Friedrich-Alexander-Universität Erlangen-Nürnberg
Technische Fakultät, Department Informatik

MOGILDEA CRISTIAN
MASTER THESIS

DEVELOPMENT OF A PLUGIN ARCHITECTURE WITH DOCKER CONTAINERS

Submitted on 25 March 2021

Supervisors:
Andreas Bauer, M. Sc.
Prof. Dr. Dirk Riehle, M.B.A.
Professur für Open-Source-Software
Department Informatik, Technische Fakultät
Friedrich-Alexander-Universität Erlangen-Nürnberg

Versicherung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, 25 March 2021

License

Erlangen, 25 March 2021

Abstract

Software licenses play a critical role when developing applications containing open source dependencies. To avoid legal risks, a proper management of these dependencies is necessary and with an increasing number of third-party dependencies in commercial products, license compliance becomes very difficult. Product Model Toolkit helps combining multiple license relevant information into one unified model to derive license compliance artifacts, like the software bill of materials.

For data gathering, Product Model Toolkit uses multiple existing license scanners. It encapsulates these tools and their dependencies into separate Docker containers. However, its implementation is only a preliminary solution and it doesn't ensure a straight forward integration of license scanners.

Considering the diverse technologies in which license scanners are implemented, a technology independent approach is necessary to facilitate the integration of such tools by defining them as plugins. This thesis presents a plugin architecture based on Docker containers. By examining known plugin architecture models we identify their common aspects as well as appropriate techniques to build a powerful and efficient Docker-based plugin architecture. Its implementation in the Product Model Toolkit proofs the feasibility of the proposed approach.

Contents

Acronyms

API Application Programming Interface

CI continuous integration

CLI command line interface

DDD Domain-Driven Design

FLOSS Free/Libre Open Source Software

HOCON Human-Optimized Config Object Notation

HTTP Hypertext Transfer Protocol

IDE integrated development environment

JMS Java Message Service

JNA Java Native Access

JSON JavaScript Object Notation

LXC Linux Containers

OpenDDL Open Data Description Language

OSGi Open Services Gateway initiative

PMT Product Model Toolkit

REST Representational State Transfer

RMI Remote Method Invocation

RPC Remote Procedure Call

SBOM software bill of materials

SDK Software Development Kit

SLP Service Location Protocol

SOAP Simple Object Access Protocol

stderr standard error

stdin standard input

stdout standard output

TCP/IP Transmission Control Protocol/Internet Protocol

TTY teletypewriter

UML Unified Modeling Language

UPnP Universal Plug and Play

VM virtual machine

WSDL Web Services Definition Language

XML Extensible Markup Language

YAML YAML Ain't Markup Language

1 Introduction

Software reuse is a common practice in software development and the phenomenon of FLOSS has had a significant impact on this aspect. Today, open source dependencies are widely accepted in software industry. However, this also raises serious concerns. Cox argues that developers "do not yet understand the best practices for choosing and using dependencies effectively, or even for deciding when they are appropriate and when not" [Cox19]. Reuse of open source software involves an ongoing consideration of associated risk factors. One of them is license compliance. Bauer et al. state that proper management of open source dependencies is necessary to avoid legal issues arising from license noncomplience [Bau+20]. The use and development of suitable tools play in this context a key role.

Various highly useful tools such as license scanners are available to allow software vendors extract significant information concerning software dependencies from their applications. Executing each of these tools individually to subsequently benefit from their generated data implies unnecessary effort. An instrument that automates these tasks is therefore beneficial and Product Model Toolkit (PMT) [Ope] proves to be effective in this regard. PMT allows integration of license and other critical information into a unified model. It contains a server application that can generate license compliance artifacts, e.g. software bill of materials (SBOM), and store component graphs into a database. Moreover, PMT also includes a client application that can run license scanners or other useful tools inside Docker containers. It then captures the generated files and sends them to the server. However, the current implementation of the PMT client application is only a temporary solution. It has certain drawbacks which make the application error-prone and unstable.

The goal of this master thesis is to provide a technology independent approach that facilitates the integration of scanner tools into the PMT client application. This implies building an extensible system which allows the user provide additional functionality and enhance the application. The plugin architecture pattern is in this context the optimal approach for building such a system. But this raises a number of questions which are outlined below.

- *What are the principles, characteristics and limitations of the plugin architecture pattern?*
- *How does the interface between the system and plugins look like?*
- *Which responsibilities lie with the system and which with the plugins?*
- *Considering the aspects of Docker container technology, how to define Docker containers that encapsulate scanner tools as plugins?*
- *Which constraints may Docker container technology impose when using the plugin architecture pattern?*

In this master thesis we aim to address all above mentioned questions and finally build a powerful, efficient Docker-based plugin architecture. First, we create an architecture model that serves as a technology independent approach in combining the plugin architecture pattern and the Docker container technology. Second, we implement this architecture model in the PMT client application to prove its feasibility and reveal the capabilities and limitations of the architecture.

1.1 Outline

The outline of this master thesis is based on the structure of an engineering thesis:

- Chapter 2: *State of the Art* explains the basic aspects of the plugin architecture pattern. It also describes the fundamentals of the Docker container technology and highlights the initial state of the PMT client application to have a better understanding of the requirements for the architecture.
- Chapter 3: *Requirements* delineates the requirements for the new Docker-based plugin architecture.
- Chapter 4: *Architecture Design* describes the design phase of the architecture development. It also strictly reflects the established requirements.
- Chapter 5: *Implementation* provides details on the implementation phase of the architecture development including the encountered challenges and the applied solutions.
- Chapter 6: *Evaluation* discusses the fulfillment of the established requirements.
- Chapter 7: *Conclusion* includes final thoughts on the entire development of the Docker-based plugin architecture.

2 State of the Art

In this chapter we discuss multiple prominent plugin architecture systems to understand the common aspects of what a plugin architecture consists of. We select three examples and start with describing their basic concepts in Section 2.1. The capabilities and limitations of the plugin architecture pattern are then outlined in Section 2.2. The fundamentals of Docker technology are explained in Section 2.3 and the initial state of PMT client application is finally described in Section 2.4.

2.1 Current Plugin-Based Systems

Birsan [Bir05] describes in his paper a pure plugin architecture based on experience gained from Eclipse IDE project[1]. The general idea is to decouple an application into multiple standalone plugins. In this case, plugins are no longer typical addons that extend the functionality of a host application, instead, the application consists entirely of plugins as shown in Figure 2.1.

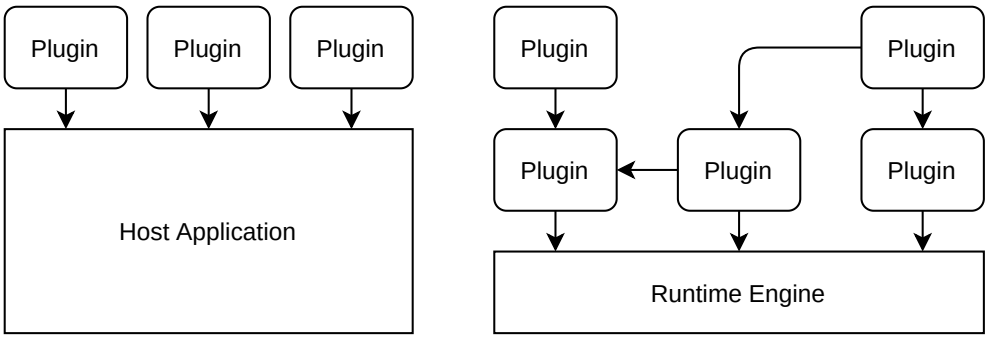

Figure 2.1: Traditional plugin system (left) vs. pure plugin system (right) [Bir05]

This allows to create a flexible and extensible system which is described as follows. The host application or the kernel is merely a runtime engine that is respons-

[1] https://www.eclipse.org/eclipseide/

ible to start and run the plugins, whereas it includes no end user functionality. A framework defines the structure of plugins and how they interact with each other. An important feature of this architecture is the extensibility of plugins, meaning that each plugin can become a host for other plugins. For this purpose, plugins offer hook points or so called *extension points* that have precise definition. This way, it is possible to establish contractual obligations between plugins. To identify the currently installed plugins, kernel maintains a plugin registry that provides information on the installed plugins and their functions. Each plugin is responsible to search, identify and run its extenders.

Richards [Ric15] presents a further architecture model which he refers to as *microkernel architecture pattern*. The basic concepts of this example are similar to those of the first example, whereas the author uses a few distinct terms. This model contains a core system and the plugin modules. The core system is responsible for making the system operational and includes only minimal or general functionality. The plugins include custom code and provide additional functionality. Therefore, it is possible to extend the core system and, as a result, the application becomes more complex. The author suggests to reduce the number of dependencies between the plugins as much as possible to avoid any issues with data exchange. A plugin registry is also present.

Figure 2.2 shows an implementation example of a business application based on the microkernel architecture. In this case, an insurance company needs to process the insurance claims based on individual regulations across different states. The core system contains only essential functionality that is necessary to process a claim, whereas each plugin holds individual rules for a particular state. By decoupling the application, it is possible to create the plugins using custom code separate from the core system. This allows to add, modify or remove state-specific rules without affecting the rest of the application. In case of a single and complex application, this realization would require a considerable amount of effort and resources.

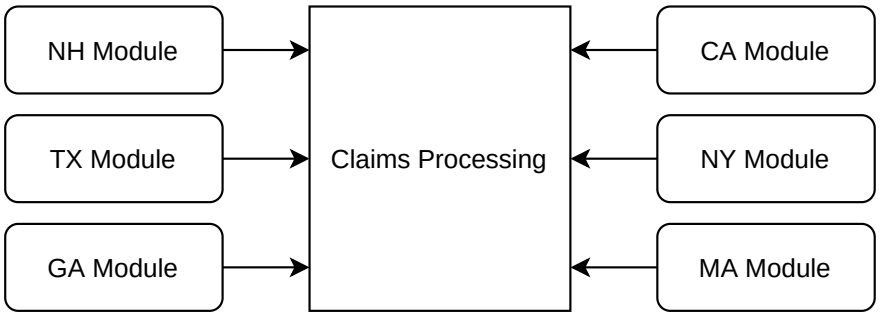

Figure 2.2: Business application based on microkernel architecture [Ric15]

Similar to first example, Wolfinger et al. [Wol+06] also describe a plugin archi-

tecture model based on Eclipse IDE project. The authors use .NET platform to create a "more readable and easier to maintain" [Wol+06] approach by specifying relevant data in source code. So called *extension slots* are hook points that specify how other plugins must extend their functionality. The term *extension* indicates the way a plugin contributes to a certain slot. The specifications of extension slots and extensions must therefore correspond. This model also has a plugin repository where all plugins are located. The system activates them at startup. Authors created a platform called *CAP.NET* to confirm their plugin architectural concepts. Its basic components are depicted in Figure 2.3. For more implementation details see their paper [Wol+06].

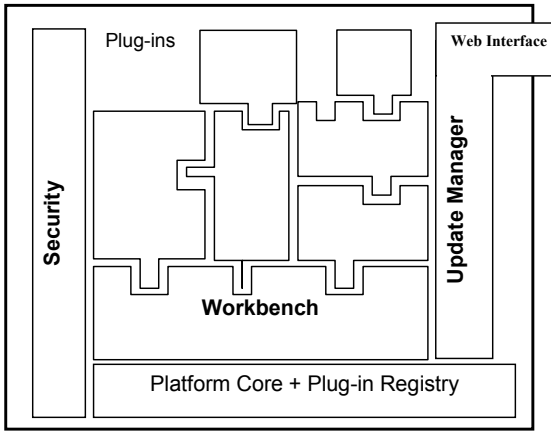

Figure 2.3: Architecture overview of *CAP.NET* platform [Wol+06]

Richards [Ric15] mentions some benefits of using the plugin architecture model. First, a combination with other architecture patterns is practicable. For example, it is possible to embed plugin architecture as a standalone layer in the layered architecture pattern, or use it to create an event processor component in the event-driven architecture pattern. Second, it facilitates the incremental development of software. Creating a core system with general functionality allows to add new features to the program without the necessity of modifying the core system significantly. This way, the application evolves incrementally with reduced effort. Last but not least, it is possible to refactor a program created using the plugin architecture pattern to another pattern whenever necessary.

2.1.1 Common Aspects of Plugin-Based Systems

In this chapter we outline the common aspects among the previously described plugin architecture models in Table 2.1. We select the mutual components and specify each plugin system's approach to identify the similarities and differences between the models.

Common aspects	**Birsan [Bir05]**	**Richards [Ric15]**	**Wolfinger et al. [Wol+06]**
Runtime engine or core system or platform core	Has minimal functionality, e.g. plugin registry initialization, resolving of plugin dependencies; optionally can offer following features: logging, tracing and security; comprises a small bootstrap code and multiple core plugins	Has minimal functionality to ensure that application operates correctly; includes general business logic, whereas no custom code is permitted	Runs plugins in a very controlled, restricted and consistent way; provides a security component for managing rights and roles of plugins; includes life cycle management and update mechanism for plugins
Plugin registry	Contains information on installed plugins and their functions; caching can reduce startup time but makes the application more complex; plugin manifest files offer support for plugin declaration functionality	Contains information on all plugins; includes details like name, data contract and information on remote access protocol; a WSDL may be necessary if the application uses SOAP	No complicated or error-prone configuration; has a central storage called plugin repository; discovers plugins simply by searching plugin repository; provides lazy-loading support
Data exchange	Runtime engine provides an interface which other plugins must implement; plugins can also define extension points	OSGi, messaging, web services, object instantiation; popular data formats: XML or Java Map; adapters for custom data contracts	Slots declare type of information required (names and value ranges); host defines interface and extension contributor provides implementation

Table 2.1: Common aspects among plugin architecture models

2.2 Pattern Analysis

A pattern analysis conducted by Richards [Ric15] indicates the potential of the plugin architecture model, in terms of common architectural characteristics among software architecture patterns. The high or low rating for each characteristic is based on a typical implementation of the model.

- **Overall agility**. Rating: High

  Software evolution is a key aspect of software engineering, as long as software systems have to remain useful through environment transformations. Newly emerged requirements, error detection as well as performance improvement are several reasons why software products have to be changed following the initial deployment [Som11]. Given that a program created using the plugin architecture pattern is decoupled into multiple standalone plugins, it is possible to isolate the changes and implement them faster. In the course of continuous software development, the core system usually becomes stable very soon, therefore requiring only minimal adjustments in the long term.

- **Ease of deployment**. Rating: High

  This can be accomplished by introducing the hot plugging support to load the plugins on the fly. As a result, the core system is not interrupted during deployment.

- **Testability**. Rating: High

  As previously stated, it is possible to isolate the changes to the plugins from the core system. Analogously, standalone testing of the plugins can be well performed, e.g. through mocking the functions of the core system.

- **Performance**. Rating: High

  With a proper implementation, highly customizable programs created using the plugin architecture pattern can achieve high level of performance. By disabling the plugin modules that are no longer needed, CPU and memory resources are used more efficiently by the rest of the program. For instance, WildFly modular application (formerly JBoss AS)[2], enables the user to customize the server and use only the features that are necessary.

- **Scalability**. Rating: Low

  Plugin architecture pattern is typically suitable for small-size applications. As stated above, it is possible to ensure high scalability through extending the plugins, however, this is not a common practice.

[2] https://www.wildfly.org/

- **Ease of development**. Rating: Low

  Considering the different elements and features required to build an application using the plugin architecture pattern, such as the core system, plugin registry, data exchange standards as well as the plugin granularity, makes the development challenging.

2.3 Docker Container Technology

Containerization technology has been around for a long time before Docker containers came into being. A brief overview of its historical development is described by Mouat [Mou15]. At first, a primitive form of file system isolation has been offered by chroot command used on Unix systems, whereas FreeBSD's jail utility has later expanded chroot sandboxing to processes. In 2001, two proprietary containerization solutions were introduced: Solaris Zones as part of Solaris OS and Virtuozzo released by SWsoft (now Parallels). Virtuozzo was later open sourced in 2005 as OpenVZ[3]. Subsequently, Google began developing CGroups and then started running its software in containers. In 2008, Linux Containers (LXC)[4] combined CGroups, kernel namespaces and other mechanisms to offer a full containerization system [Pou19]. Docker ultimately refined the technology in 2013 which has since become conventional.

Simply put, Docker containers may be viewed as a lightweight alternative of virtual machines (VMs), whereas these approaches vary in the degree of virtualization: hypervisor-based methods virtualize at hardware level and containers at OS level [Mer14]. Although hypervisor-based virtualization allows direct access to hardware, an OS is required for each VM, causing performance bottlenecks when running numerous VMs. Docker, on the other hand, "can run hyperscale numbers of containers on a host container because without a hypervisor, they sit right on top of the operating system" [And15]. Compared to VMs, Docker containers perform significantly better [Pot+20].

Besides consuming fewer resources than virtual machines, Docker containers offer further advantages:

- **Portability**

  Constructed Docker images can be copied from one machine to another and run without any compatibility issues, therefore eliminating the famous "but it works on my machine" argument [Mou15].

[3] https://openvz.org/
[4] https://linuxcontainers.org/lxc/

- **User environment consistency**

  Docker allows bundling an application with all its dependencies in a single container. As a result, no further setup and configuration are required prior to executing the application [Mou15].

- **Union file system**

  By using copy-on-write mechanism, Docker is capable to build layers of file systems and combine them to create a single instance of a file system. If something needs to be added, e.g. a new application, another layer is added on top instead of changing the whole image. Without affecting each other, multiple containers can also share the same image and change data separately. This is useful when a particular OS can be used by several containers [And15; Mer14].

Docker architecture is based on client-server model and includes multiple components which are described as follows. At its core lies Docker daemon that is responsible for managing containers. Its essential capabilities include creating, executing and distributing containers as well as controlling images, networks and volumes. The most common method to communicate with Docker daemon is to use Docker client. All commands specified by the user are transmitted to Docker daemon using REST API. Images are kept in Docker registries, which can be both public or private [Doca]. Figure 2.4 depicts the basic elements of the Docker architecture.

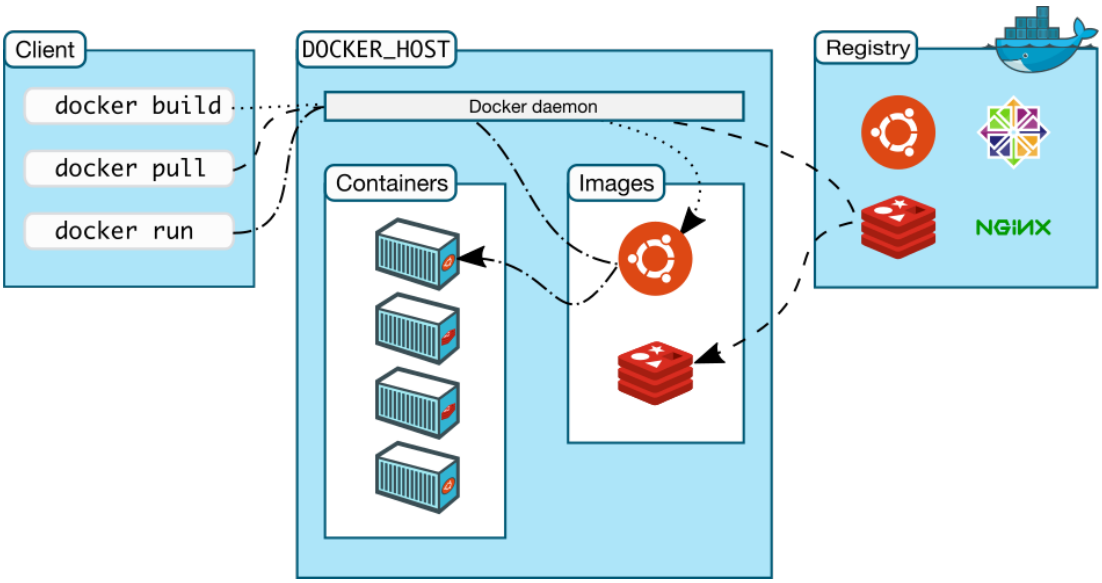

Figure 2.4: Docker architecture overview [Doca]

Docker released its source code in 2013 under open source Docker project [Docb],

which was renamed to Moby in 2017 [Jan18; Pou19]. It is a software collection that contains all resources required to build container-based applications [MG18], including the components depicted in Figure 2.4. Most of the code is written in Go programming language[5].

2.4 Product Model Toolkit (PMT)

PMT helps software developers to integrate critical license information and related metadata generated by license scanners into a unified model. It includes a client that can perform scanning operations using already available scanner tools. These external applications are bundled along with their dependencies into standalone Docker containers. PMT is written in Go programming language. Figure 2.5 shows a high level overview of the PMT client architecture.

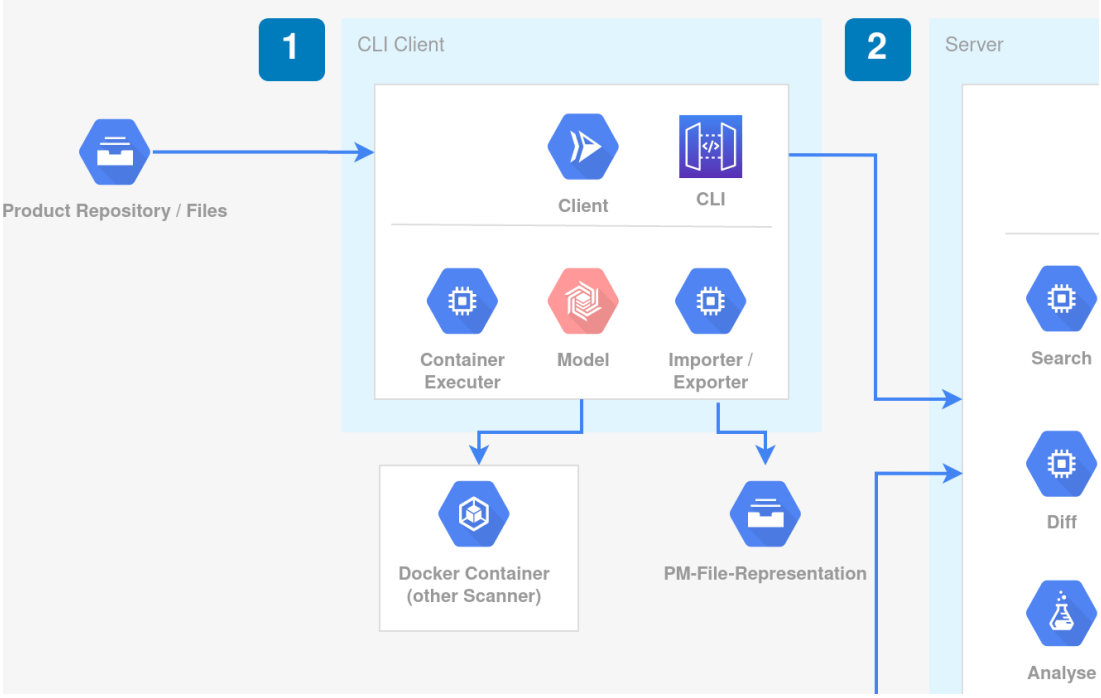

Figure 2.5: PMT client architecture overview [Ope]

Although the client in its original state performs well, its implementation is only a preliminary solution. The implementation drawbacks are explained as follows:

[5]https://golang.org/

- **Hardcoded configuration**

  The information on external scanners and the associated Docker images are stored in the source code, which implies that any modification of that data requires recompilation of code.

- **Docker client is accessed using a shell command**

  To start a container, PMT client runs a shell command to access Docker client installed on the OS, rather than directly executing Docker operations, e.g. through a library. This makes the application unreliable and insecure.

- **Result files are saved locally**

  A host directory is mounted in the container to obtain result files, then, the files are loaded into the program. This has negative impact on performance and, similar to prior issue, makes the application more OS-dependent.

- **No error handling**

  If an application error occurs inside a container, no indication is given to the user, thus making troubleshooting difficult.

- **No loading of remote images**

  Docker images are imported manually into the local Docker registry before the application start.

These implementation drawbacks makes the application error-prone, unstable and also complicates the integration of new license scanner tools.

3 Requirements

This chapter outlines the requirements for design and implementation of the plugin architecture, compiled by analyzing common aspects of plugin architecture models and problem domain.

R1: Programming Language Independent Architecture

The PMT client application uses external, standalone scanner tools for license scanning, whereas each of these tools may be written in different programming languages. It is therefore important to create a plugin architecture that accepts such tools regardless of their implementation details. In this case, interoperability between plugins and the core engine has to be provided to ensure exchange of data, such as program inputs, outputs and results.

R2: Docker-Based Encapsulation of Plugins

Applications commonly have dependencies on other applications or libraries to function properly. On top of that, versioning and configuration of such dependencies play a crucial role, since a mismatch can prevent an application to run successfully. To prevent this, all external scanner tools including their dependencies have to be encapsulated in Docker containers. This ensures correctly constructed plugins function consistently on any Docker-enabled OS or platform.

R3: Core Engine with Minimal Functionality

All three previously reviewed plugin architecture models use a small, simple and robust core engine (also called runtime engine, core system or platform core) that has, as a rule, only minimal functionality to orchestrate the plugins and ensure system stability, whereas any additional functionality is provided by plugins.

R4: Plugin Registry

Without information on available plugins or their location, the core engine is incapable to load them. Such details have to be stored in a registry. It is import-

ant to specify which metadata are relevant for a proper start and execution of plugins. Moreover, the registry is an independent component and must be stored to persistent memory, such as a file.

R5: Data Exchange

To allow data exchange between plugins and the core engine, it is necessary to decide upon an appropriate data exchange approach. Considering the vast number of different standards, protocols and data formats that are available, it is important to examine the upsides and downsides of each of them. Furthermore, the limitations of Docker containerization have to be taken into consideration as well.

R6: Error Handling

Considering that the system may get very complex with increasing number of installed plugins, it is necessary to facilitate the detection of possible errors and be able to track them back to their source. In case of PMT client, errors may occur not only internally, but can also originate from the Docker daemon or the external, standalone scanner tools. Thus, the source of error should also be distinguished.

R7: Configuration

Software configuration provides significant flexibility by simplifying the application adaptation to various user needs or environments. The options can be adjusted without recompiling the code [Say+18]. Considering different configuration approaches, a proper solution has to be found. Besides, configuration is necessary for the core engine as well as the plugin registry.

R8: Versioning

Like any other software unit, plugins may be updated periodically to remain useful through environment transformations. As a result, the core engine must support versioning to be able to prevent execution of outdated or incompatible plugins.

R9: Parallel Execution of Multiple Plugins

To make effective use of multicore systems, the application should be able to run plugins in parallel. Typically, a scanner tool only performs read operation on a specified path, therefore, no interference may occur between the plugins, which facilitates the implementation of parallel execution.

4 Architecture Design

To build an efficient and robust architecture, we first analyzed common techniques of plugin architectures in Section 2.1. Than we considered the challenges and limitations of container technology and how it can fit in a plugin architecture. The architecture design reflects the requirements gathered by the previous analysis processes.

The main aspects of the architecture design are a core engine, plugin registry, and data transfer layer. These main aspects are shown in Table 2.1. In this chapter we describe all aspects of the architecture design in detail and how they work together as a complete system. It is critical to state that a plugin can be in general any program or tool that extends the main application. To simplify explanation, we utilize for our use case license scanners as example tools which serve as plugins for PMT client.

4.1 Core Engine

The core engine represents the heart of the architecture and determines the responsibilities, as well as the functionality, of a minimal core engine and its plugins. We design the core engine to be minimal and avoid unnecessary complexity. A minimal core engine is easier to understand and maintain compared to a complex system.

It is important to state that core engine does not know anything about license scanning at all. It orchestrates all plugins in a way that the PMT client can use several plugins without detailed information about the underlying technology of a scanner or other plugin.

One important aspect of our design compared to traditional plugin architectures is the plugins are independent of any specific programming language or technology. To cover the related requirements R1 and R2 we use Docker containers as abstraction layer. Although the initial state of the program, as described in Section 2.4, satisfies R1 and R2 sufficiently, their fulfillment in the course of development is crucial. R6-R9 indicate additional core engine features, namely error

handling, configuration, versioning and parallel plugin execution. These features should add reliability and efficiency to the application.

We further divide the core engine into multiple distinct parts, each responsible for particular tasks, and propose optimal methods and mechanisms for their realization.

4.1.1 Container Creation and Execution

Plugins are based on Docker images and can include a fully operational license scanner or any other tool that provides usable information needed for the product model. Accordingly, core engine's objective is allowing the user easily integrate plugins into the application and expect a reliable, efficient and secure execution of plugins such as scanner tools.

As described in Section 2.3, Docker daemon is responsible for handling containers' life cycle and Docker client is the interface to communicate with the daemon. According to Docker documentation, Software Development Kits (SDKs) and libraries for communicating with Docker daemon are also available. They provide an effective way "to build and scale Docker apps and solutions quickly and easily" [Doca]. Considering that PMT client and Docker are both written in Go, the most suitable library to communicate with the Docker daemon is the Go client package[1] provided by Docker itself [Docb]. It is worth noting that PMT client uses Docker command line interface (CLI) based on this package, therefore its direct integration into the application yields performance improvement.

To run a Docker container, the application needs an image first. Two options for obtaining Docker images are generally possible: locally and remotely. Docker uses a local file system to store layers of images ready for use by pulling them from a registry or loading them from an archive [Sar20]. A registry can be both local or remote as well as public or private. To pull an image remotely from a registry, authentication may be necessary. For instance, GitHub Packages registry requires an access token even for public images [Git]. In this case, our core engine must allow the user specify their authentication details for obtaining images from both private or public registries.

When integrating a scanner tool into an isolated container, it is necessary to consider the configuration aspects of that container. Generally, a license scanner requires the path to the directory containing the software files and the path to a location for saving the result files. Docker Go client package includes the function `ContainerCreate` which provides numerous possibilities for configuring containers [Docb]. Table 4.1 depicts Docker container configuration settings and options that are highly relevant for our plugin architecture.

[1] https://pkg.go.dev/github.com/docker/docker/client

Settings / options	Description
Image	The basis of a container; identifiable by a string that can also include the hostname of the registry, the name of the image and a tag that specifies its version
Bind mounts	Critical for mounting directories into containers; read-only mounts are also possible
TTY	Necessary for allocating a pseudo-TTY connected to stdin of container; combined with `interactive` option, Docker creates an interactive shell inside the container [Jun20]
Command	Mandatory for starting a new container; following its execution Docker stops the container; it is possible to keep the container up and running by using a shell as the initial command, e.g. `sh` or `bash`

Table 4.1: Configuration settings and options for container creation

The initial state of PMT client compels the user to manually store images locally before using the application. By including the registry hostname in the string that identifies an image, the core engine is able to search for that particular image remotely and pull it from the corresponding registry. This significantly facilitates the integration of new plugins into the application. An example string that serves as image identifier may look like this:

```
registry.example.com/path/to/license_scanner:v1.0
```

Every license scanner requires the path to a codebase as input, therefore it is necessary to mount the specified directory into container. To add an extra layer of security, the core engine mounts the directory as read-only to prevent scanner tools altering any files.

According to Table 4.1 it is possible to keep a container up and running by using the TTY option and a shell as the first command. Although a single command can be sufficient for running a scanner tool, we consider that executing multiple commands inside a container can be necessary, e.g. for troubleshooting purposes, see Section 4.1.2. Therefore, the core engine is able to start a container and consequently execute all commands separately from each other.

Each license scanner generates one or more result files that are saved locally on the system. PMT client in its initial form mounts a host directory for saving the generated files. This implies unnecessary dependency on OS given the result files are sent to PMT server subsequently. On the other hand, we include this as an extra feature that the user can enable or disable as needed.

4.1.2 Error Handling and Logging

Lang and Stewart have long ago recognized that "component-based software needs exception detection and handling mechanisms to satisfy reliability requirements" [LS98]. In a more recent paper, Liu et. al also state that "a system without proper error-handling is likely to crash frequently" [Liu+17].

Our plugin architecture is based on multiple independent components, e.g. standalone Docker containers which encapsulate external scanner tools, external Docker Go client library as well as Docker daemon. As a result, the user may not be able to recognize the root causes of program faults without proper error handling mechanisms.

Osman et. al study in their paper the evolution of exception handling in large software projects and state that "exception handling allows developers to deal with abnormal situations that disrupt the execution flow of a program" [Osm+17]. They classify exceptions into three types. Standard exceptions can occur when using regular functions of a programming language; custom exceptions indicate additional information about domain specific errors that developers may provide; third-party exceptions can arise when calling functions from external libraries or other systems.

In general terms, an exception is an error that arises unexpectedly during the execution of an application [NVN19]. Various programming languages, e.g. Java or C#, use an error handling mechanism that includes a special exception class. The programmer can use the corresponding exception object to signal an error in the program. Go, on the other hand, doesn't comprise such mechanism, but provides a built-in error type and also allows the programmer to return or receive multiple return values.

To help user effectively troubleshoot the program errors and correct them, we use the above categorization of exceptions. We introduce following analogous terms to signal errors depending on their origin:

- **Standard errors**: originate from common Go functions within PMT itself, e.g. file opening or saving or reading values of environment variables.
- **Core engine errors**: arise when core engine is not able to function properly due to misconfiguration or incorrect user input, e.g. invalid authentication credentials.
- **External library errors**: occur when using functions from external libraries; core engine indicates the name of external library when signaling the error, e.g. Docker client error.

Every new function that core engine comprises must therefore signal errors by returning an error value where applicable. When using standard Go functions or

calling functions from external libraries, the core engine must handle their error values accordingly. It is critical to state how the core engine communicates the errors to the user. We consider that printing messages that include the origin and a brief description of the error to the console is sufficient. An example message that signals an error may look like this:

```
Core engine signals a Docker client error: the specified image
   does not exist remotely or locally, please check the image
   name for typos
```

Previously mentioned error handling approach does not apply to license scanners that run as standalone programs inside containers. We consider using logging for this purpose. Reynders mentions that "application logging plays a crucial role in tracking and identifying issues that may surface as well as in providing useful insights on the workflow processes of solutions" [Rey18].

It is common that console applications may already provide useful information by writing to output streams [Kir20]. When a program starts, a Linux or Unix-like OS opens three data streams: stdin, stdout and stderr [Bot20]. As a rule, applications communicate any errors by writing to stderr. We consider capturing both stdout and stderr when executing a command and combine and write that data into a log file. Subsequently, the user may examine the log file when a problem occurs. The core engine creates the log file in a temporary directory given by the OS and displays its path on the terminal. The contents of an example log file may look like this:

```
stdout of command mv oldName newName
[empty]
stderr of command mv oldName newName
mv: cannot stat 'oldName': No such file or directory
```

We use Unix/Linux command `mv`[2] to rename a file or folder named `oldName` to `newName` inside a container. Following its execution, the command doesn't write anything to stdout, hence it's empty, but signals an error by printing to stderr and informs that no such file or folder named `oldName` exists. Analogously, the user can examine the outputs of every command executed inside containers.

4.1.3 Configuration of Core Engine Settings

This section discusses configuration of core engine settings. For plugin registry configuration see Section 4.2.1. Configuration helps user adapt an application to their needs based on settings or options the application itself provides. We consider including following settings in the core engine configuration:

[2]https://www.gnu.org/software/coreutils/manual/html_node/mv-invocation.html#mv-invocation

- **Authentication credentials**

  As mentioned in Section 4.1.1, an access token is necessary when using GitHub Packages registry to pull images remotely, even for publicly available images. The user must therefore provide their authentication credentials in form of a username and a string that includes the password or token depending on registry conditions.

- **Setting for saving result files locally**

  This configuration setting comprises two elements. The first element is a flag that indicates that user wants to receive result files locally in addition to sending them to PMT server. When user enables this option, they may consequently provide a path for saving the result files. If no path is specified, core engine uses temporary directory given by OS.

- **Path to directory for saving log files**

  When executing commands inside containers, core engine logs all messages written to output streams and saves them in a log file for later examination. By default, core engine saves that data in temporary directory given by OS. If user provides the path to a directory as attribute value of this configuration setting, PMT client must use that directory for storing the log files.

It is critical to discuss which representation is optimal for the configuration file used by the core engine to store the settings. Rasool et. al mentions following file formats for storing data: HOCON, JSON, OpenDDL, XML and YAML. [Ras+19]. We consider JSON is the optimal choice for this purpose, as it "makes it easy for human to read and write, and for computers to generate and parse" [PCX11]. Although it has several downsides such as "lack of namespace support, lack of input validation and extensibility drawbacks" [Nur+09], they do not impair its applicability in this case.

4.1.4 Versioning of Core Engine and Plugins

Stuckenholz [Stu05] studied component evolution in component-based software development and the related compatibility conflicts that may arise due to modifications to components. He states that "versioning mechanisms are typically used to distinguish evolving software artifacts over time" and also that "these mechanisms play an important role in component based software development" [Stu05].

A version indicates a specific state of an application or component. Thus, if they change so does their associated version number. In our case, we have to consider version handling in three different parts of the architecture: core engine,

plugins and plugin registry. In this section we discuss versioning of core engine and plugins, for versioning of plugin registry see Section 4.2.1.

We consider that core engine may be subject to continuous improvement in the future. Ghezzi states that "requirements for a given application are only vaguely known when a new development starts" and "they become progressively better known as development proceeds, and in particular, as feedback information starts flowing from customers and from operation" [Ghe17]. As a result, each specific state of the core engine must be identifiable by its associated version number. The reason for doing this is to inform the user of potential changes to the core engine which they must take into consideration when creating or modifying their plugins. For instance, a particular version of the core engine may not have any constraints on which shell is used inside container, although a future version may require a particular shell, e.g. `bash`, to function properly. We consider the indication of such changes in a corresponding change log file is sufficient.

Each plugin is also identifiable by a version indicator which makes it noticeable if a new version of a plugin is used. On the other hand, the specific execution of an older version should also be possible. A plugin must therefore include the version of the tool which it encapsulates and the version of the core engine it corresponds to. Core engine must ensure the plugin can be loaded based on the core engine version the plugin specifies.

4.1.5 Parallel Plugin Execution

One single plugin is not able to cover all aspects of a product. The execution of multiple tools as plugins is therefore necessary. Considering that core engine mounts the directory containing the target code repository as read-only, no interference between plugins may occur. We therefore consider the possibility to run the plugins in parallel to improve performance of the application.

4.2 Plugin Registry

Plugin registry has the responsibility to store and handle information about installed plugins. Although many plugin architecture models describe the plugin registry as an integral part of the core engine, as mentioned in Section 2.1, R2 explicitly indicates that plugin registry must be an independent module.

The representation of our plugin registry is in essence a file that includes all relevant information about prepackaged plugins. Then, plugin registry handler imports that file during application startup and consequently provides various associated functions that core engine can use. It is worth mentioning that in general plugin registries can also be a public or company-wide service which

can then be explored by the core engine. In the future, provisioning of the plugin registry as a public service, filled with community plugins may be therefore possible.

4.2.1 Representation of Plugin Registry

To store information about ready-to-use plugins a proper representation of plugin registry is necessary. First, it is critical to consider which configuration file formats are suitable for this purpose and, second, which attributes are most appropriate.

As mentioned in Section 4.1.3, different file formats for configuration files are available. We consider that two file formats, namely YAML and JSON are well suited for our case. Both file formats are human-readable which plays a fundamental role considering that user creates configuration files manually. At the same time, both file formats offer advantages and disadvantages. For instance, YAML allows comments to be added by the user, whereas JSON doesn't provide this feature [GDB15]. In terms of serialization and deserialization, JSON outperforms YAML [SM12]. Finally, we prefer adding support for both file formats and leave it up to user to choose the convenient file format.

It is important to discuss which metadata about plugins is necessary to include in plugin registry. By analyzing the main functions of the core engine, we consider following plugin metadata are significant for our Docker-based plugin architecture:

- **Name and version of the tool**: are necessary to identify the plugin.
- **Core engine version**: helps the core engine determine whether the plugin is compatible with its current version or not; the user specifies the actual version at the time of plugin creation.
- **Image**: identifies the corresponding Docker image that core engine must load for execution of Docker container; it may contain the registry hostname as described in Section 4.1.1.
- **Shell**: indicates the shell necessary to start the tool.
- **Command**: is the actual command including the associated options and parameters to execute the tool.
- **Result files list**: specifies the result files that core engine must receive following the execution of the plugin.

The representation of the plugin registry also needs a version number to determine the attributes of this representation. Based on the version number, core engine can then determine if it can load the plugin registry specified as well as execute

the plugins included. An example for such versioning is the version number in Docker Compose files [Doca] or XML files [HKS02].

Plugin registry must also specify a default plugin which is then used by the core engine in case user doesn't indicate a plugin when executing the PMT client. An example YAML configuration file that represents a plugin registry may look like this:

```
version: R1.0
default_plugin: 1
plugins:
- name: Example Tool
  version: 3.3.3
  core_version: C1.0
  docker_image: registry.example.com/path/to/example_tool:v3.3.3
  shell: /bin/bash
  cmd: example_tool -i /input/ -o /result/result.json
  results:
  - result.json
```

4.2.2 Plugin Registry Handler

This component is responsible for providing the core engine with all data and corresponding functions necessary to orchestrate the plugins. It executes some functions automatically at startup and other functions on demand:

- Check compatibility with specified configuration file (at startup)
- Import plugin metadata from YAML or JSON configuration file (at startup)
- Signal empty registry (at startup)
- Search for a plugin by its name
- Provide a default plugin

4.3 Communication between Core Engine and Plugins

As described in Section 2.1, most plugin architecture models use same programming language for core engine and plugins. As a result, a method from the core engine can directly call another method from a plugin or vice versa. Moreover, different standardized technologies exist that help developers create applications based on plugin architecture, e.g. OSGi, but are limited to a particular programming language [Ric+11]. Considering that our plugin architecture must not be subject to such limitation, a universal approach for data exchange between core engine and plugins is necessary.

According to Docker documentation, communication between core engine and plugins is possible in two different ways, namely by using network communication or a special Docker command that can transfer files between containers and local file system [Doca]. To provide more flexibility we consider utilizing both methods and provide a configuration setting that can be adjusted by the user according to their needs.

4.3.1 Network Communication

Paraiso et. al mention following binding technologies for plugin systems: HTTP, JGroups, JMS, JNA, JSON-RPC, Java RMI, OSGi, REST, SLP, SOAP and UPnP [Par+12]. Many of these technologies rely on a specific programming language or file format, therefore we disqualify them and compare the remaining ones, namely HTTP, REST, SLP and UPnP and consider that REST fits best in our Docker-based plugin architecture as it "attempts to minimize latency and network communication while at the same time maximizing the independence and scalability of component implementations" [FT00]. REST is a software architectural style for distributed systems which is also used by Docker daemon [Doca] for its API as described in Section 2.3.

In our particular case, core engine must be able to communicate with plugins in order to receive the generated result files. Although no further interaction is currently necessary, this may change in the future. Therefore, we consider building a simple REST API that plugins can use for sending result files as well as for other prospective purposes.

We consider LabStack's Echo web framework [Lab] written in Go is the optimal choice for creating our REST API, which is also used by PMT server and therefore doesn't introduce a new dependency. Our REST API must include a single HTTP method at the moment, namely POST; user may later add various HTTP methods as necessary. The server must then check the result files enclosed in plugin's request if they correspond with specified result files in the plugin registry configuration file, see Section 4.2.1. Following the receiving of result files, core engine must also stop the running container and send the data to PMT server and, if requested, save the data locally. To provide a reliable communication, we must build the REST API in a blocking fashion, meaning that the application waits until data transmission is completed.

For plugin side, a tool is necessary to send the results back to the core engine. We choose the widely used `curl` command line tool for this purpose, which can make "transfers for resources specified as URLs using Internet protocols" [Ste18].

4.3.2 File Transfer between Container and Local Machine

As a second way to transfer files between the core engine and a plugin we can utilize the `CopyFromContainer` function of Docker Go client package [Docb]. Following the execution of the tool, core engine can therefore use this function to get the result files specified in the plugin registry. In case core engine cannot find the specified result files, it must signal a corresponding error and inform the user of a possible misconfiguration.

4.4 Plugin Compatibility Verification

Most plugin architectures, as described in Section 2.1, provide interfaces which plugins must implement. An interface ensures that the corresponding implementation strictly conforms to the defined contract, e.g. by implementing the specified methods. This makes it easier for core engine to verify if plugins are compatible with the program and it also helps to identify the cause of a fault execution of a plugin. Considering that our plugins are basically external tools that run inside Docker containers isolated from the core engine, an appropriate compatibility verification is therefore beneficial.

We consider checking plugin compatibility at two stages. First, implicitly when creating the container and, second, explicitly when it is up and running. As described in Section 4.1.1, Docker first creates the container using the specified image and configuration settings. At this point, Docker signals an error if container creation is not possible, e.g. due to a corrupted image or inability to mount directory into container.

At the next stage, namely when container is up and running, a compatibility check is necessary to ensure core engine is able to execute commands inside container. A basic command such as `echo test` can confirm that core engine can execute commands and receive data printed to output streams stdout and stderr as mentioned in Section 4.1.2. Moreover, in Section 4.3.1 we take advantage of `curl` command line tool to send result files to core engine. Therefore, the image must include this tool for plugin to function properly. We consider that execution of a related command to test if the dependency within the plugin is present without causing side effects is an appropriate solution. In case of `curl` we call the command with the `-V` flag[3] to return its version number. Depending on prerequisites or dependencies required to run the plugins, a function that checks all preconditions is therefore required.

[3]https://curl.se/docs/manpage.html

4.5 System Integration

As described previously, our plugin architecture comprises multiple components each responsible for different aspects. At this juncture, we need to put these components together and show how they function as a complete system.

Figure 4.1 depicts the fundamental building blocks of our plugin architecture and particularly how Docker fits in. Core engine uses Docker Go client to communicate with Docker daemon which is in control of containers' life cycle. Plugins, in turn, consist of containers that include the external tools and all necessary dependencies to ensure that they can properly fulfill their role in the application. Two types of registry are present. Plugin registry includes all relevant metadata about plugins. Container registry, on the other hand, is a remote repository where images are located. Plugins can use REST API to send the generated result files, whereas the server then saves the data into filestore which can be accessed by the core engine. As a result, core engine can stop containers as quickly as possible and not depend on them when sending result files to PMT server.

The fundamental interactions between components are shown in Figure 4.2. It is worth mentioning that all messages in this sequence diagram are synchronous which means that a response is necessary for the application to continue. This is related to error handling mechanisms described in Section 4.1.2. Depending on configuration of core engine, alternative scenarios and optional steps are possible.

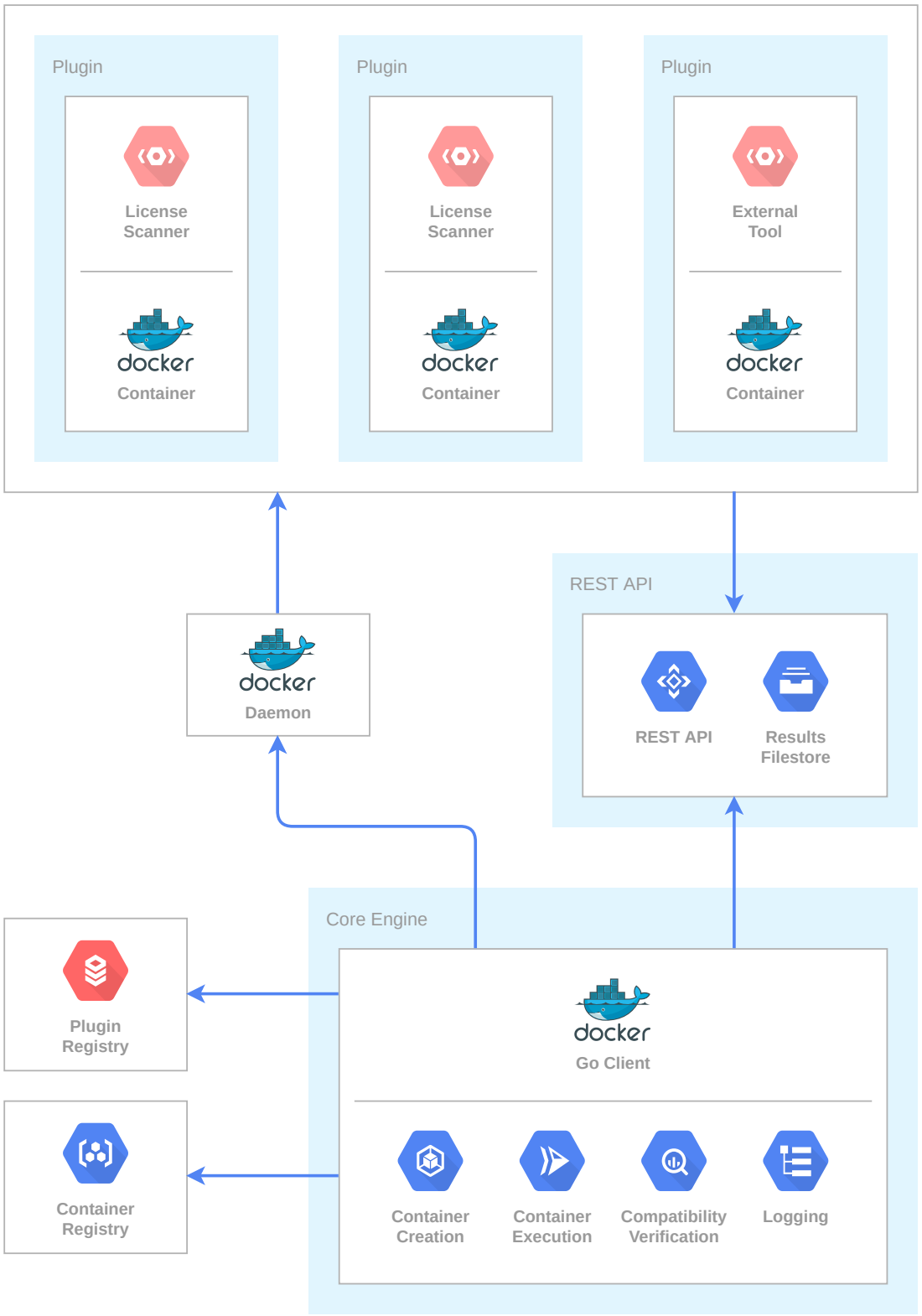

Figure 4.1: Fundamental building blocks of our Docker-based plugin architecture

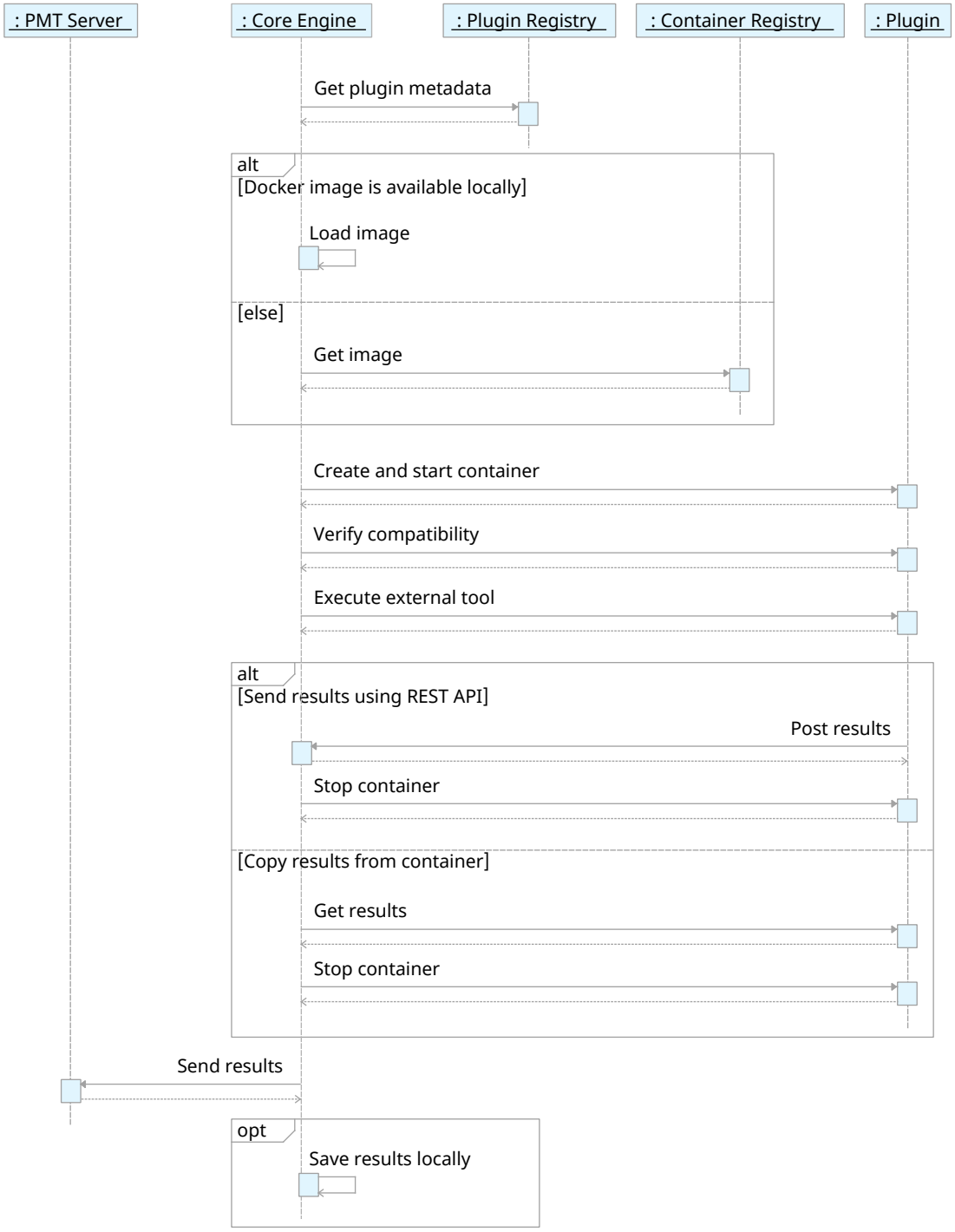

Figure 4.2: High level sequence diagram showing a complete sequence of a plugin execution

5 Implementation

To prove the feasibility of our Docker-based plugin architecture designed in Chapter 4, we implemented our architecture in PMT client. This chapter describes the different phases of the implementation including the challenges we encountered as well as the corresponding solutions we applied to meet the aforementioned design criteria and specified requirements. Before we started implementing the architecture in PMT client, we forked the code repository to be able to adjust the application without interfering with the rest of application development. We then generated pull requests to submit our modifications.

It is worth mentioning that the implementation of PMT application is based on Domain-Driven Design (DDD) which has the aim of creating "better software by focusing on a model of the domain rather than the technology" [Eva04]. In addition, DDD approach suggests using modules that "serve as named containers[1] for domain object classes that are highly cohesive with one another" [Ver13]. As a result, PMT application consists of multiple packages[2] that meet low coupling, high cohesion principle for good modularization and are each responsible for a particular concept. To conform to the current application's structure, we therefore considered creating a new package that includes all concerns regarding plugins. Hereinafter we refer to the implementation of our Docker-based plugin architecture as Docker-based plugin system.

We modeled our problem domain using UML. Figure 5.1 depicts a class diagram that shows the structure of our Docker-based plugin system including the attributes, methods and relationships among classes. It is important to state that we removed retrospectively some helper methods and other details such as method parameter data to make the class diagram easier to understand. Some class names are abbreviated, e.g. core for core engine or agent for plugin agent.

In the following sections we describe in detail each class included in the package individually and explain how their functions communicate with each other to ensure a reliable and efficient execution of plugins.

[1] Do not confuse with Docker containers

[2] Evans also refers to modules as packages [Eva04].

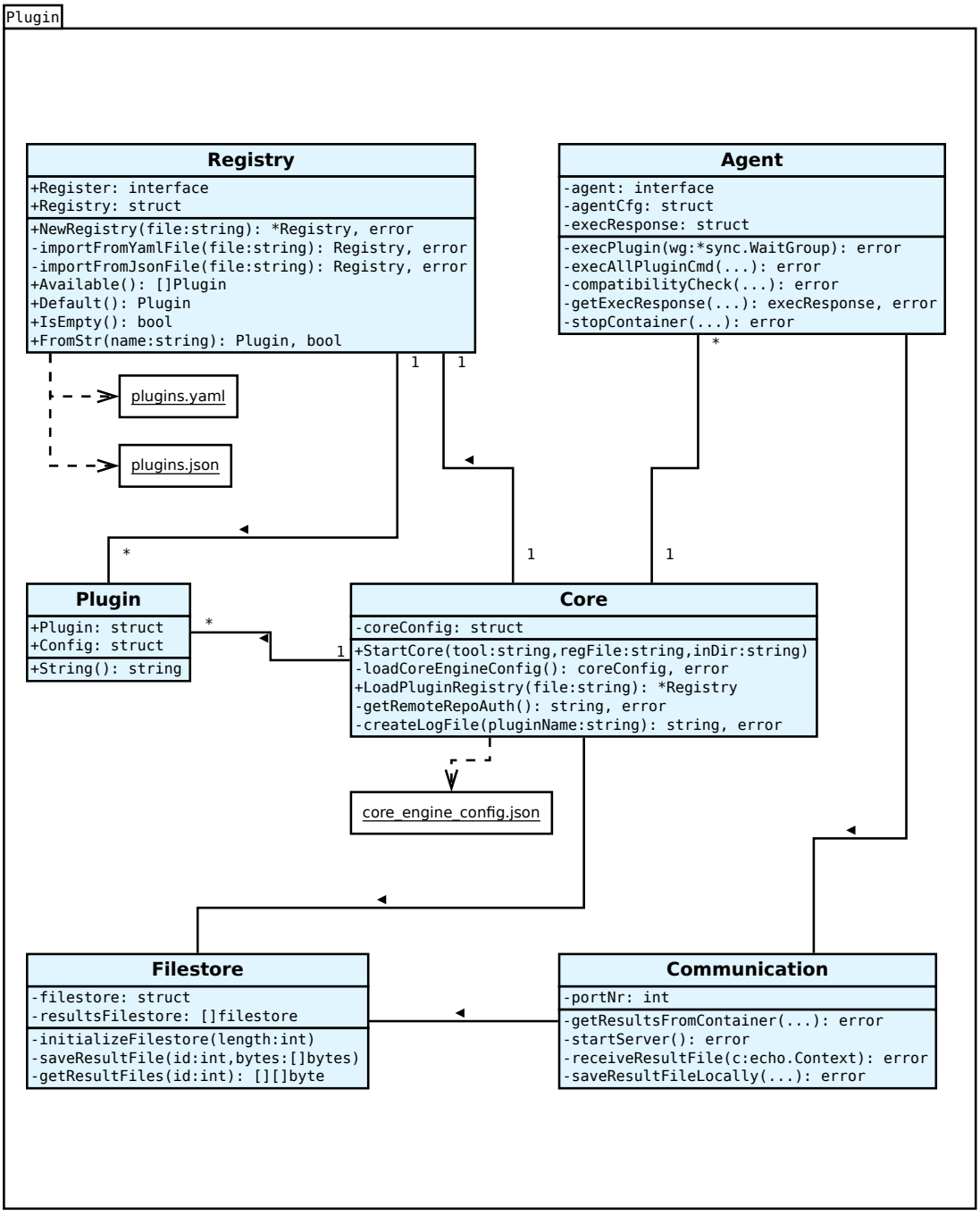

Figure 5.1: Class diagram representation of the plugin package

5.1 Core Engine

Core engine is the central point of our plugin system which is responsible for initializing all other components necessary to handle the plugins. It is critical to mention that we initially designed the core engine to be in charge of directly managing the plugins, as shown in Figure 4.2. In order to carry out the parallel execution of plugins, see R9, we bundled the functions responsible for plugin orchestration in a separate plugin agent class as described in Section 5.3.

How the plugin system exactly operates depends on core engine configuration. We included a JSON configuration file that user can adjust according to their needs. A segment of this file including the attributes and their default values is shown below:

```
"RestApi": true,
"RemoteRepoUser": "",
"RemoteRepoPass": "",
"SaveResultsLocally": true,
"PathDirResults": "",
"PathDirLogs": ""
```

The user can change the first attribute to indicate if plugins must use REST API to send the generated result files. If user disables this feature, plugin system copies the generated result files directly from container, see Section 5.5. Considering that authentication may be necessary even for public images, the user must therefore indicate their authentication credentials. We provided the possibility to include them in the configuration file or by setting the environment variables `REMOTEREPO_USER` and `REMOTEREPO_PASS` which adds an extra layer of security. The function `getRemoteRepoAuth` is responsible for providing a corresponding authentication string that Docker subsequently uses to pull a container from a container registry. The user can also specify the path to a location for saving result files or disable this feature completely. If no path is indicated, core engine creates a folder in the temporary directory given by the OS; the same applies to log files.

After reading the options and parameters specified by the user, PMT client calls a single function named `CoreStart` to start the plugin system. This also holds to the principle of low coupling. `CoreStart` function performs following tasks to make the whole plugin system operable:

- **Load core engine configuration**

  A special `loadCoreEngineConfig` helper function decodes the JSON configuration file and reads the data inside. It also automatically creates new directories for result files and log files when necessary.

- **Load plugin registry**

  Core engine initializes plugin registry by creating an instance of `Register` interface that provides all plugin registry operations.

- **Initialize filestore**

  Depending on how many plugins user wants to execute, core engine initializes filestore with corresponding amount of pointers to byte slices. This is necessary for saving result files temporarily before sending them to PMT server and therefore preventing the application from sending incomplete results.

- **Start selected plugins in parallel**

  At this point core engine uses `WaitGroup`[3] type provided by Go itself to start the plugins in parallel. Core engine runs the plugins in separate threads and waits until all plugins finish their operation. To avoid conflicts that can arise when accessing same resources, plugins system mounts the directory containing the codebase into container as read-only and uses separate memory pointers for saving result files. Moreover, each plugin has its own log file.

Finally, core engine also includes three functions that can be used by all plugins. One is `getRemoteRepoAuth` that generates the authentication string required to pull containers from container registries and two further functions for creating and writing to log files.

5.2 Plugin Registry

Plugin registry provides the core engine with all information necessary for plugin orchestration. It includes a `Register` interface that core engine instantiates using the path to configuration file specified by the user. `Registry` type, on the other hand, represents a plugin registry.

`NewRegistry` is the function that initializes the plugin registry by importing the configuration file. Depending on the file format, it uses other helper functions to import data from YAML or JSON configuration files and generates the corresponding variable of type `Registry` that includes that data. If the import is successful, this function then checks whether its version is compatible.

`Register` interface provides the following functions

- `Available`: provides all available plugins by returning a list of `Plugin` variables

[3] https://golang.org/pkg/sync/

- `Default`: returns the default plugin as specified by the user in the configuration file
- `IsEmpty`: indicates whether plugin registry is empty or not by returning the corresponding boolean value
- `FromStr`: searches for a plugin by its name and indicates with a boolean value if plugin is not in the plugin registry

Except for helper methods, we made all functions in the plugin registry class public. The reason for doing this is to allow the user examine which plugins are available without starting the core engine. PMT client allows the user to use `-l` flag to print a complete list of available plugins on the terminal.

5.3 Plugin Agent, Communication and Filestore

Plugin agent class is responsible for the complete life cycle of a plugin. It provides a function named `execPlugin` that core engine calls to start the plugin execution in a separate thread. This function then uses multiple helper methods to manage the associated Docker container. Plugin agent class is highly cohesive with other two classes, namely communication and filestore. In this chapter we describe how the functions in these classes interoperate to ensure a reliable and efficient plugin execution, whereas Figure 5.2 depicts an activity diagram which provides a visual overview of the complete procedure.

First, plugin agent prepares the container. It checks whether the image exists in the specified container repository and pulls it if found. If the image is not available remotely or the image identifier does not include the repository hostname, it searches Docker's local file system for that image. It then uses `ContainerCreate` function provided by Docker Go client to configure and create the container. At this stage plugin agent mounts directory containing the target code repository as read-only into container and also uses TTY option and the specified shell as the first command to keep the container up and running when starting it.

After starting the container, plugin agent checks the configuration of core engine if plugins must use REST API to send the generated result files or for other related purposes. If this is the case, plugin engine starts the server by calling `startServer` function in communication class. Consequently, this function opens a free TCP/IP port to listen on and uses LabStack's Echo web framework [Lab] as mentioned in Section 4.3.1 for handling POST method requests. It is worth mentioning that although this function runs in a separate thread for each plugin, the server starts only once and all plugins use the same TCP/IP port to send POST method requests. If REST API feature is disabled, then plugin engine doesn't start the server and proceeds to the next step.

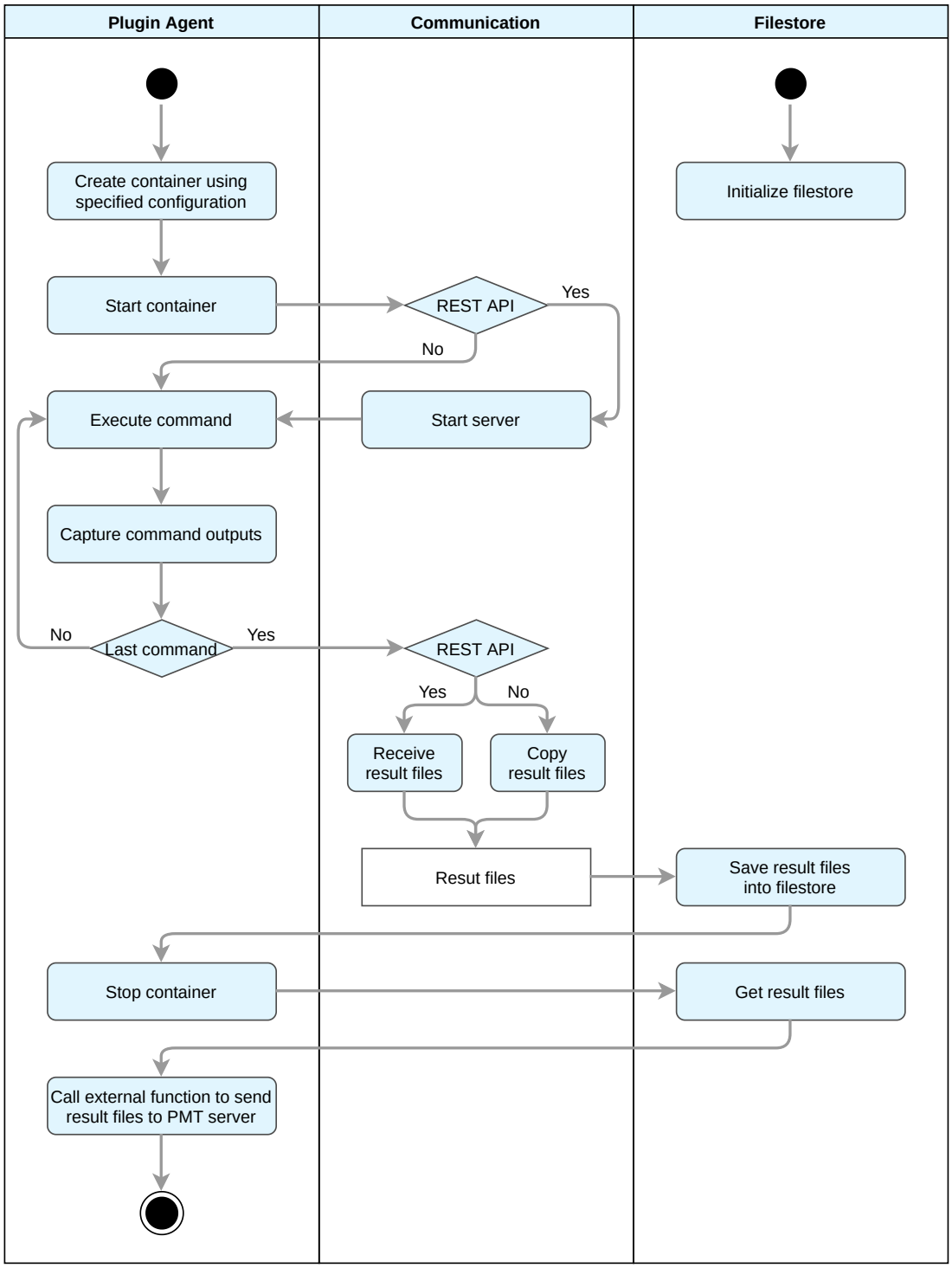

Figure 5.2: Activity diagram showing the complete behavior of plugin agent running in a separate thread

At this point container is up and running and plugin agent can start executing all necessary commands inside. However, to ensure it can successfully do this, plugin agent performs a compatibility verification as described in Section 4.4. It first executes `echo test` to verify whether it is able to capture data printed to stdout and stderr, see Section 4.1.2. Moreover, if REST API feature is enabled, plugin engine then checks if `curl` command line tool is available inside container. After checking the plugin compatibility, plugin engine creates a special folder where plugin can save the generated result files. Finally, it executes the external tool.

Plugin engine must ensure at this stage it collects all result files as specified in the plugin registry. If REST API feature is enabled, it executes `curl` command for each result file inside container as shown below:

```
for i in /result/*; do curl -F name=[name] -F id=[id] -F
   result=@$i http://127.0.0.1:[port]/save; done
```

In case REST API feature is disabled, plugin agent receives the result files by running `getResultsFromContainer` function in communication class. It is important to note that no additional verification is necessary to check whether filestore contains all result files as specified in the plugin registry because either procedure implicitly signals an error in case a result file is missing.

Finally, plugin agent calls an external function in scanning package which sends the result files to PMT server. Consequently, PMT client terminates.

5.4 Status and Error Messages

To facilitate the tracking of application progress in the course of its operation, our plugin system displays status and possible error messages on the terminal. During the implementation we also revised the error handling approach designed in Section 4.1.2 so that status and error messages conform to the same structure. Every message therefore includes the name of the class it originates from as well as the name of the associated external tool if applicable and the actual message. An example status message that originates from plugin agent is shown below:

```
[Plugin agent] [Licensee] Executing following command in
   container: mkdir /result
```

The same structure applies to error messages, whereas they include a short description of the error as well as the original error message if it originates from external library functions. As a result, error messages include more details as initially discussed in Section 4.1.2. An example error message is shown below:

```
[Plugin agent] [Composer] Unable to pull image from container
   registry, got following error: [...]
```

Regarding the log files that include stdout and stderr data of each command executed inside containers, we chose to ignore empty outputs completely as opposed to explicitly disclosing them in log files.

5.5 Tests

Jorgensen mentions two primary reasons for testing software which "are to make a judgment about quality or acceptability and to discover problems" [Jor14]; he also describes different approaches to test software. We used two approaches to test our plugin system at two different levels, namely unit tests and system tests.

For each function that doesn't depend on any external library we built unit tests that comprise various test cases. As part of the PMT application's CI pipeline, GitHub Actions then runs the corresponding workflows to execute these tests [Git; Ope]. This ensured the sustained operability of our functions during the implementation of our plugin architecture. In total, 15 test cases that cover 9 functions in our package are included.

Although we could have created integration tests to test whether the components of our plugin system properly interact with each other, which requires mocking of external dependencies, in our case Docker Go client and Echo web framework libraries, we chose to test this at a higher level, namely with system tests.

Considering that plugins created for our plugin system can generally be very distinct from each other as they are equivalent to lightweight VMs that include different applications, dependencies and even OSs, it is practically impossible cover all test cases. We therefore performed system tests manually by building multiple plugins which have various external tools that can run different operations on a codebase.

5.6 README

Prana et al. say that "README files play an essential role in shaping a developer's first impression of a software repository and in documenting the software project that the repository hosts". [Pra+19]. In our case, it is critical to provide documentation that properly describes how the user can create a plugin. We included the following information in our README file:

- A brief description of our plugin system
- A figure that depicts the fundamental building blocks
- A step-by-step guide on how to create a plugin
- Further prerequisites, e.g. authentication credentials

6 Evaluation

At this juncture we completed the design and implementation of our Docker-based plugin architecture and in this chapter we evaluate the architecture and its implementation by reviewing whether the specified requirements in Chapter 3 are met. We assess the fulfillment of each requirement individually.

R1: Programming Language Independent Architecture (✓)

Our plugin architecture accepts plugins which consist of Docker containers that encapsulate external tools. The implementation details of these tools, including their programming language, execution environment or used dependencies, are therefore abstracted as executable Docker containers. The successful operation of external tools inside containers is the only premise for ensuring interoperability between plugins and our plugin system. Hence, the tools inside a plugin can be based on any programming language or technology and this requirement is therefore satisfied.

R2: Docker-Based Encapsulation of Plugins (✓)

As mentioned in Section 2.3, various containerization technologies exist that can run applications including their dependencies inside isolated containers. This requirement explicitly indicates that the architecture must support Docker-based encapsulation of plugins. We designed the architecture to support Docker containers from the very beginning so it fully complies with this requirement which, in essence, also represents the goal of this master thesis.

R3: Core Engine with Minimal Functionality (✓)

Core engine is responsible for a reliable, efficient and secure execution of plugins and this can easily lead to a very complex system. Therefore, this requirement makes a good point in regard to complexity of the architecture. Core engine must include only the necessary functionality to orchestrate the plugins and nothing more than that. Except for one aspect which we later eliminated in the imple-

mentation phase, see review of R8, we cannot find any further element of our plugin architecture that we can omit and simultaneously ensure a failsafe operation. All building blocks of our plugin architecture as shown in Figure 4.1 or functions included in our plugin package as outlined in Figure 5.1 are certainly necessary to handle the plugins, therefore our core engine has minimal and at the same time sufficient functionality to ensure a reliable orchestration of multiple plugins and this requirement therefore is met.

R4: Plugin Registry (✓)

This requirement is very specific regarding the details of the plugin registry: it contains information about plugins, specifies which metadata are relevant and is also an independent component. We strictly followed these criteria and designed and implemented the plugin registry as a configuration file representation and added support for both YAML and JSON file formats. The plugin registry handler provides the core engine with all data and corresponding functions necessary for plugin orchestration. We further suggested that public, community maintained registries may be possible in the future.

R5: Data Exchange (✓)

This requirement signifies a delicate aspect of the architecture. Considering the execution of external tools inside isolated containers, a suitable data exchange approach is necessary to ensure these tools can send the generated result files to the application. Fortunately, Docker Go client provides the possibility to copy files from container's file system. But this functionality doesn't allow a reliable communication between plugins and core engine. In this case, network communication is the only way to ensure platform independence. We compared different standards and protocols and ultimately built a REST API that plugins can use to easily communicate with the core engine, e.g. to provide the generated result files or signal the status of an operation. Our plugin architecture supports both approaches and therefore entirely satisfies this requirement.

R6: Error Handling (✓)

With an increasing number of plugins which collectively perform various tasks, the root cause of a possible disruption of plugin system operation can be very difficult to determine without proper error handling mechanisms. We took this aspect very seriously into consideration when designing and implementing our plugin architecture and provided two error handling approaches. The first approach addresses errors that occur during plugin system operation internally, it namely communicates to user all related details about their origin. The second

approach involves logging the outputs of every command run inside Docker containers which facilitates the detection of plugin-related errors. These approaches fully comply with this requirement.

R7: Configuration (✓)

Our Docker-based plugin architecture allows the user to configure the application according to various settings and options we provided, e.g. enable or disable REST API, set authentication credentials or specify path to directory for saving result files. We also analyzed multiple data representation formats to find the suitable ones and added support for both YAML and JSON file formats as representation of the plugin registry. The user can therefore use the one they prefer most. This requirement is thus met.

R8: Versioning (✗)

This requirement indicates that the architecture must support versioning in order to determine whether plugins are compatible or not. In the design phase we discussed version handling in three different parts of our architecture. Later in implementation phase we omitted versioning of core engine as we considered that this aspect is already covered when plugin agent performs plugin compatibility verification. Nevertheless, we included versioning of plugins as this is necessary to distinguish between different versions of containers when pulling them from container registries and we also included versioning of plugin registry to prevent loading of an unsupported representation in regard to its attributes. We consider that our plugin system accomplishes the goal of this requirement but not its specification as it doesn't explicitly verify the plugin compatibility based on its version.

R9: Parallel Execution of Multiple Plugins (✓)

The fundamental idea behind the plugin architecture pattern is to support a relatively high number of plugins that extend the functionality of application. Our Docker-based plugin architecture is no exception as multiple plugins are necessary to cover all aspects of a product. Parallel execution of plugins allows effective utilization of multicore systems and saves time. Considering that external tools perform only read-only operations on the codebase, this significantly facilitated the implementation of this feature in our plugin system and improved its performance. This requirement is therefore fully satisfied.

7 Conclusion

At the very beginning of this master thesis we raised questions regarding the characteristics of the plugin architecture pattern and how can we combine this pattern with Docker containers to build an efficient plugin system that facilitates the integration of individual tools into the PMT client application. In the course of our development we considered all these questions and provided appropriate explanations and solutions. We were able to create an architecture model that serves as a technology independent approach and subsequently reveal its capabilities and limitations by implementing it in the PMT client application.

The complexity of our Docker-based plugin architecture plays a decisive role in assessing its suitability for a particular system. In Section 2.2 we already identified the difficulties in implementing the plugin architecture and mentioned that it requires creation of multiple components such as a core engine, plugin registry or communication mechanisms. This can make the implementation very challenging. Therefore, software vendors should establish a trade-off with respect to the number of plugins and the overall implementation effort. Our Docker-based plugin architecture promises a flexible and efficient approach towards the orchestration of a relatively high number of plugins.

As future work we consider that provisioning of plugin registries as a a public, company-wide or even community maintained service can further facilitate the management of open source dependencies in products. Instead of creating and integrating each plugin separately, software vendors could simply use services filled with already created plugins and directly run all associated operations to extract significant information from their applications.

7.1 Acknowledgment

I would like to thank my supervisor Andreas Bauer whose continuous support, guidance and expertise in the field has been invaluable throughout the whole thesis process.

References

[And15] Charles Anderson. 'Docker [software engineering]'. In: *Ieee Software* 32.3 (2015), pp. 102–c3.

[Bau+20] Andreas Bauer et al. 'Challenges of tracking and documenting open source dependencies in products: A case study'. In: *IFIP International Conference on Open Source Systems*. Springer. 2020, pp. 25–35.

[Bir05] Dorian Birsan. 'On plug-ins and extensible architectures'. In: *Queue* 3.2 (2005), pp. 40–46.

[Bot20] David Both. 'Data Streams'. In: *Using and Administering Linux: Volume 1*. Springer, 2020, pp. 239–271.

[Cox19] Russ Cox. 'Surviving software dependencies'. In: *Communications of the ACM* 62.9 (2019), pp. 36–43.

[Doca] Docker Inc. *Docker documentation*. URL: `https://docs.docker.com/` (visited on 2021-03-24).

[Docb] Docker Inc. *Moby project [source code]*. GitHub repository, commit: dea989e. License: Apache 2.0. URL: `https://github.com/moby/moby`.

[Eva04] Eric J Evans. *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional, 2004.

[FT00] Roy Thomas Fielding and Richard N. Taylor. 'Architectural Styles and the Design of Network-Based Software Architectures'. AAI9980887. Doctoral dissertation. 2000. ISBN: 0599871180.

[GDB15] P Greenfield, M Droettboom and E Bray. 'ASDF: A new data format for astronomy'. In: *Astronomy and Computing* 12 (2015), pp. 240–251.

[Ghe17] Carlo Ghezzi. 'Of software and change'. In: *Journal of Software: Evolution and Process* 29.9 (2017), e1888.

[Git] GitHub Inc. *GitHub documentation*. URL: `https://docs.github.com/en` (visited on 2021-03-24).

[HKS02] Matthias Hansch, Stefan Kuhlins and Martin Schader. 'XML-Schema'. In: *Informatik-Spektrum* 25.5 (2002), pp. 363–366.

[Jan18] Kinnary Jangla. *Accelerating Development Velocity Using Docker: Docker Across Microservices*. Apress, 2018.

[Jor14] Paul C Jorgensen. *Software testing: a craftsman's approach*. CRC press, 2014.

[Jun20] Josh Juneau. 'Deploying to Containers'. In: *Jakarta EE Recipes*. Springer, 2020, pp. 813–823.

[Kir20] Philip Kirkbride. 'Hardware Details and /dev'. In: *Basic Linux Terminal Tips and Tricks*. Springer, 2020, pp. 185–203.

[Lab] LabStack LLC. *Echo [source code]*. GitHub repository, commit: dec96f0. License: MIT. URL: `https://github.com/labstack/echo`.

[Liu+17] Jinyu Liu et al. 'IdenEH: Identify error-handling code snippets in large-scale software'. In: *2017 17th International Conference on Computational Science and Its Applications (ICCSA)*. IEEE. 2017, pp. 1–8.

[LS98] Jun Lang and David B Stewart. 'A study of the applicability of existing exception-handling techniques to component-based real-time software technology'. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 20.2 (1998), pp. 274–301.

[Mer14] Dirk Merkel. 'Docker: lightweight linux containers for consistent development and deployment'. In: *Linux journal* 2014.239 (2014).

[MG18] Russ McKendrick and Scott Gallagher. *Mastering Docker: Unlock new opportunities using Docker's most advanced features*. Packt Publishing Ltd, 2018.

[Mou15] Adrian Mouat. *Using Docker: Developing and Deploying Software with Containers*. "O'Reilly Media, Inc.", 2015.

[Nur+09] Nurzhan Nurseitov et al. 'Comparison of JSON and XML data interchange formats: a case study.' In: *Caine* 9 (2009), pp. 157–162.

[NVN19] Tam Nguyen, Phong Vu and Tung Nguyen. 'Recommending exception handling code'. In: *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE. 2019, pp. 390–393.

[Ope] Open Source Research Group. *Product Model Toolkit [source code]*. GitHub repository, commit: d3f7c82. License: Apache 2.0. URL: `https://github.com/osrgroup/product-model-toolkit`.

[Osm+17] Haidar Osman et al. 'Exception evolution in long-lived Java systems'. In: *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE. 2017, pp. 302–311.

[Par+12] Fawaz Paraiso et al. 'A federated multi-cloud PaaS infrastructure'. In: *2012 IEEE Fifth International Conference on Cloud Computing*. IEEE. 2012, pp. 392–399.

[PCX11] Dunlu Peng, Lidong Cao and Wenjie Xu. 'Using JSON for data exchanging in web service applications'. In: *Journal of Computational Information Systems* 7.16 (2011), pp. 5883–5890.

[Pot+20] Amit M Potdar et al. 'Performance evaluation of docker container and virtual machine'. In: *Procedia Computer Science* 171 (2020), pp. 1419–1428.

[Pou19] Nigel Poulton. *Docker Deep Dive*. JJNP Consulting Limited, 2019.

[Pra+19] Gede Artha Azriadi Prana et al. 'Categorizing the content of GitHub README files'. In: *Empirical Software Engineering* 24.3 (2019), pp. 1296–1327.

[Ras+19] Raihan ur Rasool et al. 'A novel JSON based regular expression language for pattern matching in the internet of things'. In: *Journal of Ambient Intelligence and Humanized Computing* 10.4 (2019), pp. 1463–1481.

[Rey18] Fanie Reynders. 'Logging and Error Handling'. In: *Modern API Design with ASP. NET Core 2*. Springer, 2018, pp. 113–130.

[Ric+11] S Richard et al. *OSGi in Action: creating modular applications in Java*. Manning, 2011.

[Ric15] Mark Richards. *Software architecture patterns*. Vol. 4. O'Reilly Media, Incorporated 1005 Gravenstein Highway North, Sebastopol, CA . . ., 2015.

[Sar20] Edwin M Sarmiento. 'Docker Images and Containers'. In: *The SQL Server DBA's Guide to Docker Containers*. Springer, 2020, pp. 69–98.

[Say+18] Mohammed Sayagh et al. 'Software configuration engineering in practice: Interviews, survey, and systematic literature review'. In: *IEEE Transactions on Software Engineering* (2018).

[SM12] Audie Sumaray and S Kami Makki. 'A comparison of data serialization formats for optimal efficiency on a mobile platform'. In: *Proceedings of the 6th international conference on ubiquitous information management and communication*. 2012, pp. 1–6.

[Som11] Ian Sommerville. 'Software engineering 9th Edition'. In: *ISBN-10* 137035152 (2011).

[Ste18] Daniel Stenberg. *Everything curl*. 2018. ISBN: 978-91-639-6501-2.

[Stu05] Alexander Stuckenholz. 'Component evolution and versioning state of the art'. In: *ACM SIGSOFT Software Engineering Notes* 30.1 (2005), p. 7.

[Ver13] Vaughn Vernon. *Implementing domain-driven design*. Addison-Wesley, 2013.

[Wol+06] Reinhard Wolfinger et al. 'A component plug-in architecture for the. NET platform'. In: *Joint Modular Languages Conference*. Springer. 2006, pp. 287–305.