Combining Languages to Set a PACS on FHIR

Sébastien Jodogne[0000−0001−6685−7398]

Computer Science and Engineering Department (INGI),
Institute of Information and Communication Technologies, Electronics and Applied Mathematics (ICTEAM), UCLouvain, 1348 Louvain-la-Neuve, Belgium
sebastien.jodogne@uclouvain.be
https://perso.uclouvain.be/sebastien.jodogne/

Abstract. FHIR has quickly emerged as a crucial standard for the interoperability of clinical data. While FHIR supports medical imaging through its `ImagingStudy` resource, the absence of openly available software combining FHIR with DICOM limits the development of applications linking clinical and imaging data. On another front, deep learning applied to medical imaging is becoming increasingly popular, and it is essential to educate a broad audience about its possibilities. However, the current lack of open, easy-to-use environments capable of executing a library of open-access models directly on DICOM images and on a standard computer poses an issue from a pedagogical perspective. In this paper, the Orthanc server is presented as a promising solution to both challenges. Plugins are developed to enable Orthanc to call software libraries written in the Java and Python programming languages. These plugins are then used to turn Orthanc into a FHIR server using the HAPI framework, as well as into a platform for running the inference of a deep learning model for breast imaging using either PyTorch or Deep Java Library. The resulting source code is released as free and open-source software, aiming to promote support for medical imaging in FHIR, to share technological knowledge about medical interoperability and deep learning, and to provide a test environment for the integration of imaging data into clinical workflows.

Keywords: Medical imaging · FHIR · DICOM · Java · Python · Deep learning.

1 Introduction

Healthcare interoperability refers to the ability of different medical systems, software, and devices to communicate and exchange data effectively and efficiently [4]. Healthcare interoperability is critical for patient-centered care and for continuity of care because patient information is typically spread over multiple software entities, such as electronic health records, hospital information system, laboratory information system, or pharmacy information system, possibly across different hospitals and different general practitioners. Healthcare

interoperability is also becoming increasingly important because of the growing interest in telemedicine, in tele-expertise, in patient empowerment, and in artificial intelligence.

Many healthcare interoperability standards have emerged over the years. In the context of medical imaging, DICOM (*Digital Imaging and Communications in Medicine*) has been the *de facto* international standard for the encoding, transmission, storage, and sharing of digital images since 1985 [21]. In the typical clinical workflow, imaging modalities create DICOM instances that are collected on a central PACS server (*Picture Archiving and Communication System*), which in turn provides features for the viewing and distribution of the collected medical images. DICOM is adopted by any hospital in the world for the management of digital images, and its recent DICOMweb extensions have progressively introduced REST APIs (*Representational State Transfer Application Programming Interfaces*) for medical imaging since the 2010s.

Thanks to the availability of DICOM as an open standard, many free and open-source software libraries dedicated to medical imaging have been created. When developing a desktop or a back-end application, the choice of a software library is primarily guided by the programming language used by the development team. Java developers will typically choose `dcm4che` [28], Python developers will use `pydicom` [19], while C++ developers will rely on DCMTK [9]. Beyond these general-purpose libraries that handle the DICOM file format and network protocol, specialized libraries focused on the rendering and processing of medical images have also been developed. For instance, the C++ libraries VTK [24] and ITK [20] are commonly used in software applications that display and analyze medical images. The alignment of medical images is a specific field of research, which has led to the development of Plastimatch, also written in C++ [25]. In radiation therapy and nuclear medicine, the Python library `dicompyler` offers a research platform dedicated to processing the DICOM RT substandard [22]. Finally, developers of deep learning applications for medical imaging commonly rely on libraries written in Python, such as TensorFlow [1] and PyTorch [15].

Besides DICOM, another healthcare interoperability standard is currently attracting increasingly more attention. FHIR (*Fast Healthcare Interoperability Resources*) is an international, modern standard for exchanging clinical information electronically [3]. FHIR is actively developed as an open specification by the HL7 International organization and is released in the public domain under the Creative Commons CC0 license. The original proposal for FHIR was released in 2011, its first draft version (referred to as DSTU1, which stands for *First Draft Standard for Trial Use*) was published in 2014, and its first normative content (called R4 — *Release 4*) appeared in 2019. At the time of writing, the current version of FHIR is R5 (*Release 5*), which was issued in March 2023.

FHIR is based on a set of standardized resources, which represent discrete pieces of healthcare data, such as patients, medications, allergies, or observations. These resources act as building blocks that can be combined to represent complex clinical concepts and workflows. FHIR employs REST APIs to access and manipulate these resources. The fact that REST APIs are widely used for

the development of Web and mobile applications explains the growing popularity of FHIR among software engineers. FHIR also fully embraces existing medical terminologies such as SNOMED-CT and LOINC, which enables semantic interoperability, ensuring that healthcare data is understood consistently across different systems and organizations. The most complete implementation of the FHIR standard is the HAPI framework [2]. HAPI closely follows the release cycle of the FHIR standard. It can be used as a building block to design both FHIR clients and FHIR servers. HAPI is free and open-source software that is written in the Java programming language.

Due to the universal adoption of DICOM by hospitals, researchers, and vendors, FHIR does not attempt to introduce a new competing standard for medical imaging. Instead, FHIR defines the `ImagingStudy` resource as its official method for connecting clinical and imaging data. The `ImagingStudy` resource does not store all the DICOM tag values or pixel data. Instead, it uses the DICOMweb API to reference DICOM entities stored on a separate PACS server. Unfortunately, there is currently a lack of open, reference implementation combining FHIR with DICOM. While public implementations of `ImagingStudy` do exist, such as those found in HAPI and available in public FHIR test sandboxes, their content is not linked to an actual DICOMweb endpoint. The availability of a DICOM server implementing both the DICOMweb and the FHIR REST APIs could hopefully promote the support of medical imaging in FHIR, while providing a testing environment for the development of new applications combining clinical with imaging data.

In addition, as discussed above, a complex application that processes medical images according to standard-compliant workflows will presumably integrate multiple programming languages to benefit from different software libraries. For instance, adding FHIR support to a medical imaging application will likely involve using Java, while deploying an artificial intelligence algorithm will likely involve using Python. The typical software architecture therefore consists in implementing multiple microservices written in different languages that operate alongside each other, each dedicated to the handling of a part of the processing pipeline. These services can then be encapsulated as containers and orchestrated using technologies like Docker Swarm or Kubernetes. This approach is perfectly adapted if creating a proprietary platform with a professional team dedicated to its administration and maintenance. It is less suitable if developing a desktop or back-end application that needs to be deployed in a less controlled environment or where users have less expertise in system administration.

The contributions of this paper are threefold. Firstly, a free and open-source PACS server supporting DICOMweb and written in C++ is extended to incorporate the capability of invoking plugins written in Java or Python. This allows software integrators to choose the programming language (C++, Java, or Python) that is best suited for calling an existing software library for medical imaging. Python plugins are especially adapted to automate imaging workflows. Secondly, it is demonstrated how a Java plugin can be created to integrate with HAPI, allowing the PACS server to serve the parts of the REST API of FHIR

related to medical imaging in addition to the DICOMweb endpoint. Because these software modules are running in the same process, the FHIR resources are guaranteed to remain continuously synchronized with the DICOM instances. Thirdly, it is shown how both Java and Python plugins can execute deep learning inference directly on the DICOM instances stored by the PACS server. The Java plugin is proven to be particularly well-suited for simplifying the deployment of deep learning models by distributing the deep learning inference engine as a single `.jar` archive. The preliminary version of this paper that appeared in the Proceedings of the 17th International Joint Conference on Biomedical Engineering Systems and Technologies (BIOSTEC 2024) introduced the Java plugin together with the FHIR server [14]. This extended version introduces the Python plugin and the deep learning applications.

2 Background

This section reviews the various building blocks that will be combined to introduce a PACS server compatible with multiple programming languages. It successively describes the Orthanc server, the DICOM model of the real world, the `ImagingStudy` resource of the FHIR standard, and the HAPI framework.

2.1 Orthanc Server

In the context of medical imaging, multiple free and open-source PACS servers are available, such as `dcm4chee` [28] and Dicoogle [8]. This work is focused on the Orthanc PACS server, which is licensed under the GPLv3 license and is developed in the C++ programming language [12]. Besides its native DICOM primitives, Orthanc comes with a REST API that can be used to script imaging workflows, which is widely used to set up auto-routing between local DICOM servers or between remote sites. This REST API can be used to search the content of the DICOM database and to retrieve the content of individual DICOM resources, which will be used in this work. Orthanc has the advantage of being lightweight, cross-platform, and extensible: Multiple Orthanc servers can run on any computer at low cost, without necessitating any complex configuration.

Very importantly, Orthanc proposes a plugin mechanism that can be used to extend its core features using the C language. An Orthanc plugin takes the form of a shared library (i.e., a `.so` file under GNU/Linux distributions or a DLL under Microsoft Windows). Plugins notably have full access to the REST API of Orthanc and can add new routes to this REST API. The fact that Orthanc plugins can extend the REST API of Orthanc has already been used to provide an implementation of the DICOMweb standard [12], as well as to integrate popular Web viewers such as OHIF [30] and Kitware VolView [29]. In this work, the extensibility of Orthanc will be utilized to seamlessly integrate it with the Java and Python computer languages, providing a way to efficiently design processing pipelines that can be deployed on any computer by leveraging the small footprint of Orthanc.

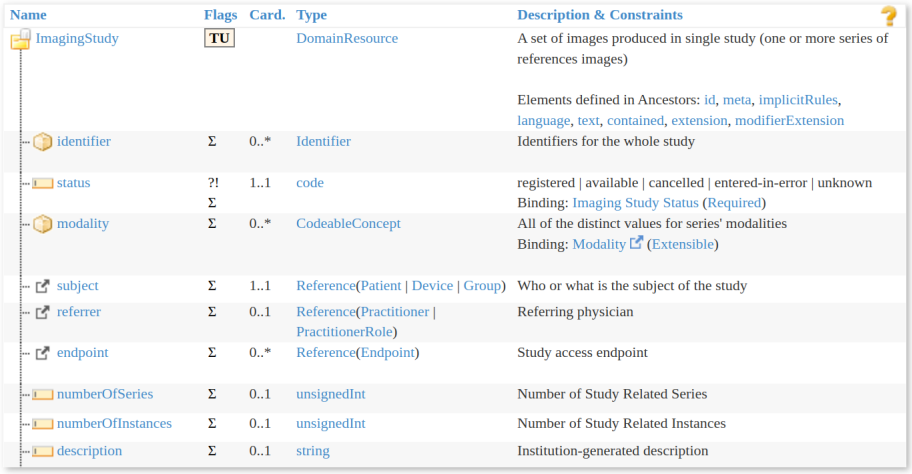
2.2 DICOM Model of the Real World

To understand the connection between DICOM and FHIR, it is essential to comprehend how the DICOM specification organizes the medical imaging resources. In this so-called "DICOM model of the real world," a patient benefits from a sequence of imaging studies during his or her life. Each of these studies consists of a set of series. In turn, each series is made of a set of DICOM instances that are created by the same imaging device. An isolated DICOM instance can often (but not always) be thought of as one 2D image slice that is associated with a dataset providing patient-related and acquisition-related information that enriches the pixel data. This dataset is a tree-like structure that associates values to standardized DICOM tags, identified by a pair of hexadecimal numbers. For instance, a DICOM study generated by a PET-CT-scanner would typically correspond to a set of two DICOM series (representing the PET volume and the CT volume), with the two series encoded as a set of DICOM instances that contain the individual 2D axial slices of the parent 3D volume.

Each resource in the patient/study/series/instance hierarchy is identified by one specific DICOM tag. The patient level is identified by the value of the "*Patient ID*" `(0x0010,0x0020)` DICOM tag, the study level by the "*Study Instance UID*" `(0x0020,0x000d)` tag, the series level by the "*Series Instance UID*" `(0x0020,0x000e)` tag, and finally the instance level by the "*SOP Instance UID*" `(0x0008,0x0018)` tag. While the UIDs (*Unique IDentifiers*) at the study, series, and instance levels are assumed to be globally unique, the "*Patient ID*" tag is only locally unique within an institution, which necessitates reconciliation in multi-centric contexts. PACS systems typically use the values of these four important DICOM tags to index the DICOM resources in a relational database. The basic structure of the REST API of Orthanc is directly mapped on this four-level DICOM model of the real world. The DICOMweb substandard, for its part, organizes its REST API using only the globally unique identifiers at the study, series, and instance levels.

2.3 FHIR for Medical Imaging

FHIR brings support for medical imaging through its `ImagingStudy` resource. The `ImagingStudy` resource includes the globally unique identifiers of DICOM resources, along with minimal information about their content. This allows linking patient data stored in the FHIR server to a separate DICOMweb server that is responsible for the actual storage of the medical images. Consequently, the `ImagingStudy` resource can be thought of as a higher-level service on the top of DICOMweb. `ImagingStudy` is associated with maturity level 4 in FHIR R5. This maturity level indicates that `ImagingStudy` has been implemented in multiple prototype projects, hereby showing a certain level of stability. Nonetheless, this also means that the resource has been implemented in less than five independent production systems, or in no more than one country yet. This fact reinforces the interest in developing an open implementation of `ImagingStudy` to raise the maturity level of the associated part of the FHIR standard.



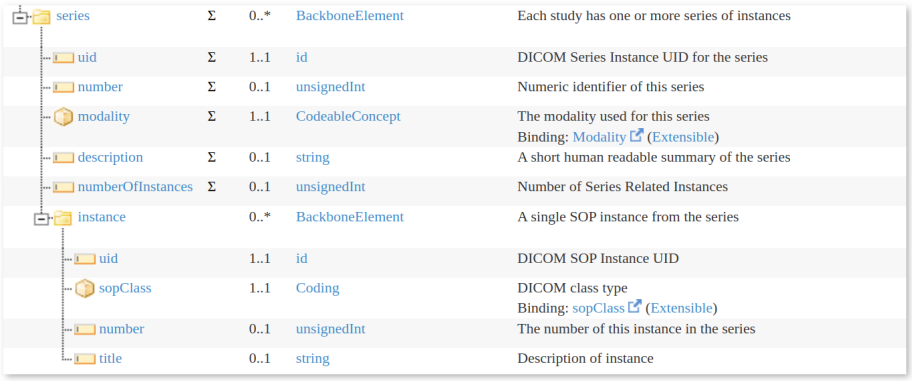
Name	Flags	Card.	Type	Description & Constraints
ImagingStudy	TU		DomainResource	A set of images produced in single study (one or more series of references images) Elements defined in Ancestors: id, meta, implicitRules, language, text, contained, extension, modifierExtension
identifier	Σ	0..*	Identifier	Identifiers for the whole study
status	?! Σ	1..1	code	registered | available | cancelled | entered-in-error | unknown Binding: Imaging Study Status (Required)
modality	Σ	0..*	CodeableConcept	All of the distinct values for series' modalities Binding: Modality (Extensible)
subject	Σ	1..1	Reference(Patient | Device | Group)	Who or what is the subject of the study
referrer	Σ	0..1	Reference(Practitioner | PractitionerRole)	Referring physician
endpoint	Σ	0..*	Reference(Endpoint)	Study access endpoint
numberOfSeries	Σ	0..1	unsignedInt	Number of Study Related Series
numberOfInstances	Σ	0..1	unsignedInt	Number of Study Related Instances
description	Σ	0..1	string	Institution-generated description

Fig. 1. Excerpt of the main structure of the `ImagingStudy` resource in FHIR R5 (Figure from [14]). For each field in the resource, the "Card." column indicates its cardinality, and the "Type" column indicates the data type of its values. Fields whose minimum cardinality is 1 are mandatory. This is a screenshot of the official specification of FHIR available at: https://hl7.org/fhir/R5/ImagingStudy.html.

The main content of the `ImagingStudy` resource is depicted in Figure 1. One instance of this FHIR resource corresponds to one DICOM study. To create the link between the `ImagingStudy` resource and its associated DICOM study, the `identifier` field of the FHIR resource must contain the value of the "*Study Instance UID*" DICOM tag. This is because FHIR does not store the images, but rather pointers to DICOM resources. The DICOM resources themselves must be stored in a separate DICOMweb-compliant server, the address of which must be provided in the `endpoint` field. The latter field contains a reference to a separate `Endpoint` FHIR resource that specifies the connection parameters to the DICOMweb server, including its URL. If needed, the referenced DICOM instances can be retrieved or rendered using the DICOMweb WADO-RS[1] service offered by the DICOMweb endpoint. Most existing public sandboxes for FHIR do not contain a working `endpoint` field, which prevents access to the content of the DICOM studies that are indexed by the `ImagingStudy` resources.

As can be seen in Figure 1, besides the `Endpoint` and `ImagingStudy` resources, a FHIR server for medical imaging must also support the `Patient` resource, because the `subject` field is mandatory and must contain a reference to the patient who is associated with the DICOM study. Consequently, similarly to the one-to-one relation that exists between a FHIR `ImagingStudy` and a DI-

[1] The WADO acronym stands for "*Web Access to DICOM Objects,*" while RS stands for "*RESTful Service.*"



series	Σ	0..*	BackboneElement	Each study has one or more series of instances
uid	Σ	1..1	id	DICOM Series Instance UID for the series
number	Σ	0..1	unsignedInt	Numeric identifier of this series
modality	Σ	1..1	CodeableConcept	The modality used for this series Binding: Modality (Extensible)
description	Σ	0..1	string	A short human readable summary of the series
numberOfInstances	Σ	0..1	unsignedInt	Number of Series Related Instances
instance		0..*	BackboneElement	A single SOP instance from the series
uid		1..1	id	DICOM SOP Instance UID
sopClass		1..1	Coding	DICOM class type Binding: sopClass (Extensible)
number		0..1	unsignedInt	The number of this instance in the series
title		0..1	string	Description of instance

Fig. 2. Excerpt of the information to be provided in the `ImagingStudy` for each of its associated DICOM series (Figure from [14]). This screenshot is the continuation of Figure 1 in the FHIR standard.

COM study, there exists a mapping between a `Patient` FHIR resource and a DICOM patient.

In addition to the patients and to the studies, FHIR for medical imaging also provides pointers to both the DICOM series and the DICOM instances. To this end, the `ImagingStudy` resource contains a field named `series`, as shown in Figure 2. This field stores an array, each item of which declares one DICOM series whose "*Series Instance UID*" DICOM tag is contained in the `uid` subfield. In turn, each item in the `series` field contains a subarray called `instance` that provides the list of the DICOM instances of the parent series, in which the `uid` subfield indicates the "*SOP Instance UID*" DICOM tag of the individual instances. This means that, contrary to DICOM studies that are mapped as separate `ImagingStudy` resources, FHIR does not encode DICOM series and DICOM instances as standalone resources: The series or instance information can only be retrieved by accessing their parent study.

Figure 3 summarizes the relations between the DICOM and FHIR models of the real world. As can be observed in this figure, careful attention must be given to continuously updating FHIR resources, as the content of the FHIR server needs to be updated each time a new instance is added to or deleted from the DICOMweb server. This requires robust communication mechanisms between these two servers. In this work, we will adopt an approach where the `ImagingStudy` FHIR resources are not explicitly stored but are generated on-the-fly as needed by querying the content of the Orthanc server.

Previous research work about the `ImagingStudy` resource has consisted in providing static databases as pedagogical resources to learn FHIR for medical imaging [11], in deploying a multi-site infrastructure dedicated to pediatric scoliosis [26], in collecting radiation dose information [5], and in implementing complex clinical imaging workflows [27]. However, to the best of our knowledge,

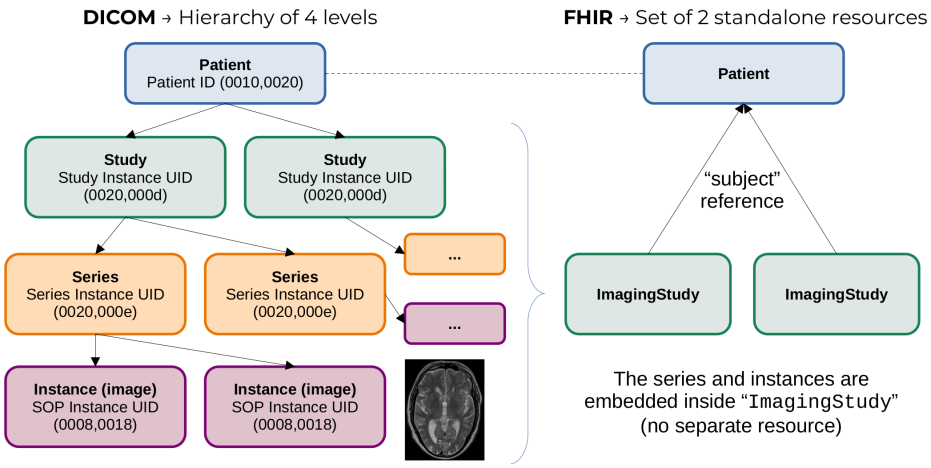


Fig. 3. Comparison between the DICOM and FHIR models of the real world.

no free and open-source PACS server can currently act as a standalone FHIR server that transparently, automatically synchronizes with the DICOM resources it publishes using DICOMweb.

2.4 HAPI Framework

As explained in the Introduction, the HAPI framework proposes an implementation of the FHIR standard in the Java programming language. The HAPI framework can be used either as a software library for creating FHIR clients and servers, or as a fully-fledged FHIR server. Importantly, HAPI supports the `ImagingStudy` resource that is needed to implement medical imaging in FHIR. HAPI is licensed under the Apache Software License 2.0.

More precisely, when it comes to designing a FHIR server, the HAPI framework can be used in two distinct ways. On the one hand, the HAPI project provides the so-called "HAPI Plain Server" as a software library that can be used to create custom FHIR endpoints against arbitrary data sources. On the other hand, the "HAPI JPA Server" is a fully standalone application that incorporates JPA (*Java Persistence API*) to store and retrieve FHIR resources in a relational database such as PostgreSQL. This paper will focus on the combination of the HAPI framework with the Orthanc server to provide a tight integration between FHIR and DICOM.

3 Methods

In this section, the software architecture that is used to create Java and Python plugins for Orthanc is first introduced. Then, the integration between Orthanc and HAPI is described.

3.1 Wrapping the Orthanc SDK in Java and Python

The Orthanc plugin SDK (*Software Development Kit*) proposes a large library of functions declared in a plain C header that can be used to invoke the various primitives implemented by the Orthanc core. For instance, the Orthanc plugin SDK allows registering callbacks to process incoming HTTP requests, which can be used to extend the built-in REST API of Orthanc. The SDK can also make internal calls to the built-in REST API, enabling a wide variety of operations. Consequently, it is essential for the Java and Python plugins to be able to call these native functions implemented by the Orthanc runtime.

To this end, this work introduces different Python scripts to automatically wrap the Orthanc plugin SDK as a collection of Java and Python classes. The first Python script analyzes the content of the `OrthancCPlugin.h` header that defines the Orthanc plugin SDK, then creates a code model that is saved in the JSON file format. This script detects all the global functions that are part of the Orthanc SDK and analyzes their signature. In addition, it automatically discovers the native objects that are published by the Orthanc core (such as DICOM instances, images, or responses to REST requests) and that are managed through C global functions defining their methods, constructors, and destructors. The resulting code model also contains the documentation. This analysis is achieved using the `libclang` library that is part of the LLVM project [16].

Once the code model is extracted, additional Python scripts are used to generate language-specific wrappers around the native global functions of the Orthanc SDK. These Python scripts depend upon the target language and take the code model as their input. If targeting Java, one separate `.java` class file is generated to encapsulate each native Orthanc object, and a separate `Functions.java` class contains the wrappers around the remaining global functions that do not manage objects. The generation script automatically converts the C data types to their corresponding Java equivalents. Each generated Java class belongs to the `be.uclouvain.orthanc` package and is enriched with a Javadoc-compliant documentation. All the wrapped methods are tagged using the `native` Java keyword, which indicates that the actual implementation of the method is not provided in Java. A separate Python script generates a single, large C++ source file called `NativeSDK.cpp` that contains the implementation of each of those `native` methods. This C++ code is responsible for first reading the content of the Java objects passed as arguments through JNI (*Java Native Interface*), then calling the concrete implementation provided by the Orthanc plugin SDK, and finally converting the possibly resulting C value as a Java data type (which may involve the creation of a Java wrapper around an object managed by Orthanc).

Another Python script has been developed based on the same principle if targeting the Python language. In the case of Python, there is no equivalent to the `native` keyword and it is not necessary to explicitly create standalone `.py` files. Indeed, the Python C API library (often referred to as `libpython`) provides the functionality for defining new Python classes and modules directly from C or C++ code. Therefore, our generation script takes the code model and produces a single C++ source file. This file first registers all the classes, methods,

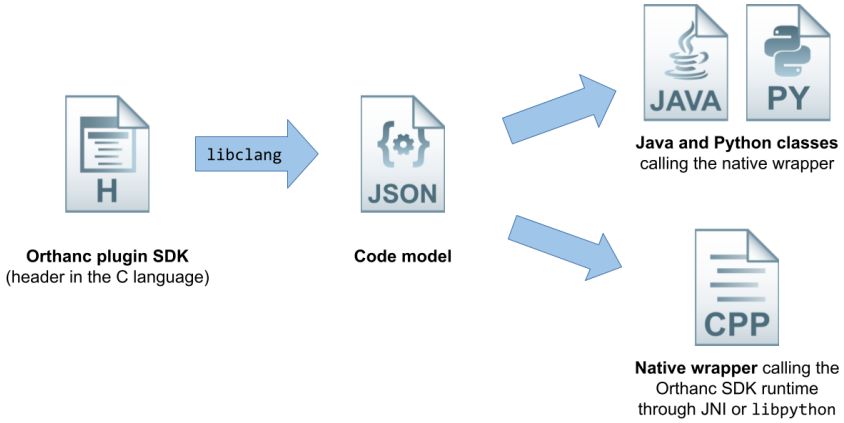


Fig. 4. Mapping the Orthanc SDK in the Java and Python languages. Python classes are not explicitly stored as standalone files but are directly registered using `libpython`.

and global functions in a Python module named `orthanc`, and then associates each of them with a C++ native function. As in the case of Java, these native functions are in charge of reading the content of the Python objects passed as arguments through `libpython`, of calling the corresponding C primitives in the Orthanc SDK, and possibly of creating a result value as a Python object using `libpython` (again, this last step may involve the creation of a Python wrapper around an Orthanc object). One separate Python file called `orthanc.pyi` is also generated, which is a Python interface providing documentation and type hints for static type checking and auto-completion in an IDE (*Integrated Development Environment*).

The global generation process is represented in Figure 4. Thanks to this automated mechanism, 122 functions from the Orthanc plugin SDK are automatically wrapped in the Java and Python languages, out of a total of 165 primitives available in Orthanc 1.10.0. This represents a coverage of about 74%. No human intervention is required, which would have been highly time-consuming and error-prone, especially when keeping up with new releases of the Orthanc SDK. Note that the extracted code model could be used to wrap the Orthanc plugin SDK in other programming languages beyond Java and Python.

The primitives that are not automatically wrapped mostly correspond to the registration of callbacks, which requires a manual implementation due to their language-specific nature. The most important example of a callback corresponds to the handling of HTTP requests received by the Web server of Orthanc, which can be used to add new routes in the REST API. Another example is the callback that is triggered when specific events occur in the Orthanc server, for instance when Orthanc starts or stops, or when a new DICOM instance is received. To deal with such situations, the Java and Python plugins register internal C functions as native callbacks using the Orthanc plugin SDK. Once one of these callback functions is invoked by the core of Orthanc, it uses either the JNI or

`libpython` to call the corresponding Java callable or Python handler that were registered by the user.

The C++ code that is generated either for Java or Python is linked as a shared library that can be loaded as a plugin for Orthanc, as explained in Section 2.1. This shared library also contains code for configuring the plugin and for starting either a Java virtual machine (using JNI) or an embedded Python interpreter (using `libpython`). In the case of Java, the configuration specifies a user-defined Java class that is included in the Java classpath, enabling this class to install the Java callables that will handle the various events processed by Orthanc. Likewise, in Python, the configuration provides the filesystem path to a user-defined Python script that is responsible for registering handlers.

3.2 Integrating Orthanc and HAPI

This section describes how FHIR support can be added to Orthanc by integrating HAPI. Figure 5 depicts three possible high-level architectures to combine HAPI with Orthanc. The first architecture consists in executing the standalone HAPI JPA Server next to the Orthanc server, in two separate processes. In this architecture, a third process is used to continuously synchronize the content of the HAPI JPA server with Orthanc. To this end, the synchronization process monitors the addition and removal of DICOM resources using the REST API of Orthanc, and replicates the detected modifications in the HAPI JPA Server. In this architecture, the FHIR and DICOM databases are kept separate, which results in a substantial risk of de-synchronization between the two servers, and which requires a specific infrastructure dedicated to the execution of the JPA-compliant database.

The second possible architecture consists in replacing the standalone HAPI JPA Server by the HAPI Plain Server software library. In this architecture, a custom HAPI Plain Server is deployed in a standalone Java servlet container and acts as a facade in front of the Orthanc server. This facade is responsible for generating the `ImagingStudy` and `Patient` FHIR resources after querying the REST API of Orthanc. This solution has the great advantage of using a single database that is shared by the FHIR and DICOM servers. However, this architecture implies the deployment of two distinct processes and two distinct Web servers (one for the servlet container, and one for Orthanc), which keeps the setup slightly complex, especially when it comes to the mapping of the DICOMweb API.

Consequently, this work focuses on a third architecture, in which a HAPI Plain Server is directly branched into the Web server of Orthanc. In this architecture, Orthanc acts similarly to a Java servlet container, by redirecting all the FHIR requests to the HAPI framework. This architecture offers the advantage of requiring a single database and a single process, which further simplifies the setup, and of providing consistent access to both FHIR resources and DICOMweb routes, as only a single Web server is involved. The technical challenge with this third architecture is that Orthanc is developed in C++, while HAPI

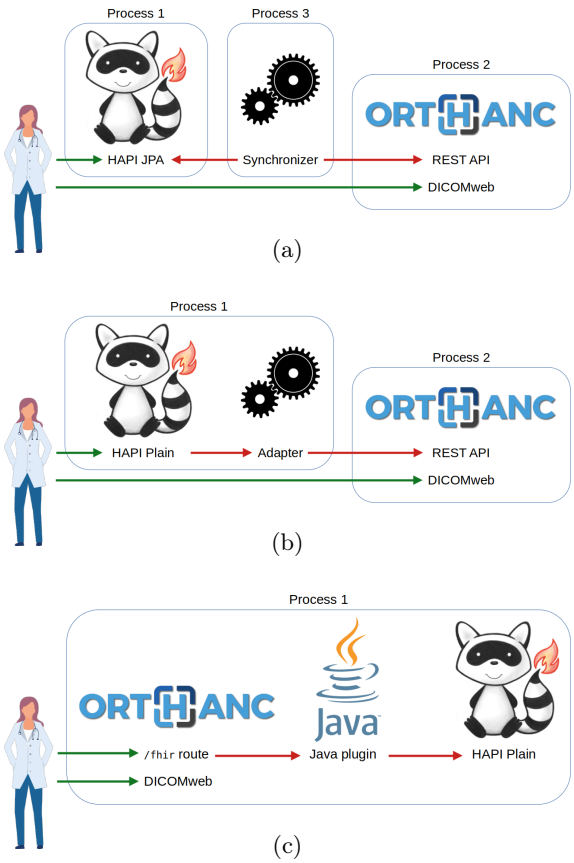


Fig. 5. Three possible architectures to integrate the Orthanc server with the HAPI framework: (a) continuous synchronization between Orthanc and the HAPI JPA Server, (b) custom HAPI Plain Server with a data source corresponding to Orthanc, or (c) branching HAPI Plain Server as a plugin to Orthanc (Figure from [14]).

is written in Java, necessitating the creation of a bridge between these two languages. The solution is to utilize the Java plugin presented in Section 3.1 to enable the use of the HAPI Plain Server library directly within Orthanc.

Nonetheless, the HAPI Plain Server library is based on the Java servlets API, which does not directly align with the primitives offered by the Orthanc plugin SDK. This compatibility issue is addressed by utilizing the two standard classes `MockHttpServletRequest` and `MockHttpServletResponse` provided by the Spring Framework. These two classes can be used to turn the Web server of Orthanc into a servlet container: Whenever an HTTP request must be handled by the Java callable, an instance of the `MockHttpServletRequest` class is filled with the parameters of the HTTP request and is sent to the Java servlet. In return, the Java servlet fills a `MockHttpServletResponse` whose content can be

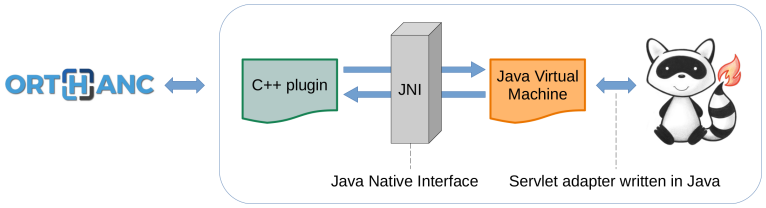


Fig. 6. Integration between Orthanc and HAPI Plain Server. The arrow pointing to the right (resp. to the left) in the servlet adapter corresponds to Java objects of the class `MockHttpServletRequest` (resp. `MockHttpServletResponse`) of the Spring framework.

unwrapped and sent back to the client through the Orthanc Web server. By default, the HAPI Plain Server servlet is branched at the root path `/fhir/` in the REST API of Orthanc. This integration is depicted on Figure 6. It will be referred to as the "FHIR plugin for Orthanc," although it is a slight misuse of language, as this integration actually involves executing user-defined Java code that utilizes the HAPI Plain Server library within the environment provided by the Java plugin for Orthanc.

The last step to bring FHIR to Orthanc consists in implementing read-only HAPI providers that are responsible to retrieve and search the `ImagingStudy`, `Patient`, and `Endpoint` FHIR resources. To retrieve resources, the FHIR plugin implements the `IResourceProvider` interface, as defined by the HAPI Plain Server library, for each of those categories of resources. The `Endpoint` resource simply declares the address of the DICOMweb plugin, which is a purely static behavior that does not require querying the content of the Orthanc database. As far as the `ImagingStudy` (resp. `Patient`) resources are concerned, the content of an individual DICOM study (resp. DICOM patient) with identifier `id` can be retrieved by issuing a GET request to the `/studies/{id}` (resp. `/patients/{id}`) route in the built-in REST API of Orthanc. This route returns a JSON object whose content can easily be converted as an `ImagingStudy` (resp. `Patient`) Java object, as expected by the HAPI framework.

Searching over the `ImagingStudy` and `Patient` FHIR resources is implemented by issuing POST requests to the `/tools/find` route of the built-in REST API of Orthanc. This route executes a lookup operation on the Orthanc database, which can be filtered by specifying values for selected DICOM tags. These constraints can directly be derived from the parameters of the FHIR query. The route answers with a JSON array that contains the matching resources, which are converted into the `List<ImagingStudy>` and `List<Patient>` Java objects that are expected by the HAPI framework.

4 Results

Section 3.2 explained how the HAPI framework can be integrated as a plugin to Orthanc, which provides a working, free and open-source implementation of

a FHIR server for medical imaging that is tightly coupled with a PACS server. This integration leverages another key deliverable of this research work described in Section 3.1, namely the ability to create plugins for Orthanc that can use the Java and Python programming languages. Some examples of Java and Python plugins, as well as an interaction with the FHIR server plugin for Orthanc, are now discussed. This section concludes with an application of the Java and Python plugins to deep learning for breast imaging.

4.1 Using `dcm4che` in Orthanc

The `dcm4che` project was previously mentioned as a highly popular software library that provides an implementation of the DICOM standard in Java. This contrasts with Orthanc that internally uses the DCMTK library written in C++. The new Java plugin for Orthanc can be used to complement DCMTK with `dcm4che`, allowing it to benefit from the respective strengths of these two software libraries inside the Orthanc ecosystem. Depending on their technical background, a software developer might also find it easier to develop a DICOM pipeline as a Java plugin rather than as a C++ plugin.

As an illustration, Figure 7 shows a sample Java plugin that uses the `dcm4che` library. The configuration file of Orthanc must specify both the classpath for the Java virtual machine (which must contain the `.jar` archive providing `dcm4che` and the bytecode of the user-defined `Main` class), and the name of the main Java class whose `static` method will be executed during the initialization of the plugin (in this case, `Main`). In this sample plugin, the `static` method registers a new route in the REST API of Orthanc using the method `Callbacks.register()` that is manually implemented in the Java plugin. Whenever an HTTP request is triggered against the `/dcm4che-parse` route, the custom Java callable is invoked. Inside the Java callable, the body of incoming HTTP POST requests is parsed as a DICOM instance using `dcm4che`. On success, the client receives the dump of the DICOM dataset, as generated by `dcm4che`.

4.2 FHIR Server in Orthanc

Starting the FHIR server plugin for Orthanc simply consists in writing a configuration file that loads the Java classes of the FHIR plugin for Orthanc together with the Java classes of the HAPI framework. A single precompiled `.jar` archive that bundles all the required dependencies is freely available for download on the Orthanc official homepage. This package is generated by Maven and its `maven-assembly-plugin` plugin. As soon as Orthanc is running, its FHIR server is accessible at default URL: http://localhost:8042/fhir/. FHIR clients can access the content of Orthanc using this URL. The `Endpoint` resource of the FHIR server is automatically mapped to the route of the DICOMweb server of Orthanc, whose default location corresponds to: http://localhost:8042/dicom-web/studies. This means that only one single Web server is running at any time, with this Web server being shared by the REST API of Orthanc, by the DICOMweb endpoint, and by a HAPI Plain Server.

```
/**
 * This plugin can be triggered from the command line as follows:
 * $ curl http://localhost:8042/dcm4che-parse -X POST --data-binary @sample.dcm
 **/

import org.dcm4che3.data.Attributes;
import org.dcm4che3.io.DicomInputStream;

public class Main {
    static {
        Callbacks.register("/dcm4che-parse", new Callbacks.OnRestRequest() {
            @Override
            public void call(RestOutput output,
                             HttpMethod method,
                             String uri,
                             String[] regularExpressionGroups,
                             Map<String, String> headers,
                             Map<String, String> getParameters,
                             byte[] body) {
                if (method != HttpMethod.POST) {
                    output.sendMethodNotAllowed("POST");  // Answer with HTTP status 405
                } else {
                    ByteArrayInputStream stream = new ByteArrayInputStream(body);

                    try (DicomInputStream din = new DicomInputStream(stream)) {
                        Attributes dataset = din.readDataset();
                        output.answerBuffer(dataset.toString().getBytes(), "text/plain");
                    } catch (IOException e) {
                        output.sendHttpStatus((short) 400, "Cannot parse DICOM
                             file\n".getBytes());
                    }
                }
            }
        });
    }
}
```

Fig. 7. Example of a Java plugin for Orthanc that uses `dcm4che` to dump the content of a DICOM file provided in the body of a POST request (Figure adapted from [14]).

As an illustration, Figure 8 shows an interactive session in which the FHIR server plugin for Orthanc answers queries issued by the FHIRPACK client. Importantly, the reported FHIR resources (i.e., `Patient` and `ImagingStudy`) are generated on-the-fly using a HAPI Plain Server, directly from the DICOM resources that are stored by Orthanc: No separate database dedicated to FHIR resources is used, and all the information is extracted from the Orthanc database. This ensures that the DICOMweb and the FHIR views of the content of the Orthanc server are always perfectly synchronized.

4.3 Using `pydicom` in Orthanc

The `pydicom` library is the Python equivalent to `dcm4che` and DCMTK. Just like `dcm4che` can be accessed from Orthanc through the Java plugin, `pydicom` is readily available through the Python plugin. Many end users will find it natural to implement custom pipelines as Python plugins for Orthanc, since this approach does not require the use of a compiler, which contrasts with the C, C++,

```
$ fp -s http://localhost:8042/fhir -o "getPatients" -p all -o "gatherSimplePaths id name.family birthDate"

0      fYET5.0   [COMUNIX]  1941-09-01
1        5Yp0E   [BRAINIX]  1949-03-01
2     Vafk,T,6    [PHENIX]  1991-01-01
3       SOtNwu   [INCISIX]        None
4  ozp00SjY2xG      [KNIX]        None
5       vAD7q3       [VIX]        None


$ fp -s http://localhost:8042/fhir -o "getImagingStudies" -p all -o \
  "gatherSimplePaths identifier.value subject.reference description numberOfSeries numberOfInstances"

                      identifier.value    subject.reference                description  numberOfSeries  numberOfInstances
0  [urn:oid:2.16.840.1.113669.632.20...       Patient/vAD7q3  Pied_cheville_UHR (Adulte)               1                250
1  [urn:oid:2.16.840.1.113669.632.20...        Patient/5Yp0E  IRM cérébrale, neuro-crâne               7                232
2  [urn:oid:2.16.840.1.113669.632.20...     Patient/Vafk,T,6       CT2 tête, face, sinus               3                723
3  [urn:oid:1.2.840.113745.101000.10...      Patient/fYET5.0       Neck^1HEAD_NECK_PETCT               2                166
4  [urn:oid:1.2.840.113619.2.176.202...  Patient/ozp00SjY2xG                   Knee (R)               6                135
5  [urn:oid:2.16.840.1.113669.632.20...       Patient/SOtNwu        Tête^Dental (Adulte)               1                166
```

Fig. 8. Sample interactive session of the FHIRPACK client against an Orthanc server that contains sample DICOM studies provided by the OsiriX project [23] (Figure adapted from [14]). The first request lists the `Patient` FHIR resources, while the second lists the `ImagingStudy` FHIR resources.

and Java plugins for Orthanc. The Python source code in Figure 9 implements the same behavior as the Java source code of Figure 7, substituting `dcm4che` by `pydicom`. Moreover, the Python source code can access any module that is installed in the Python environment. For instance, the sample code of Figure 10 shows how to combine Orthanc with PIL (*Python Imaging Library*) to generate thumbnails of DICOM instances stored in the Orthanc database.

4.4 Deep Learning for Breast Imaging

The previous section has indicated that any Python library can be invoked from Orthanc through the Python plugin. This evidently includes Python libraries for deep learning such as TensorFlow and PyTorch. This capability has very recently been exploited to create a platform integrated into Orthanc for the automated detection of masses on mammograms [7]. A RetinaNet model [18] based on a ResNet-50 backbone [10] was trained on the CBIS-DDSM (*Curated Breast Imaging Subset*) dataset [17] to detect the bounding boxes of breast masses. The trained model offers detection performance in line with the state-of-the-art on the CBIS-DDSM dataset. The Python plugin for Orthanc described in Section 3.1 of this work was then used to invoke PyTorch for performing inference with the trained model on the DICOM instances stored in the Orthanc database. The output bounding boxes were then encoded as standard-compliant DICOM-SR (*Structured Report*) instances using the `highdicom` Python library [6]. These DICOM-SR instances were stored into Orthanc next to the input DICOM mammograms. Finally, the Stone Web viewer [13] was specifically extended to render such DICOM-SR instances on the top of the mammograms.

Even though this workflow using the Python plugin for Orthanc is perfectly functional, it requires the installation of a heavyweight Python virtual environ-

```python
# This plugin can be triggered from the command line as follows:
# $ curl http://localhost:8042/pydicom-parse -X POST --data-binary @sample.dcm

import io
import orthanc
import pydicom

def ParseInstance(output: orthanc.HttpMethod, uri: str, **request):
    if request['method'] != 'POST':
        output.SendMethodNotAllowed('POST')  # Answer with HTTP status 405
    else:
        stream = io.BytesIO(request['body'])
        dicom = pydicom.dcmread(stream)
        output.AnswerBuffer(str(dicom), 'text/plain')

orthanc.RegisterRestCallback('/pydicom-parse', ParseInstance)
```

Fig. 9. Example of a Python plugin for Orthanc that uses `pydicom` to dump the content of a DICOM file. This offers the same functionality as the Java sample code of Figure 7.

```python
# This plugin can be triggered from the command line as follows:
# $ curl http://localhost:8042/thumbnail/19816330-cb02e1cf-df3a8fe8-bf510623-ccefe9f5

import PIL.Image
import io
import orthanc

def CreateThumbnail(output: orthanc.HttpMethod, uri: str, **request):
    if request['method'] == 'GET':
        # Retrieve the instance ID from the regular expression (*)
        instanceId = request['groups'][0]

        # Render the instance, then open it in Python using PIL/Pillow
        png = orthanc.RestApiGet('/instances/%s/rendered' % instanceId)
        image = PIL.Image.open(io.BytesIO(png))

        # Downsize the image as a 64x64 thumbnail
        image.thumbnail((64, 64), PIL.Image.ANTIALIAS)

        # Save the thumbnail as a JPEG image, then send the buffer to the caller
        jpeg = io.BytesIO()
        image.save(jpeg, format = 'JPEG', quality = 80)
        jpeg.seek(0)
        output.AnswerBuffer(jpeg.read(), 'image/jpeg')

    else:
        output.SendMethodNotAllowed('GET')

orthanc.RegisterRestCallback('/thumbnail/(.*)', CreateThumbnail)  # (*)
```

Fig. 10. Sample Python plugin that generates the thumbnail of a DICOM instance stored in the Orthanc database using the PIL module. The identifier of the DICOM instance of interest must be part of the URL.

ment that includes numerous third-party dependencies and whose proper setup may pose challenges for end users. But the Java ecosystem includes the Deep Java Library (DJL), a free and open-source deep learning framework that simplifies the development and deployment of machine learning models in Java applications. Moreover, unlike Python virtual environments, Java allows the bundling of all the third-party dependencies into a single, durable, cross-platform `.jar` archive, which enhances the portability of the runtime environment for deep learning inference. These elements position Java as an interesting alternative to Python when deep learning models need to be integrated into Orthanc and deployed for inference to a broad audience.

In this work, the same RetinaNet model that was trained using Python was exported as a TorchScript, a serialization format for PyTorch models suitable for various runtime environments. Models serialized with TorchScript can readily be imported by the Deep Java Library. Thanks to this approach, the Java plugin for Orthanc can in theory immediately replace the Python plugin to perform inference of the exported RetinaNet model on DICOM instances managed by Orthanc. However, certain components of the RetinaNet architecture, such as the non-maximum suppression step, cannot be exported as a TorchScript. This posed a significant technical challenge, necessitating a manual re-implementation of these components to fully restore the functionality of the RetinaNet model in Java. Furthermore, given the absence of a Java library that offers features like `highdicom`, the generation of the DICOM-SR instances had to be re-implemented from scratch in the Java language.

Figure 11 depicts the resulting user interface, which is identical to that of the original Python plugin [7]. However, the deployment of the Java plugin is now straightforward for the end user: It is sufficient to download one single `.jar` archive and to load it using the Java plugin for Orthanc. Thanks to the lightness of Orthanc, this shows that deep learning models can easily be executed on any computer equipped with a Java virtual machine.

4.5 Software Availability

All the contributions of this paper are released as free and open-source software. The source code of the Java plugin for Orthanc and of its sample codes is available under the GPLv3 license, which matches the license of the Orthanc server, at: https://orthanc.uclouvain.be/hg/orthanc-java/. This amounts to more than 10,000 lines of code, as reported by the `cloc` command-line tool, indicating its complexity. Technical documentation and additional examples are available in the Orthanc Book, the official documentation of the Orthanc project. The weights of the deep learning model for the detection of masses on mammograms are available as open data as well. The source code of the Python plugin for Orthanc and its sample codes, which also contains more than 10,000 lines of code, is available under the AGPLv3 license at: https://orthanc.uclouvain.be/hg/orthanc-python/. Extensive technical documentation and examples can be found in the Orthanc Book. Finally, note that the official download site of the

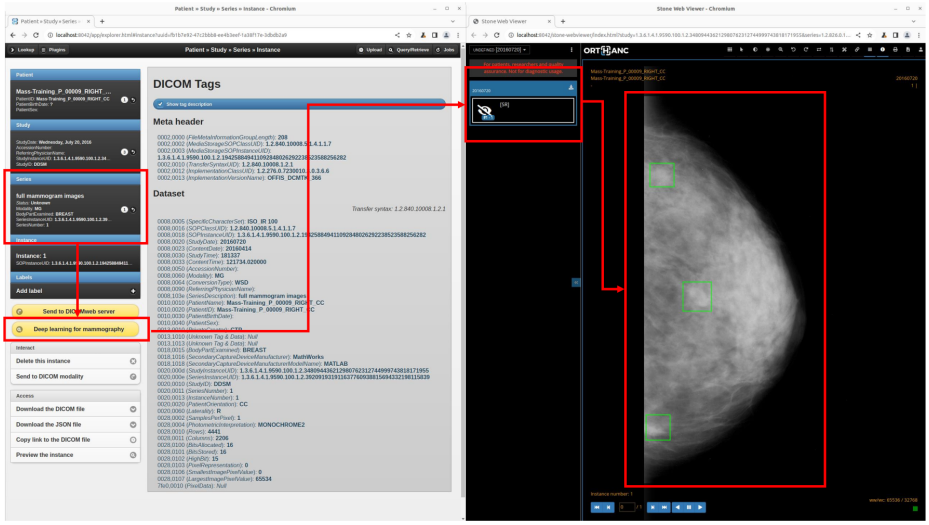


Fig. 11. Mammography inference integrated within the Orthanc ecosystem using the Java plugin. *Left:* The user selects a DICOM instance in the built-in user interface of Orthanc, and executes the inference by clicking on a button. *Right:* This operation generates a standard DICOM-SR instance that can be displayed using the Stone Web viewer. This figure is identical to the one presented in the recent paper that executed inference of the RetinaNet model using the Python plugin [7], as both the Python and Java plugins share equivalent features.

Orthanc project provides precompiled binaries and installers for both the Java plugin and the Python plugin.

5 Conclusions

This paper presents several technical contributions. Firstly, it explains how plugins extending the core features of the Orthanc server can be developed using the Java and Python programming languages. These languages enable users unfamiliar with C++ to create Orthanc plugins implementing specific DICOM pipelines and leveraging popular libraries for medical imaging and image processing. Secondly, a framework to deploy a FHIR server for medical imaging alongside a DICOMweb server is introduced by integrating Orthanc with HAPI through the Java plugin. The resulting architecture is lightweight, easy to deploy, and robust, ensuring the continuous synchronization of FHIR and DICOM resources. Thirdly, this paper demonstrates that the Java plugin for Orthanc represents an attractive solution to deploy pre-trained deep learning models for medical imaging to a broad audience.

All contributions in this paper are released as free and open-source software, in the hope to raise the maturity level of the `ImagingStudy` FHIR resource and

to promote the deployment of open deep learning models for medical imaging that could be used in academic contexts and in emerging economies. Future work includes the development of a pedagogical platform for teaching the principles of modern health interoperability in medical imaging. Thanks to the code model that encapsulates the primitives of the Orthanc plugin SDK, plugins could also be developed for additional programming languages like C#, Go, or Rust. Moreover, the main limitation of the designed FHIR server is that it currently only implements the `ImagingStudy`, `Patient`, and `Endpoint` FHIR resources in a read-only mode. From this perspective, it would be particularly interesting to enrich the FHIR plugin for Orthanc by creating a bridge between DICOM worklists and FHIR using the `Task` and `ServiceRequest` resources. Finally, the value of FHIR for artificial intelligence and for integrating multi-modal data sources including clinical, imaging, and multi-omics data will continue to be explored.

References

1. Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D.G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., Zheng, X.: TensorFlow: A system for Large-Scale machine learning. In: 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16). pp. 265–283. USENIX Association, Savannah, GA (Nov 2016)
2. Ayaz, M., Pasha, M.F., Alzahrani, M.Y., Budiarto, R., Stiawan, D.: The Fast Health Interoperability Resources (FHIR) standard: Systematic literature review of implementations, applications, challenges and opportunities. JMIR Medical Informatics **9**(7), e21929 (Jul 2021). https://doi.org/10.2196/21929
3. Bender, D., Sartipi, K.: HL7 FHIR: An agile and RESTful approach to healthcare information exchange. In: Proceedings of the 26th IEEE International Symposium on Computer-Based Medical Systems. IEEE (Jun 2013). https://doi.org/10.1109/cbms.2013.6627810
4. Benson, T., Grieve, G.: Principles of Health Interoperability. Springer International Publishing (2021). https://doi.org/10.1007/978-3-030-56883-2
5. Boufahja, A., Nichols, S., Pangon, V.: Custom FHIR resources definition of detailed radiation information for dose management systems. In: Pesquita, C., Fred, A.L.N., Gamboa, H. (eds.) Proceedings of the 14th International Joint Conference on Biomedical Engineering Systems and Technologies, BIOSTEC 2021, Volume 5: HEALTHINF, Online Streaming, February 11-13, 2021. pp. 467–474. SCITEPRESS (2021). https://doi.org/10.5220/0010251104670474
6. Bridge, C.P., Gorman, C., Pieper, S., Doyle, S.W., Lennerz, J.K., et al.: Highdicom: A Python library for standardized encoding of image annotations and machine learning model outputs in pathology and radiology. Journal of Digital Imaging **35**(6), 1719–1737 (Aug 2022). https://doi.org/10.1007/s10278-022-00683-y
7. Chatzopoulos, E., Jodogne, S.: Integrated and interoperable platform for detecting masses on mammograms. In: Proceedings of the 34th Medical Informatics Europe Conference (MIE 2024) (Aug 2024)
8. Costa, C., Ferreira, C., Bastião, L., Ribeiro, L., Silva, A., Oliveira, J.L.: Dicoogle: An open-source peer-to-peer PACS. Journal of Digital Imaging **24**(5), 848–856 (Oct 2010). https://doi.org/10.1007/s10278-010-9347-9

9. Eichelberg, M., Riesmeier, J., Wilkens, T., Hewett, A.J., Barth, A., Jensch, P.: Ten years of medical imaging standardization and prototypical implementation: the DICOM standard and the OFFIS DICOM toolkit (DCMTK). In: Ratib, O.M., Huang, H.K. (eds.) SPIE Proceedings. SPIE (Apr 2004). https://doi.org/10.1117/12.534853
10. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: Conference on Computer Vision and Pattern Recognition (CVPR). pp. 770–778. IEEE (Jun 2016). https://doi.org/10.1109/cvpr.2016.90
11. Hussain, M.A., Langer, S.G., Kohli, M.: Learning HL7 FHIR using the HAPI FHIR server and its use in medical imaging with the SIIM dataset. Journal of Digital Imaging **31**(3), 334–340 (May 2018). https://doi.org/10.1007/s10278-018-0090-y
12. Jodogne, S.: The Orthanc ecosystem for medical imaging. Journal of Digital Imaging **31**(3), 341–352 (May 2018). https://doi.org/10.1007/s10278-018-0082-y
13. Jodogne, S.: On the use of WebAssembly for rendering and segmenting medical images. Comms. in Computer and Information Science **1814**(1), 393–414 (2023). https://doi.org/10.1007/978-3-031-38854-5_20
14. Jodogne, S.: Setting a PACS on FHIR. In: Proceedings of the 17th International Joint Conference on Biomedical Engineering Systems and Technologies (HEALTHINF). vol. 2, pp. 123–131. SCITEPRESS (2024). https://doi.org/10.5220/0012384600003657
15. Ketkar, N.: Introduction to PyTorch. In: Deep Learning with Python, pp. 195–208. Apress (2017). https://doi.org/10.1007/978-1-4842-2766-4_12
16. Lattner, C., Adve, V.: LLVM: A compilation framework for lifelong program analysis & transformation. In: Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04). Palo Alto, California (Mar 2004)
17. Lee, R., Gimenez, F., Hoogi, A., Miyake, K., Gorovoy, M., Rubin, D.: A curated mammography data set for use in computer-aided detection and diagnosis research. Scientific Data **4**(1) (Dec 2017). https://doi.org/10.1038/sdata.2017.177
18. Lin, T., Goyal, P., Girshick, R., He, K., Dollar, P.: Focal loss for dense object detection. In: Proc. of the 2017 IEEE International Conference on Computer Vision (ICCV). pp. 2999–3007. IEEE Computer Society, Los Alamitos, CA, USA (Oct 2017). https://doi.org/10.1109/ICCV.2017.324
19. Mason, D.: SU-e-t-33: Pydicom: An open source DICOM library. Medical Physics **38**(6Part10), 3493–3493 (Jun 2011). https://doi.org/10.1118/1.3611983
20. McCormick, M., Liu, X., Jomier, J., Marion, C., Ibanez, L.: ITK: Enabling reproducible research and open science. Frontiers in Neuroinformatics **8** (2014). https://doi.org/10.3389/fninf.2014.00013
21. NEMA: National Electrical Manufacturers Association PS3 / ISO 12052, Digital Imaging and Communications in Medicine (DICOM) standard (2024), http://www.dicomstandard.org/
22. Panchal, A., Keyes, R.: SU-GG-t-260: Dicompyler: An open source radiation therapy research platform with a plugin architecture. Medical Physics **37**(6Part19), 3245–3245 (Jun 2010). https://doi.org/10.1118/1.3468652
23. Rosset, A., Spadola, L., Ratib, O.: OsiriX: An open-source software for navigating in multidimensional DICOM images. Journal of Digital Imaging **17**(3), 205–216 (Jun 2004). https://doi.org/10.1007/s10278-004-1014-6
24. Schroeder, W., Martin, K., Lorensen, B.: The Visualization Toolkit (4th ed.). Kitware, New York (2006)
25. Shackleford, J., Kandasamy, N., Sharp, G.: Plastimatch: An open-source software for radiotherapy imaging. In: High Performance Deformable Image Registration

Algorithms for Manycore Processors, pp. 107–114. Elsevier (2013). https://doi.org/10.1016/b978-0-12-407741-6.00006-2
26. Shi, W., Giuste, F.O., Zhu, Y., Carpenter, A.M., Iwinski, H.J., Hilton, C., Wattenbarger, J.M., Wang, M.D.: A FHIR-compliant application for multi-site and multi-modality pediatric scoliosis patient rehabilitation. In: Huang, Y., Kurgan, L.A., Luo, F., Hu, X., Chen, Y., Dougherty, E.R., Kloczkowski, A., Li, Y. (eds.) IEEE International Conference on Bioinformatics and Biomedicine, BIBM 2021, Houston, TX, USA, December 9-12, 2021. pp. 1524–1527. IEEE (2021). https://doi.org/10.1109/BIBM52615.2021.9669649
27. Tang, S.T., Tjia, V., Noga, T., Febri, J., Lien, C.Y., Chu, W.C., Chen, C.Y., Hsiao, C.H.: Creating a medical imaging workflow based on FHIR, DICOMweb, and SVG. Journal of Digital Imaging **36**(3), 794–803 (Feb 2023). https://doi.org/10.1007/s10278-021-00522-6
28. Warnock, M., Toland, C., Evans, D., Wallace, B., Nagy, P.: Benefits of using the DCM4CHE DICOM archive. Journal of Digital Imaging **20**(S1), 125–129 (Oct 2007). https://doi.org/10.1007/s10278-007-9064-1
29. Xu, J., Thevenon, G., Chabat, T., McCormick, M., Li, F., Birdsong, T., Martin, K., Lee, Y., Aylward, S.: Interactive, in-browser cinematic volume rendering of medical images. Computer Methods in Biomechanics and Biomedical Engineering: Imaging & Visualization **11**(4), 1019–1026 (Nov 2022). https://doi.org/10.1080/21681163.2022.2145239
30. Ziegler, E., Urban, T., Brown, D., Petts, J., Pieper, S.D., Lewis, R., Hafey, C., Harris, G.J.: Open Health Imaging Foundation viewer: An extensible open-source framework for building Web-based imaging applications to support cancer research. JCO Clinical Cancer Informatics **4**, 336–345 (Nov 2020). https://doi.org/10.1200/cci.19.00131