongoDB, for ArangoDB a schema may be enforced on application level [71].

5.3.5 Database Selection

So, a conclusion of the this analysis of the suitability of database technologies for a central metadata store.

Relational databases are a solid choice. As shown in the corresponding suitability section, modern SQL provides all features necessary to conveniently implement the required functionality. But the computational complexity of joins is

$$O(log(n)) + O(m * log(n)) + ... + O(m * log(n))$$

As traversing relationships is involved in the prospectively most frequently used queries, this is a quite important measure.

Document databases are not the technology for this application. As it does not allow for significant de-normalization since the access patterns are versatile, the document database does not have an advantage over a relational database.

Graph databases are optimized for relationship traversal. The previous section has also shown that the cypher queries are simple in comparison to SQL. Due to its purpose, its computational complexity of traversing relationships is better than with relational databases

$$O(log(n)) + O(m) + ... + O(m)$$

So, the graph database seems to be the theoretically best database technology for this application. In practice, it also has to be considered, that the application has to be supported and that there is less experience around graph databases in the industry. This also increases the chances of unforeseen queries or other situation, where the graph database may be a problem.

For the scope of this work, the graph database is chosen for the implementation to further evaluate its suitability and practicality.

The concrete graph database solutions considered were primarily Neo4j and ArangoDB. Neo4j because it has an open source community edition, good documentation and a supportive community. ArangoDB mainly because it is multi model and therefore enables storing JSON, which offers greater flexibility.

Finally, Neo4j was chosen as it seemed to be the better fit over all. Even in the situations where JSON has to be stored, it is rarely necessary to traverse into the document structure. So it is usually sufficient to access the JSON document by its key which may also be done in Neo4j.

5.4 Programming Language

The *programming language* used to implement the services comprising the Security and Compliance Data Lake is *Go* [72]. Generally, the services could probably be implemented with any common programming language.

Since SAP Gardener is a managed kubernetes service and kubernetes is primarily written in Go, Go is the main programming language of the team. Therefore, Go was primarily chosen to fit in into the SAP Gardener ecosystem. This increases interoperability with other services of the team. But most importantly, this enables other members of the team to further maintain the application.

5.5 Application Architecture

To further discuss the application and the other important design decisions involved, especially concerning the API, a rough understanding of how the application is composed may be helpful. Thus, the below figure 5.10 illustrates the application's architecture.

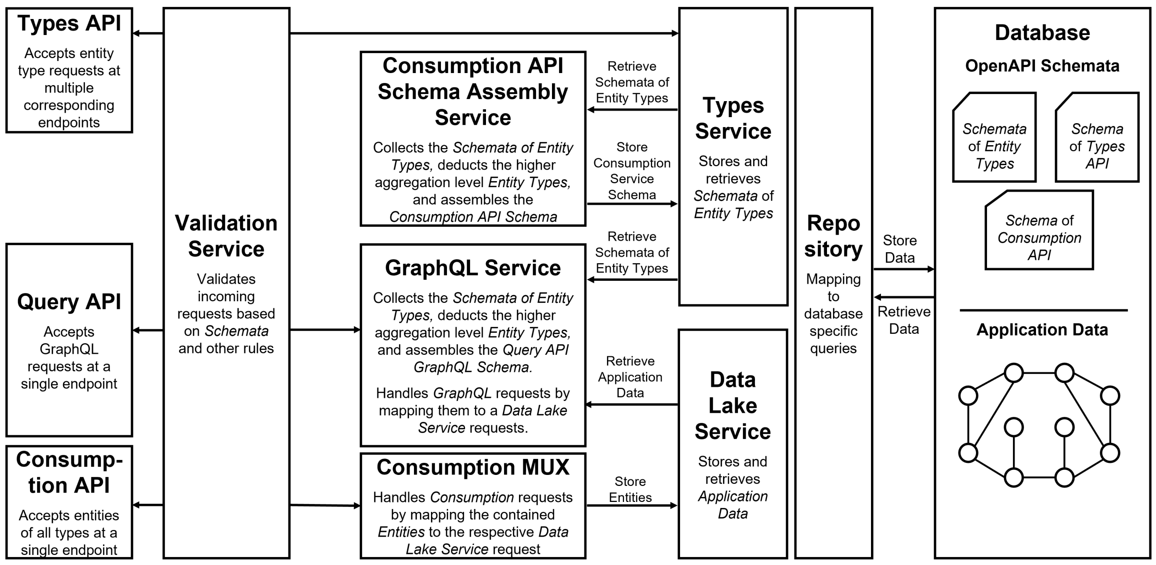

Types API
Accepts entity type requests at multiple corresponding endpoints
Query API
Accepts GraphQL requests at a single endpoint
Consumption API
Accepts entities of all types at a single endpoint
Validation Service
Validates incoming requests based on Schemata and other rules
Consumption API Schema Assembly Service
Collects the Schemata of Entity Types, deducts the higher aggregation level Entity Types, and assembles the Consumption API Schema
GraphQL Service
Collects the Schemata of Entity Types, deducts the higher aggregation level Entity Types, and assembles the Query API GraphQL Schema.
Handles GraphQL requests by mapping them to a Data Lake Service requests.
Consumption MUX
Handles Consumption requests by mapping the contained Entities to the respective Data Lake Service request
Retrieve Schemata of Entity Types
Store Consumption Service Schema
Retrieve Schemata of Entity Types
Retrieve Application Data
Store Entities
Types Service
Stores and retrieves Schemata of Entity Types
Data Lake Service
Stores and retrieves Application Data
Repository
Mapping to database specific queries
Store Data
Retrieve Data
Database
OpenAPI Schemata
Schemata of Entity Types
Schema of Types API
Schema of Consumption API
Application Data

Figure 5.10: Application Architecture
Source: Own Representation

This section only provides an overview of the application. The following *API & Services* sections cover further details.

The application shall be independent of a specific component model and also open to a variety of data sources. To provide this flexibility and extensibility, the application's data model has to be configurable at runtime. This configurability is offered through the *Types API*, which allows the application administrator to define and adjust the data model to a specific use case. Therefore, *OpenAPI* schemata of entity types have to be sent to an entity type specific endpoint of the *Types API*. These OpenAPI schemata are then validated by the *Validation Service*. Finally, the *Types Service* stores the schemata in the database. Through this *Types API*, the administrator is also able to retrieve existing schemata.

As shown in the figure, the *Types Service* is the central service to store and retrieve schemata from the database. Correspondingly, the *Data Lake Service* is the central service to store and retrieve application data. To make the application rather persistence agnostic, thus, to make it easy to exchange the underlying database, the services' interaction with the database is performed through a *Repository*.

To enforce the data model configured by the administrator, the *Consumption API Schema Assembly Service* retrieves the entity type schemata through the *Types Service* and assembles them to a complete *Consumption API Schema*. This *Consumption API Schema* is then also stored in the database.

The entire *Consumption API* is based on this dynamically, at runtime, constructed schema. Thus, the *Validation Service* validates every request with all entities to be stored against this schema, before it is passed on for further processing. The *Consumption API* exposes only a single endpoint instead of entity type specific endpoints. This makes the upload easier and more flexible from the adapter's perspective, as requests do not have to be routed to a specific endpoint. Furthermore, considering future performance optimizations, this allows sending larger chunks of data at once.

As the entities of different entity types still have to be processed differently, the *Consumption MUX* identifies the type of each entity contained in a request and routes it to the respective *Data Lake Service* function.

Finally, in order to being able to query the stored data based on the data model, the *GraphQL Service* constructs a *GraphQL* schema based on the *OpenAPI* schemata of the entity types. The *Query API* therefore also only exposes a single *GraphQL* endpoint. The queries against this endpoint, thus, against the dynamically constructed *GraphQL* schema, are resolved by corresponding functions of the *Data Lake Service*.

The application architecture is essentially a *layered architecture*. In the traditional sense of a the three principle layers, the *APIs* represent the *presentation logic layer* in above figure. Thus, the *APIs* receive HTTP requests, interpret these requests into actions upon the *domain layer* and return information to the user in form of a HTTP responses [73].

The *Validation Service*, *Types Service*, *Consumption API Schema Assembly Service*, *GraphQL Service*, *Consumption MUX* and *Data Lake Service* are on the *domain logic layer*. Thus, these services validate data that comes in from the *presentation layer* and perform tasks based on inputs and stored data [73].

Finally, the *Repository* is on the *data source logic layer* as it is responsible for communicating with the database to carry out tasks on behalf of the application [73].

Beyond that, these services are layered even further, as the services further left, thus, service on a higher layer, use functionality provided by services further right (relative to themselves), thus, services on a lower layer. But the lower layer services are unaware of the higher level services [73].

5.5.1 Types API & Services

As explained in the overview, every service of the application builds on the schemata which may be created through the *Types API* and respective *Services*. Thus, the rationale behind this service as well as the design decisions involved, are discussed first.

Schema

As the whole complexity of the architecture kind of revolves around the dynamically configurable schemata, a common question especially considering the *schema-less* database may be, why the application even requires such rigid complete schemata?

The term schema-less often leads to the misconception that the data somehow does not need to abide to any schema. Technically, this is true for most databases attributed with the term schema-less, like Neo4j. So the database will allow storing any key-value pairs. The problem is, without prior knowledge about the keys, the data cannot be retrieved. And without prior knowledge about the data types, the data may hardly be aggregated. Thus, even with schema-less databases, *partial schemata* are enforced in most use cases.

In this context, partial schema usually means that the application requires a specific predefined set of properties with particular data types. But the application is not limited to those properties. Therefore, a set of unspecified properties may be stored alongside the data defined by a schema. Such partial schemata may be enforced by the database through constraints or by the application through frameworks, libraries and standards like *JSON Schema* [74] and OpenAPI.

But there is a rather simple reason, why this approach to flexibility and extensibility is not sufficient for this application. The data, thus the entities, are provided by other systems based on policies. Looking back at section 4.4 "Integration of the Security and Compliance Data Lake", the component model data may be fetched from a component repository once a day. Correspondingly, the artifact composition data, thus the packages contained in an artifact, may also be fetched from scanning tools once a day. To determine whether an entity such as a component or package already exists in the database, the application has to know the identity of the respective entity. But different component models may identify components by different sets of properties. Different scanning tools may identify packages by different sets of properties. Furthermore, different *Info Snippets* such as vulnerabilities and license are quite unlikely to be identified by the same set of properties. Therefore, it is also impossible to predefine even a partial schema.

The dynamically configurable schemata of this application are even implemented to be complete. So, the application does not really leverage the flexibility of using a schema-less database. This is because allowing properties that are not defined by a schema often leads to a decreased data quality as it entices to just store everything.

Since properties not included in the schema are also rarely used in queries, they frequently do not add value.

Besides, the real value provided by the application is the queryability of software composition. In the rare occasions where some information may be actually missing, it is easy to manually look it up in the original data source.

Meta Data Model

Providing simple OpenAPI schemata would not be sufficient. A common OpenAPI component schema enables to define objects with a set of properties. Thereby, it may also be specified what data type these properties have and whether they are required [75]. While such a specification is enough for validation purposes, it lacks the information which properties are part of the entity type's identity.

To solve this problem, the *Types API* enforces a *Meta Data Model*, which defines schemata for the entity type schemata. Below listing 5.9 shows the YAML representation of the *Component Type* schema schema.

```
ComponentType:
  type: object
  required:
  - kind
  - version
  - name
  - identity_schema
  - attributes_schema
  properties:
    kind:
      type: string
    version:
      type: string
    name:
      type: string
    identity_schema:
      type: object
    attributes_schema:
      type: object
```

Listing 5.9: Entity Type Schema Schema

Thus, the entity type schemata have to have a *kind*, *version*, *name* and especially an *identity_schema* and an *attributes_schema* property. As the names suggest, the *identity_schema* and the *attributes_schema* of type object are themselves again schemata. The *identity_schema* thereby defines all the properties comprising the identity of the entity type and the *attributes_schema* specifies all other properties. As this is very abstract and meta level, below listing shows an exemplary JSON representation of a request body to create a *Component Version* schema.

```
{
  "kind": "ComponentType",
  "version": "v1alpha1",
  "name": "Component",
  "identity_schema": {
    "type": "object",
    "required": [
      "name",
      "version"
    ],
    "properties": {
      "name": {
        "type": "string",
        "maxLength": 255
      },
      "version": {
        "type": "string"
      }
    }
  },
  "attributes_schema": {
    "type": "object",
    "properties": {
      "labels": {
        "type": "array",
        "items": {
          "type": "object",
          "required": [
            "name",
            "value"
          ],
          "properties": {
            "name": {
              "type": "string",
            },
            "value": {}
          }
        }
      }
    }
  }
}
}
```

Listing 5.10: Request Body for Component Version Schema Creation

So the identity of this *Component Version* entity type is comprised of a *name* property of type string and a maximum length of 255 characters and *version* property also of type string with no further restriction. Both of these properties are required. The only further property in this example is *labels*, which is an array of objects with a required *name* property of type string and a *value* property of any type. Thus, this array may hold arbitrarily nested objects.

The other properties provided in this request, so *kind*, *version* and *name*, are not part of the entity types schema. These properties are used for processing logic.

kind describes the *class of entity types*. Based on the data model defined in the section 5.2, this is necessary for *Native Package Version* and *Info Snippet* where multiple entity types of the same class of entity types, or rather *kind*, may be created. In order to have uniform schemata but also especially for extensibility reasons, this property is required to be included with all entity type schemata. This way, the application could be easily extended to store multiple component models.

version describes the version of the particular entity type schema. This property theoretically provides the possibility to update the entity type schema in case an additional property is required. But currently, the application does not provide any functionality to support such version upgrades by automatically updating the existing entities of that entity type.

name is the actual name of the entity type. Thereby, every node of that entity type will be labeled with this name (correspondingly, in a relational database, this would make up the name of the table). Thus, the *name* is required when querying for nodes of that entity type in the database.

Currently, these schemata can only be provided for the lowest aggregation level entity types, or rather classes of entity types. Looking back at the data model in figure 5.1, this means only the schemata of *Component Version*, *Resource Version*, *Source Version*, *Native Package Version* and *Info Snippets* may be defined.

This is because these are the types of entities that are provided by the component model and data sources, thus, the types of entities that are actually consumed. The higher aggregation level entities are rather abstract groupings of those entities, thus, aggregations. As discussed in the data model section, these entities therefore mostly store information that has to be manually created by the user such as *triage policies* or information that is merged from lower level entities such as the common properties of *Native Package Versions* in *Package Versions*. Furthermore, especially their identities depend on the lower level entity's identity.

For example, the identity of a *Component* is equal to the identity of a *Component Version* without considering the version property. Similarly for all other entities with a corresponding relationship. Therefore, although this unfortunately is and cannot be represented in the OpenAPI schema of the entity types schema, the application enforces all the entity type's *identity_schemas* except the ones' of *Info Snippets* to provide a *version* property. While consuming and storing entities through the *Consumption API*, the higher aggregation level entities are then created implicitly with the respective identity.

The *Consumption API Schema Assembly Service* which builds the whole *Consumption API* schema retrieves the created schemata of these lowest level entity types and assembles *identity_schema* and *attributes_schema* to a single component schema for the entity type as shown in the listing below.

```
"Component.v1alpha1": {
  "type": "object",
  "properties": {
    "metadata": {
      "type": "object",
      "required": ["kind", "type", "version"],
      "properties": {
        "kind": {
          "type": "string",
          "pattern": "^ComponentType$"
        },
        "type": {
          "type": "string",
          "pattern": "^Component$"
        },
        "version": {
          "type": "string"
        }
      }
    },
    "identity": {
      "type": "object",
      "required": ["name","version"],
      "properties": {
        "name": {
          "type": "string",
          "maxLength": 255
        },
        "version": {
          "type": "string"
        }
      }
    },
    "attributes": {
      "type": "object",
      "properties": {
        "labels": {
          "type": "array",
          "items": {
            "type": "object",
            "required": ["name", "value"],
            "properties": {
              "name": {
                "type": "string",
               },
              "value": {}
            }
          }
        }
      }
    }
  }
}
```

Listing 5.11: Component Version Schema

So, this is the schema against which entities of type *Component Version* that shall be created through the *Consumption API* are validated. The *metadata* properties are needed by the *Consumption MUX* in order to route the entity to the appropriate *Data Lake Service* function.

This schema creation process is almost identical for all entity types, or rather classes of entity types, except for *Native Package Version.* As common properties of different *Native Package Version*, for example of BDBA and Mend, shall be merged on the *Package Version* aggregation level, all *Native Package Versions* have to share this set of common properties. Therefore, this creation process is slightly more modular, as will be shown in the next section discussing the endpoints.

Furthermore, there are currently predefined schemata for the *Relationships* between the entity types. These schemata are all quite similar. They all require a *source*, which refers to the identity of the entity type for which the relationship is outgoing, and a *destination*, which refers to the identity of the entity type for which the relationship is incoming. Therefore, the schema for a *Component Reference*, thus, for a *Component Version to Component Version* relationship requires the identity of a *Component Version* as a *source* and as a *destination.* Correspondingly schema for a *Component Version to Resource Version* relationship requires the identity of a *Component Version* as a *source* and the identity of a *Resource Version* as a destination. This is necessary in order to create the relationships independently of the entities which makes the creation process even more flexible.

Imagine each *Component Version* schema also includes references to all related *Component Versions*. As entities, or rather nodes, have to exist before relationships including them may be created, this would require to resolve these dependencies and create a corresponding execution order. While the same may be done with the above approach, it also offers the possibility of creating all entities first and then all relationships. Of course, this comes at the cost of having to send the identities redundantly.

Besides that, all *Relationship* schemata only have two further properties, the *labels* property, which allows arbitrarily nested objects and a *name* property of type string. As mentioned before, Neo4j can store JSON documents. But as each node and relationship only stores key-value pairs, nested properties are stored as string and cannot be leveraged in queries (except for using string comparisons, which is quite inefficient).

Endpoints

The *Types API* exposes the following HTTP endpoints:

```
/types/component

/types/resource

/types/source

/types/info-snippets

/types/packages
/types/package-identity
/types/package-attributes
/types/package-native-attributes
```

Listing 5.12: Types API Endpoints

Each endpoint accepts HTTP POST, PUT, GET and DELETE requests[8] for managing one *kind* of schemata, the `/types/component` endpoint for *Component Version* schemata, the `/types/resource` endpoint for *Resource Version* schemata, the `/types/source` for *Source Version* schemata, the `/types/info-snippets` for *Info Snippets* schemata and the other four endpoints for *Native Package Version* schemata.

As indicated in the previous section, the creation process of the *Native Package*

[8]The current version of the application at the time of writing the thesis actually just supports POST requests to those endpoints to create the schemata. But it is planned to extend this to also support GET request to retrieve the respective schemata and also PUT and DELETE requests to edit accidental mistakes in newly created schemata that are not used yet.

Version schemata is similar to the *Component Version* example, but more modular. Therefore, the *identity_schema* and the *attributes_schema* have to be created in separate requests to separate endpoints. Moreover, the *attributes_schema* is further broken down into *attributes* and *native attributes*, where *attributes* is the set of properties that may be merged in the corresponding *Package Version* entity. The individual schemata are then tied together by a request to the `/types/packages` endpoint, referencing the respective schemata *identity*, *attributes* and *native attributes schemata*.

Finally, an open question may be, why even expose multiple endpoints? Especially considering that the *kind* is present in the request anyway, as shown in listing 5.10. Thus, the further routing could also be handled by a custom multiplexer evaluating that property.

The primary reason that such a single endpoint approach is taken for the *Consumption API* but not for this *Types API* is that they are used completely differently.

The *Consumption API* only has to *consume* huge amounts of data that are provided by other systems. The *Types API* is used by a human administrator to configure the application by *creating*, *reading* and in edge cases also *updating* and *deleting* schemata. Also, this is probably done only a few times during the entire application life time. Therefore, considering the entity type schemata as REST resources and exposing corresponding endpoints allows to leverage the intuitiveness of a REST API leveraging the common HTTP verbs.

Besides, there is another inconvenience with APIs that expose only a single endpoint which enables to submit multiple different entity types in a single request.

OpenAPI has a construct called "oneOf" which allows to specify that all entities and relationships in the request have to fulfill exactly *one of* a specified set of schemata. Thus, generally OpenAPI supports specifying that kind of endpoint. The problem is, validation functions provided by OpenAPI libraries in any programming language cannot produce helpful error messages for that kind of endpoint when trying to validate the whole request.

Imagine a request contains multiple entities. One of them is faulty. Now, the general validation function does not know what kind of entity this is supposed to be, thus, against which specific schema it has to be validated. It only knows that this faulty entity does not fulfill any of the provided schemata.

Therefore, in order to get helpful error messages, the validation would have to be implemented at a later point, evaluating the *kind* property and calling the validation function individually for each entity in the request with the corresponding schema. But this is more of an implementation detail.

5.5.2 Consumption API & Services

The *Consumption API* is fairly simple, once the *Types API & Services* are understood. It exposes a single HTTP endpoint:

```
/datalake
```

Listing 5.13: Consumption API Endpoint

This endpoint only accepts HTTP PUT and DELETE requests[9]. The operation performed by the application when receiving a PUT request is *merging*, or in the SQL context rather known as *upsert*, of the respective entities and relationships. The request is validated against the *Consumption API Schema* assembled by the *Consumption API Schema Assembly Service.*

A REST API approach similar to the *Types API* would not add much value here. The API has to consume huge amounts of data provided by other systems. Multiple endpoints would result in additional unnecessary requests. A single endpoint, on the other hand, makes the implementation of the adapters simpler and entities in a request may be bundled more flexibly.

The downside of such an endpoint, as previously mentioned, are having to deal with the obscure error messages or alternatively implementing a more complex validation. But this is more of an implementation detail and the effort to implement the more complex validation is comparably low considering the rest of the application.

Furthermore, such an API also requires additional efforts, as a custom multiplexer has to be implemented to handle the routing which is commonly done by libraries and frameworks based on the URL. Popular examples therefore are *Java's Spring* [76] or *Javascript's Express* [77]. In the context of the *Consumption API*, this routing is handled by the *Consumption MUX* individually for each entity in the request based on its *kind* property.

5.5.3 Query API & Services

While the *Types API* and *Consumption API* may be interesting to administrators and adapter developers, the *Query API* is the part of the application which actually provides value. Therefore, the *Query API* and respective *Services* have to enable

[9]The current version of the application at the time of writing the thesis actually just support PUT requests. But as in production there will likely be cases where some entity or relationship has to be removed, it is planned to extend this to also support DELETE requests

users to conveniently perform queries and analysis by traversing to answer questions such as "Which Component Versions contain the Log4J Vulnerability?".

A brief summary of the API development process may help in understanding the design decisions that lead to the current *GraphQL API*.

REST API

REST APIs are currently probably the most popular types of APIs. This is due to their simplicity and intuitiveness leveraging the built in semantics of the web, the HTTP verbs, and mapping the main entity types of an application to resources and exposing them through individual endpoints.

Therefore, the initial idea was to design a single REST API for *querying* as well as *consuming* the data. Thus, at that point a separation of *Consumption API* and *Query API* was not planned yet.

There were several different approaches for endpoint design. There was the common mapping of entity types and relationships to resources and corresponding endpoints. And there were several different deviations of the common REST approach, trying to deal with the same *three main problems* the classic REST API is facing in this use case.

The below listings outlines the endpoints of a common mapping of the entity types and relationships to resources for the example of the component entity types:

```
/components
/components/{component_name}
/components/{component_name}/versions
/components/{component_name}/versions/{version}
/components/{component_name}/versions/{version}/component_version_references
/components/{component_name}/versions/{version}/source_version_references
/components/{component_name}/versions/{version}/resource_version_references
```

Listing 5.14: REST API Endpoints

There is a problem arising from such common REST API design regarding the *data consumption*. Theoretically, to create *Component Versions*, the adapter would have to send a *huge number of separate requests* and route them to the correct endpoints. Thus, to merge a version v0.15.0 of the component named *github.com/gardener/etcd-druid*, a PUT request would have to be sent to `/components/github.com/gardener/etcd-druid/`. To merge the relationships, further PUT requests would have to be sent to the respective endpoints. This would result in a lot of separate requests. The relationships could not be created before the *Component Version* exists. Thus, even the sequence of the requests is important. Therefore, this approach is impractical.

But, considering that the data sources only provide the lowest aggregation level entities, thus, *Component Versions*, and the respective higher aggregation level entities, thus, *Components*, usually are created implicitly anyway, a deviation from the REST standard for data consumption was deemed acceptable. Therefore, to solve this problem, all *Component Version* entities could just be sent to the `/components` endpoint with the references as nested objects.

There is another problem concerning the *queryability* of this REST API. Designing endpoints like this does enable to retrieve all information about the individual entities, but it does not provide a possibility to traverse the relationships with a single request. As the relationship itself is not really interesting for most queries, this problem could be solved by adding following endpoints:

```
/components/{component_name}/versions/{version}/component_versions
/components/{component_name}/versions/{version}/source_versions
/components/{component_name}/versions/{version}/resource_versions
/components/{component_name}/versions/{version}/package_versions
/components/{component_name}/versions/{version}/vulnerabilities
/components/{component_name}/versions/{version}/licenses
...
/components/{component_name}/components
/components/{component_name}/sources
/components/{component_name}/resources
/components/{component_name}/packages
/components/{component_name}/vulnerabilities
/components/{component_name}/licenses
...
```

Listing 5.15: REST API Query Endpoints

So, the REST API would also enable to traverse the graph and intuitively answer the most common questions. Through URL parameters, further filtering of the corresponding result set would also be possible.

This deviation of a REST API seemed simple and intuitive and still powerful. But there is a third problem, *parsing*. In the end, this also lead to the final reason for not pursuing this approach further.

Part of the problem is already visible in the data consumption example. Common routing frameworks and libraries are not able to handle URLs such as `/components/github.com/gardener/etcd-druid/` since the slashes in the name will be interpreted as a path separator.

Intuitively, this could be solved by replacing the slash by a different character such as an underscore. But for this to work properly, the respective character cannot be allowed to be used in the name. This would definitely cause compatibility issues with OCM and other component models.

Another approach is to implement a custom parser. But in order for this to work, the URLs have to be adjusted to always end on a keyword. For example, the previous URL to retrieve a component with a specific name would have to be changed to something like `/components/github.com/gardener/etcd-druid/list` with *list* being the keyword. Then a custom parser could parse the URL backwards.

But even with URLs always ending on keywords, the parsing may not be unambiguous, as shown by the listing below.

```
/components/github.com/gardener/etcd-druid/versions/1.2.3/list
```

Listing 5.16: Ambiguous REST API URL

This request could be meant to list all *Component Versions* with *component_-name = github.com/gardener/etcd-druid*. But it could also be meant to list all *Components* with *component_name = github.com/gardener/etcd-druid/versions/1.2.3*.

The problems become even more difficult to deal with when considering corresponding endpoints for *Sources* and *Resources*. Using the identity properties of the OCM, respective endpoints would look like this:

```
/sources/{component_name:component_version:source_name:extra_id}
```

Listing 5.17: REST API Source Endpoint

While this is not really clean, it still seems as this may work as long as the character separating the individual properties of the sources' identity, in this example the colon, is guaranteed to not occur within one of the properties. But as previously established, such a separator would lead to compatibility issues. Besides, per OCM specification, the *extra_id* is a flat map containing an arbitrary number of key-value pairs. Thus, the key-value pairs would have to be included in the URL.

Therefore, it is apparent that there is no clean solution to identifying entities by integrating their dynamically configurable identities into the URL.

To solve this parsing issues, assigning additional *surrogate keys*, thus *UUIDs* or *auto incremented IDs*, as it is done in common REST API use cases was considered. But this would require additional requests to find the surrogate key of the respective entity. Thus, the API would have to support requests using query parameters as indicated by the listing below.

```
/sources?component_name=github.com/gardener/etcd-druid&version
   =0.15.0&source_name=etcd&extra_identifier=etcd-druid-source

/sources/{id}/versions
```

Listing 5.18: Querying for Entitiy ID

The user first sends the first request to retrieve the ID of the respective *Source*. Once the ID is acquired, it may be used to navigate further through the API and, for example, retrieve all versions of the *Source* with that ID.

Although every API that exposes its operations through individual resource endpoint is called REST API today [78], this form of usage is already much closer to the original REST architectural style defined by Roy Fielding. It is derived from the web and promotes hypermedia as the engine of application's state [79]. Thus, the original idea is that RESTful architectures and consequently also REST APIs may be navigated just as websites. So a user may know the host address of the API. Sending a GET request to that host address will then return a list hyperlinks of further resources. For this API this may be `/components`, `/sources`, ... Again, following one of those hyperlinks by sending a GET request will return another list of hyperlinks of further resources available under the currently selected resource, in the case of `/components` a list of hyperlinks to all available individual *Components*. This is great for APIs that have a potentially huge number of unknown users, as it is very intuitive and requires almost no documentation to navigate through, just as a website. But it is not suitable and creates a lot of overhead for this kind of application which requires running specific potentially expensive queries based on already known entities.

Finally, hashing the identity properties and using the hash as ID was considered. While this would allow to issue the queries directly without having to retrieve the ID first, it would require to calculate the hash. Due to implications regarding normalization, this would require an additional small program in order to being able to use the API. Therefore, this approach also was not pursued further.

Remote Procedure Calls

As the REST API approach was deemed unsuitable after thorough analysis, the next best idea was to implement the most common queries as *Remote Procedure Calls (RPC)*.

This approach is simple and clean. The application specifies a set of functions, or rather procedures, that may be called through the API. Furthermore, it defines which properties have to be provided for each procedure and the format in which these properties have to be provided. Typically, those procedures are called by HTTP POST requests, specifying the procedure to be called as well as the corresponding properties in the corresponding format in the request body. Thus, there would be no issues regarding embedding properties in the URL and parsing.

While implementing the most common queries is convenient, it is also quite inflexible. An intuitive, more flexible approach which gives users the full capability to run precise and complex queries through the API and answer questions such as

"Which Package Versions depend on the spring-context:1.2.12 Package Version?" is indicated by the listing below.

```
POST /processCypher(
 "MATCH (p1:PACKAGE_VERSION {Name:"spring-context",Version:"1.2.12"
   })
  MATCH (p2:PACKAGE_VERSION)
  WHERE (p1)<-[:DEPENDS_ON*1..]-(p2)
  RETURN (p2)"
)
```

Listing 5.19: Exposing Cypher through API

Thus, exposing the entire database query language through the API. But compared to the REST API, where answering such a question did not require any knowledge about graph databases, for this approach the user has to know the entire database specific query language. And even then, the queries may be quite error prone. Besides, there are several security implications which make this approach impractical.

The basic idea of exposing a query language through the API is still powerful. But the set of available operation has to be smaller and the permissions stricter than the ones' of the database query language. Thus, in the past there were a few attempts to design such API query languages - for example SPARQL, FIQL and a lesser-known one by Meta, FQL [80]. While these API query languages may be unsuitable for graph traversal, designing a language tailored to this application would be an option. But this would be quite time intensive and complex. Besides, the general problems with such API query languages are similar to exposing the database query language. Users have to be skilled in those languages in order to being able to use the API properly.

Thus, these RPC approaches were not satisfying either.

GraphQL

On a basic level, a *GraphQL API* is also essentially RPC, as *GraphQL* is also query language for APIs [81] designed by Meta. Thereby, a GraphQL API exposes a single endpoint which is typically called `/graphql`.

But GraphQL has a special approach to specifying APIs and defining queries, which makes it very intuitive and easy to use. A GraphQL API is defined by *types* which correspond to resources in a REST API, and *fields* on this *types.* To specify those *types* and *fields*, GraphQL provides its own *GraphQL schema language.* Listing 5.20 shows an exemplary GraphQL schema language representation of a *Component*

Version based on the OCM data model instance in section 5.2.3 "Application of the Data Model".

```
type Query {
  Component(Identity: ComponentIdentityArgs): [Component]
  ComponentVersion(Identity: ComponentVersionIdentityArgs, Attributes:
    ComponentVersionAttributesArgs): [ComponentVersion]
  ...
}
type ComponentVersion {
  Identity: ComponentVersionIdentity
  Attributes: ComponentVersionAttributes
  IncomingReferences(type: String, name: String, labels: String): [Reference]
  OutgoingReferences(type: String, name: String, labels: String): [Reference]
  ReferencedComponents(Identity: ComponentIdentityArgs): [Component]
  ReferencedComponentVersions(Identity: ComponentVersionIdentityArgs, Attributes:
    ComponentVersionAttributesArgs): [ComponentVersion]
  ReferencedResources(Identity: ResourceIdentityArgs): [Resource]
  ReferencedResourceVersions(Identity: ResourceVersionIdentityArgs, Attributes:
    ResourceVersionAttributesArgs): [ResourceVersion]
  ReferencedSources(Identity: SourceIdentityArgs): [Source]
  ReferencedSourceVersions(Identity: SourceVersionIdentityArgs, Attributes:
    SourceVersionAttributesArgs): [SourceVersions]
  ContainedPackages(Identity: PackageIdentityArgs, Attributes: PackageAttributesArgs): [Package]
  ContainedPackageVersions(Identity: PackageVersionIdentityArgs, Attributes:
    PackageVersionAttributesArgs): [PackageVersion]
  ContainedBDBAPackageVersions(Identity: BDBAIdentityArgs, Attributes: BDBAAttributesArgs): [
    BDBAPackageVersion]
  ContainedVulnerabilities(Identity: VulnerabilityIdentityArgs, Attributes:
    VulnerabilityAttributesArgs): [Vulnerability]
}
type ComponentVersionIdentity {
  name: String
  version: String
}
type ComponentVersionAttributes {
  labels: String
}
...

input ComponentVersionIdentityArgs {
  name: String
  version: String
}
input ComponentVersionAttributesArgs {
  labels: String
}
...
```

Listing 5.20: GraphQL Component Version Schema

The GraphQL schema language is quite intuitive. The *Query type* is a special type which defines the entry point for GraphQL queries, listing all *types* a user may query for. These available *types* further specify their fields and therefore declare what may be queried on them, and so on. Thereby, the GraphQL schema resembles a REST API which properly implements hypermedia as the engine of application state. But instead of having to send multiple queries to navigate through the API step by step and actually retrieve data, with GraphQL the entire schema may be retrieved at once. This reduces the overhead in requests and data transfer significantly.

The *Component Version object type* is a field within the *Query type* and has two *input objects* as arguments, the *ComponentVersionIdentityArgs* object and the *ComponentVersionAttributesArgs* object, defined at the bottom of the listing. *[ComponentVersion]* thereby represents an array of *Component Version* objects.

The basic structure of the previously described REST API and of the GraphQL API defined by this schema is quite similar. Both hide the relationship traversals necessary to answer questions such as "Which Vulnerabilities are contained in Component Version "github.com/gardener/etcd-druid:0.15.0"?". Instead, for the user of the API, it looks and feels like *Component Versions* are stored like documents in a document database with all related entities directly nested inside of them. Thus, a GraphQL query to answer the respective question based on the above GraphQL schema looks like this:

```
{
  ComponentVersion(Identity: {name:"github.com/gardener/etcd-druid", version: "0.15.0"}) {
    ContainedVulnerabilities {
      Identity {
        cve
      }
      Attributes {
        summary
        cvss
      }
    }
  }
}
```

Listing 5.21: GraphQL Query

In reality, the application assigns functions to the respective fields on a GraphQL schema. These functions generate a corresponding *cypher* query to retrieve the data from the database and resolve the request.

As shown in listing 5.21, the query language even allows to specify which properties of the *Vulnerabilities* shall be retrieved. While this is not really significant for this application, compared to a REST API this may reduce the overhead even further by exactly describing what shall be queried and retrieved.

More importantly, the GraphQL query language gets rid of parsing issues as the data model and queries do not have to be mapped to URL paths. Additionally, it is also more powerful than the REST API, as it supports arbitrary nesting of the queried objects. Thus, considering above query, it is also possible to specify to retrieve all packages the respective *Vulnerability* occurs in. Depending on the exact GraphQL schema of the *PackageVersion type* the respective query would look something like this:

```
{
  ComponentVersion(Identity: {name:"github.com/gardener/etcd-druid", version: "0.15.0"}) {
    ContainedVulnerabilities {
      Identity {
        cve
      }
      Attributes {
        summary
        cvss
      }
      PackageVersions{
        Attributes {
          name
        }
      }
    }
  }
}
```

Listing 5.22: Nested GraphQL Query

Thus, as GraphQL provides a clean, intuitive and powerful API, this is the approach that was finally selected for the Query API. GraphQL also has capabilities to manipulate data, but for this application, these are also less suitable than the *Consumption API*.

As already mentioned, the application cannot define the GraphQL schema statically. Instead, the schema has to be created and adjusted dynamically by the application based on the OpenAPI schemata created through the *TypesAPI*. This is the biggest downside of this approach, as this is cumbersome to implement and adds a lot of complexity to the application.

5.5.4 Limitations and Problems

The current application design as presented in the previous sections is not flawless and has limitations.

The main limitation of the current design is the *lack of explicit schemata for higher aggregation level entity types* and the *implicit creation of the respective entities*. Generally, this is rather convenient for administrators, adapter developers as well as users of the API, as the ability to query the software composition on multiple aggregation levels comes seemingly for free. The administrator does not have to configure respective schemata for *Component*, *Source*, *Resource*, *Package*, *Package Version* and *Relationships*. Instead, only the schemata of entity types he is actually dealing with have to be configured, as the respective entities are provided by the component model and other data sources. And the adapter developer does not have to deduct these entities and send them to the API as this is done implicitly by the application itself. So, for most use cases, this *hides complexity* and may even be seen as an *advantage*.

But answering questions about the software composition is not the sole purpose of the application. Looking back the requirements in sections 5.1, the application shall also enable assessments (R.6). The data model section 5.2 therefore mentioned that *triage policies* may be defined on *Sources* and *Resources* and the actual *triage* information shall be stored on the relationships such as between *Component Versions* and *Source Versions* or *Resource Versions*. Furthermore, these higher aggregate level entities shall be able to store semantic information about the grouping. These capabilities are not enabled by the current application design.

There are essentially two approaches to adjusting the application to support this functionality. One approach would be to explicitly deduct the *identity_schemata* for higher level entity types such as *Component*, thus, the *Component Versions identity_schema* without the *version* property, and expose additional *Types API* endpoints to add *attrite_schemata*. Correspondingly, *Types API* endpoints may be added to define the schemata of *Relationships*. This approach matches the general extensibility and flexibility of the application, but also requires the most efforts to implement and adds a substantial amount of complexity for the administrators.

The other option is to statically predefine the properties available for *triage policies* and *triaging*. To add general semantic information, a *labels* property which may store arbitrary key-value pairs may be added. This approach is quite low effort and does not add any complexity, but it is also significantly more rigid. Also, the *labels* property does not enable proper filtering upon the nested properties.

As the *Consumption API* is tailored towards handling huge amounts of data provided by other systems, an additional *Triage API* may be added to manually store this information.

Furthermore, the flexibility and extensibility of the application comes at the cost of *complexity*. This is partially represented in the previous sections, especially the *Types API & Services* with its *Meta Data Model*. But it is also especially represented in the application's code. The code dealing with schema creation and translation, thus, the code making up the *Consumption API Schema Assembly Service* and *GraphQL Service*, is quite lengthy. As the data types are not predefined, the whole application code has to work extensively with abstract data types such as maps[10], while leveraging reflection and casting. Furthermore, the whole processing logic had to be designed to being able to handle this level of flexibility. Even data base queries are constructed dynamically based on the respective configured schemata. This makes *application maintenance* more difficult and error prone. Moreover, the implications of this amount of flexibility and extensibility for the *application's security*, especially also concerning attacks like "SQL injection", have not been sufficiently analyzed and estimated so far.

[10]The name of this abstract data type varies between programming languages. Other common terms are associative array or dictionary.

Chapter 6

Result

6.1 Evaluation

6.1.1 Requirements

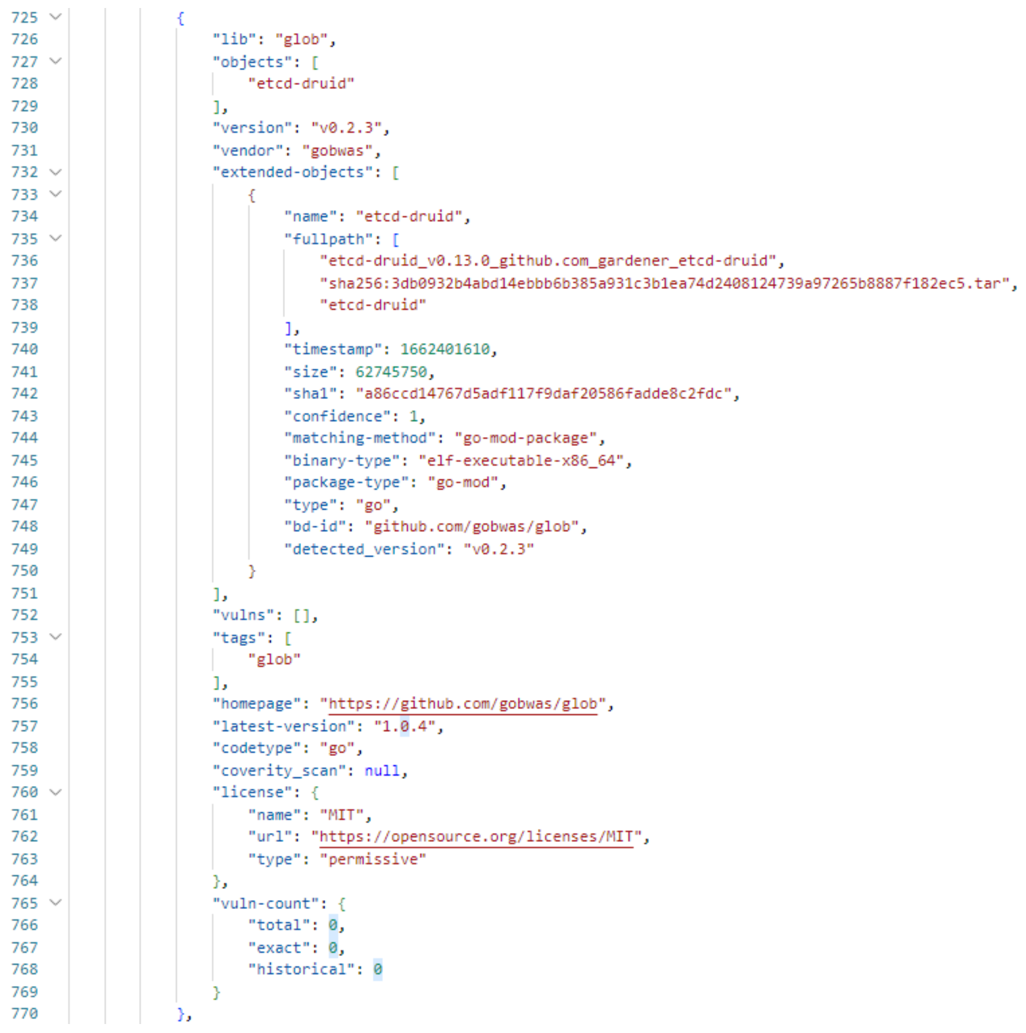
Another limitation concerns requirement R.7 data aggregation and filter functions. Generating a R.8 SBOM format
–> these limitations concern functionality is just not implemented yet but may easily be added without any major adjustments to the architecture/design, thus not mentioned in the application architecture chapter

6.1.2 Research Question

Appendix A

Additional Information

A.1 Black Duck Binary Analysis Result



```
{
    "lib": "glob",
    "objects": [
        "etcd-druid"
    ],
    "version": "v0.2.3",
    "vendor": "gobwas",
    "extended-objects": [
        {
            "name": "etcd-druid",
            "fullpath": [
                "etcd-druid_v0.13.0_github.com_gardener_etcd-druid",
                "sha256:3db0932b4abd14ebbb6b385a931c3b1ea74d2408124739a97265b8887f182ec5.tar",
                "etcd-druid"
            ],
            "timestamp": 1662401610,
            "size": 62745750,
            "sha1": "a86ccd14767d5adf117f9daf20586fadde8c2fdc",
            "confidence": 1,
            "matching-method": "go-mod-package",
            "binary-type": "elf-executable-x86_64",
            "package-type": "go-mod",
            "type": "go",
            "bd-id": "github.com/gobwas/glob",
            "detected_version": "v0.2.3"
        }
    ],
    "vulns": [],
    "tags": [
        "glob"
    ],
    "homepage": "https://github.com/gobwas/glob",
    "latest-version": "1.0.4",
    "codetype": "go",
    "coverity_scan": null,
    "license": {
        "name": "MIT",
        "url": "https://opensource.org/licenses/MIT",
        "type": "permissive"
    },
    "vuln-count": {
        "total": 0,
        "exact": 0,
        "historical": 0
    }
},
```

Figure A.1: Snippet of BDBA Analysis Result
Source: [34]

A.2 Relational Model of the Example Data Model Instance

This list shows a exemplary relational model corresponding to figure 5.2 in section 5.2.3. The properties are deducted from the OCM, but as stressed several times, they are just one possible example. As this relational model is mainly supposed to support the general understanding of the problem domain, it treats each attribute as atomic. Thus, complex attributes such as repository context are not further normalized for the scope of this relational model.

Primary key attributes are bold and underlined and *foreign key attributes* are italic and underlined. Naturally, in this data model which rarely uses surrogate keys, but still has to represent several hierarchical relationships, attributes are frequently part of both, the primary and the foreign key.

- **Component**(**name**)
- **ComponentVersion**(***name***, **version**, labels, repository-context, provider)
- **ComponentReference**(**source-component-name**, **source-component-version**, **destination-component-name**, **destination-component-version**, **name**, **extra-identity**, labels)
- **Resource**(**name**, **extra-identity**, ***component-name***, ***component-version***)
- **ResourceVersion**(***name***, **version**, ***extra-identity***, ***component-name***, ***component-versi*** digest, labels, resource-type)
- **Source**(**name**, **extra-identity**, ***component-name***, ***component-version***)
- **SourceVersion**(***name***, **version**, ***extra-identity***, ***component-name***, ***component-version*** labels)
- **IsBuiltFrom**(**component-name**, **component-version**, **resource-name**, **resource-version** **resource-extra-identity**, **source-name**, **source-version**, **source-extra-identity**)
- **Package**(**id**, name)
- **PackageVersion**(**id**, *package-id*, name, version, ecosystem, platform)
- **BDBAPackageVersion**(**name**, **version**, *package-version-id*, ecosystem, platform, matching-method)

- **SourceVersionIsComprisedOf**(**component-name**, **component-version**, **source-name**, **source-version**, **extra-identity**, **package-version-id**)
- **ResourceVersionIsComprisedOf**(**component-name**, **component-version**, **resource-name**, **resource-version**, **extra-identity**, **package-version-id**)
- **DependsOn**(**package-version-id**, **package-version-id**)
- **Vulnerability**(**cve**, summary, cvss, cwe)
- **Contains**(**package-version-id**, **cve**)

The IsBuiltFrom-relationship contains a particularly interesting detail about this mapping. As explained in the respective sections, due to the *Component Version-Local Identities* of *Artifacts* in the OCM, *Resource Versions* may only be built from *Source Versions* within the same *Component Version*. Thus, even though the component name and component version are part of the identities of both *Source Version* and *Resource Version*, it is only contained once in the IsBuiltFrom relation manifesting this (n:m)-relationship.

Another detail, of which the purpose may not be immediately obvious, are the surrogate keys, id, of *Package Version* and *Package*. The scanning tools and other data sources may use different identifiers for the same technical package or even identify different technical packages with the same identifier. Thus, the decision, that for example a *BDBA Package Version* and a *Mend Package Version* actually represent the same technical package has to be done by humans. Until then, the *BDBA Package Version* and the *Mend Package Version* have to be stored without merging and therefore relating to two different *Package Versions* and *Packages*. With natural keys, this could therefore lead to duplicate *Package Versions* and *Packages*.

A.3 Relation of Number of Keys and Number of Leaves in a B-tree

Given $N+1$ is the number of leaf nodes of a B-tree. $n_l \epsilon \mathbb{N}$ is the number of nodes on a particular level $l \epsilon \mathbb{N}$. Thereby, $l = 1$ is the lowest level, thus the leaf node level, and consequently n_1 is the number of leaf nodes, thus $n_1 = N + 1$. As a B-tree is a tree, it always has a root node. So $l = root$ is the highest level with $n_l = 1$.

As per constraint, the number of keys in a node is $k - 1$. Since all the nodes on $l = 1$ are children of some node on $l = 2$ and every node on $l = 2$ has one key less than it has children, the total number of keys on l_2 is equal to $K_2 = n_1 - n_2$. This correlation is generally true, the number of keys on a level is equal to the number of nodes on the level below minus the number of nodes on the respective level (with the only exception of $l = 1$).

$$K_l = n_{l-1} - n_l$$

The resulting sequence $(K_l)_{l=2}^{root}$ can be used to form a series calculating the cumulative number of keys up to a particular level.

$$\Sigma K_l = \Sigma_{i=2}^{l} n_{i-1} - n_i$$

And as every tree has a root node, n always reaches 1 at some point. This can be written as follows.

$$\Sigma K_{root} = (n_0 - n_1) + (n_1 - n_2) + (n_2 - n_3) + ... + (n_{l-1} - 1)$$

Obviously, all elements but the first, n_0, and the last, 1, appear twice, once with a positive and once with a negative sign and thus reduce each other.

$$\Sigma K_{root} = n_0 - 1$$

And since $n_0 = N + 1$, the number of keys in a B-tree is

$$\Sigma K_{root} = N + 1 - 1 = N$$

A.4 Data Definition Statement for SQL Sample Data

```
CREATE TABLE PackageVersion(
ID int,
Name varchar(100),
Version varchar(100)
);

INSERT INTO PackageVersion VALUES
(1, 'log4j', '1.5.2'), -- depends on spring-context directly
(2, 'spring-context', '1.2.12'),
(3, 'java-jdk', '1.3.1'), -- direct dependency of spring context
(4, 'spring-boot', '1.15.3'), -- depends on spring-context
    transitively (spring-boot -> log4j -> spring-context)
(99, 'spring-core', '2.1.4'); -- direct dependency of spring
    context
```

```
CREATE TABLE DependsOn(
PackageVersionID int,
RelatedPackageVersionID int
);

INSERT INTO DependsOn VALUES
(1, 2),
(2, 3),
(2, 99),
(4, 1),
(99, 3);
```

Listing A.1: DDL Statements

A.5 Data Definition Statement for MongoDB Sample Data

```
db={
  "orders": [{
    "_id": 1,
    "date": "04.01.2023",
    "shippingAddress": {
      "city": "Berlin",
      "street": "Kreuzberg" },
    },
    {
    "_id": 2,
    "date": "05.01.2023",
    "shippingAddress": {
    "city": "Hamburg",
    "street": "Sesame Street" },
  }],
  "relations": [{
    "_id": 1,
    "order": 1,
    "product": 1
    },
        {
    "_id": 2,
    "order": 1,
    "product": 2
    },
    {
```

```
    "_id": 3,
    "order": 2,
    "product": 2
  }],
  "products": [{
    "_id": 1,
    "name": "The Hitchhiker's Guide to the Galaxy",
    "price": 7.50,
    },
    {
    "_id": 2,
    "name": "The Art of Computer Programming",
    "price": 120.00,
  }]
}
```

Listing A.2: JSON

A.6 Data Definition Statement for Neo4J Sample Data

```
MERGE (p1:PACKAGE_VERSION {Name:"log4j",Version:"1.5.2"})
MERGE (p2:PACKAGE_VERSION {Name:"spring-context",Version:"1.2.12"})
MERGE (p3:PACKAGE_VERSION {Name:"java-jdk",Version:"1.3.1"})
MERGE (p4:PACKAGE_VERSION {Name:"spring-boot",Version:"1.15.3"})
MERGE (p5:PACKAGE_VERSION {Name:"spring-core",Version:"2.1.4"})

MERGE (p1)-[r1:DEPENDS_ON]->(p2)
MERGE (p2)-[r2:DEPENDS_ON]->(p3)
MERGE (p2)-[r3:DEPENDS_ON]->(p5)
MERGE (p4)-[r4:DEPENDS_ON]->(p1)
MERGE (p5)-[r5:DEPENDS_ON]->(p3)
```

Listing A.3: Cypher Statements