Patient Monitoring Service (PMS)

Part 3

Architectural extension

Ngo

Academic Year 2021 – 2022

H09B5B: Software Architectuur
H07Z9B: Software Architecture

Triet Ngo (r0869104)

Contents

1. Introduction

In this extension assignment, we extend the initial PMS software architecture which is driven by 4 (out of 5) scenarios. As a result we add the following adjustments/additions diagram to the initial architecture:

- Sequence diagram for processing sensor data in gateway D.7.
- Sequence diagram for processing incoming data in PMS back-end D.8.
- Sequence diagram for the security checking procedure D.9.
- Sequence diagram for patient login D.10.
- Sequence diagram for the procedure of registering a new patient D.11.
- Sequence diagram for the procedure of requesting a on-demand consultation D.12.
- Sequence diagram for sending notification to relevant users D.13.
- Sequence diagram for updating user information such as risk level D.14.
- Decomposition for component `GatewayApplication` B.2.
- Decomposition for component `PMSRequestProcessor` B.5.
- Decomposition for component `PMSSecurityManager` B.3.
- Updating the deployment diagrams C.2 C.3 C.4.

In the next section we will discuss the design as well as the Rationales of our decision then end the report with a discussion of the process as well as the scenarios. The QAS we decided to present are presented below and are dicussed in the same order:

- Sec1: Patient identification.
- Av2: Communication between patient gateways and pms backend.
- P1: Data exchange with physicians.
- Av1: Internal PMS database failure.

2. QAS Decisions

2.1 Sec1: Patient identification

Key decisions

- A component that is responsible for the security operations of the PMS back-end as well as gateway application `PMSSecurityManager`.
- A `UserStorage` contains abstract user information such as patient Id or user Id as well as their hashed password, registered devices, sensors .
- Decompose the `PMSSecurityManager` into modules that carry different responsibilities in order to make the component modular and have the ability to exchange the implementation of the module easily e.g. delegate some of the security tasks to third parties, etc. The modules are:
  - `SecurityRequestProcessor`: provide `SecurityAPI` contains high-level security operations for security task such as authenticate user, request, get public key, etc. The operations in this module utilize the modules in the component.
  - `SecurityLogger`: maintain a logging scheme for the component.
  - `IntegrityManager`: Managing operations that ensure the integrity of the system.
  - `AuthManager`: Managing operations that ensure the authentication of the system.
  - `AccessControlManager`: Managing operations that ensure the access control management of the system.
  - `HISComManager`: Managing to retrieve information from HIS (e.g. confirming the existence of a patient).
- Using RSA with 512 bytes key size for signing and encrypting/decrypting data packages, requests as well as signing a signed user summary as access token.
- Using symmetric encryption scheme (ES-256) to transmit sensor data package confidentially.

Employed tactics and patterns: Verify message integrity, Detect message delay, Authenticate actors, Authorize actors, Encrypt data.

Rationale

The patient registration procedure ensures that the PMS back-end has the necessary information to identify the patient gateway and the different sensors associated with the newly-registered patient's account To illustrate the registration process we provide the diagram `RegisterPatient`. To ensure that the registration process ensure the following constraints: The user register the new patient has the necessary access control, the patient exists according to HIS, the password strength has to be sufficient (User passwords are enforced to be minimally 8 characters and maximally 64 characters long, and should not be commonly used passwords), we design modular modules such as `AuthManager` to ensure the password strength and register user to storage, `AccessControlManager` to verify the access control of the nurse who registered the patient, `HISComManager` to verify the existence of the patient, `SecurityLogger` to log the process to storage. Hence the registration process ensures both the authenticity and access control and also makes sure to include the user's device id as well as its sensor to the User JSON which is stored in UserStorage. The benefit of having a JSON type instance is that it is modular and flexible. Since it does not require rigid datatype like Datatype and can be easily updated or queried using the keys in JSON.

Interaction between the patient gateways and the back-end system is subject to authentication, i.e. each patient gateway is authenticated before any further operations such as providing new sensor data readings or retrieving patient information are allowed every requests from gateway to PMS is managed by `GatewayComChannel` and the operations required the check from `PMSSecurityManager`. One example would be the process of sending data (or emergency notification which is managed by the emergency flag in the function sendSensorData) as illustrated in diagram IncomingSensorData. GatewayComChannel immediately call for decryptAndVerifyPackage to check for both authenticity (access token), integrity (signature), access control. If the process succeeds then PMSSecurityManager will return the decrypted **SensorDataPackage** for further processing.

Access to the PMS app is subject to authentication, i.e. a patient has to provide a username and password before further app functionality can be used as mentioned in the previous point, if the security check fails then the Gateway will receive an exception and automatically log out the user and require login again. Therefore, it is safe to say that gateway will ensure that the user has to go through the authentication process by providing a username and password so that the gateway receives and store the correct access token for the session.

Patient information is subject to access control all users' access control is manage by AccessControlManager. The access control is divided into scopes that represent roles or actions. For example, to register new staff the user has to have the scope "register_new_staff" in their access control. Also, this module ensures that the data they receive in their requirements fall into their scopes like their own record (patient) or the records of patients that are assigned to the cardiologist.

The confidentiality and integrity for all information transmitted between the patient's app and the PMS back-end is actively preserved the **SensorDataPackage** that are sent to the PMS back-end are encrypted and signed (incl. access token in request) to prevent tampering during sending. This ensures that the package that the PMS back-end received is either the one that is sent or PMSSecurityManager will detect that there the package has been tampered with. Also the sensor package will be encrypted with AES-256 algorithm so the confidentiality is ensure.

Each attempt to interact with the back-end system is logged all operations that are run by PMSSecurityManager is subjected to be logged by SecurityLogger. We stored the logs in UserStorage which is the same storage as user information. The logs are in JSON format and contain the following information:

- the time and date of the request,
- identification of the requesting patient gateway and/or patient (before actual authentication, these are the IP addresses),
- identification of the requested operation(s), and
- whether the request was granted.

Considered Alternatives

Third-party security services Nowadays there are many services that provide security management, one primary example would be Firebase. The advantage of this method is that Firebase can provide all the necessary services for our application (from authentication, managing users, etc.) but it comes with a cost. The cost of using Firebase with high-frequency data among lots of users can lead to degradation in performance and also introduce latency if the Firebase server is far away. That leads to the next alternative/improvement.

Parallel PMSSecurityManager Since the operations of security are independent of one another, it is also possible to increase performance by deploying scalable processors. There will be a monitor to monitor the flow of requests and adjust the number of PMSSecurityManager according to the load. This also has a trade-off between cost and performance which will be discussed in a Performance requirement.

Asymmetric key for encryption We can decide between a symmetric key or public/private key for message encryption. The advantages of public/private key is that it provide a scheme so that all user can encrypt the message but none can decrypt them while the symmetric key can be exploit so that the user can use it to decrypt data. But symmetric keys are usually included with easier encryption scheme that can have small key size as well as able to encrypt large data, it also has a feature that make the encrypted version smaller than the decrypted one which can improve the performance of the application. The most widely used symmetric algorithm is AES-128, AES-192, and AES-256.

Further responses in case of security breach In the requirement, there is no mention of actions that are required when there is a security breach. One improvement would be to either block the access of the user for some time after a number of failed attempts or to contact the security administrator for further actions.

Sensitivity points and trade-offs

In the below section where we discuss the performance requirement of PMS, we choose the preserve the heavy scheme of the Security design decision such as checking all the security criteria of the requests (confidentiality, integrity, access control, authentication) rather than relieve a few constraints exchange for a boost in performance. The important of Security out-weights the performance gain by relieving it.

2.2 Av2: Communication between patient gateways and pms back-end

Key decisions

- Introduce status monitor for Gateways and PMS communication channels: `StatusMonitor` and `PMSComHealthMonitor`.
- Deploy `GatewayFaultDectector` for monitoring the availability of gateways.
- Introduce `GatewayStorage` for temporary storing sensor data when the gateway in degraded mode.

Employed tactics and patterns: Ping/echo, Heartbeat, Monitor, Rollback, Passive Redundancy, State Resynchronization.

Rationale

Detection

The PMS back-end services should be able to autonomously detect any failures based on the lack of sensor data updates from one or more patient gateways. We introduce a component `GatewayComChannel` for Gateways to call and send data to the PMS back-end. The data sent by Gateways are stored in `UserStorage`. In this component, data are stored in a minimal way with time-stamp as the time the patient sent sensor data. We also have a component `GatewayFaultDectector` which monitors the storage changes and checks for the lack of updates from patient gateways periodically. In case of lack of data, the component use interface `PMSComHeartBeat` to report exceptions to `PMSComHealthMonitor`.

The pms back-end services should be able to autonomously detect any failures of the relevant internal communication subsystems. As part of the above, the component `PMSComHealthMonitor` has the `PMSComHeartBeat` so that the components in the `PMSComNode` can call to report its status/exception to the authority (system administrator). The advantages of this design using `PMSComHeartBeat` tactic is that we can monitor faults and exceptions using only 1 component making it simple yet feasible. But this is also the down-side, since we overload the `PMSComHeartBeat` with the responsibility of fault-detection for the whole communication system, if its processing power is low then there can be overhead throughout the system.

The patient gateway app should be able autonomously detect any communication failures and goes into degraded mode, which involves temporarily storing sensor data and notifications for later synchronization, systematic retrying to complete the communications, and use of possible back-up communication channels. In `GatewayApplication`, we introduce module `StatusMonitor` implement ping/echo to proactively monitor the services with the pre-defined frequency for each (for example, monitor `PMSComChannel` more frequently than `HardwareMgmt`). Also the ping/echo happens hierarchically (`DataSync` will be responsible for ping `GatewayStorageMgmt` to monitor its health. In degraded mode (which switched by `StatusMonitor` to `DataSync`) informing `DataSync` that from now on temporarily store the data from `WearableDataCapturing` to `GatewayStorage` instead of forwarding it to `PMSComChannel` to send to PMS back-end. We also have a `BackupComChannel` which will use SMS or emergency notification using DeviceAPI (API that have access to the device hardware, since we also need to specify component for hardware and hanging interface is unacceptable for the plugin so we omit this interface).

The PMS keeps track of how long there was a lack of communication. Since `GatewayFaultDectector` has access to how long a patient's gateways has been disconnected, we store log file of these faults in `UserStorage`.

Prevention

The app on the patient's gateway warns the patient in time when the battery of their patient gateway unit is running low. The component `StatusMonitor` in `GatewayApplication` also has access to DeviceAPI (which provides the necessary device hardware monitor API so that the component can periodically call and monitor the battery and inform the patient via notification of the battery. The same argument is applied for **The app on the patient's gateway warns the patient when they have no or limited network connectivity**.

Resolution

As discussed, we have a component `GatewayFaultDectector` that detects faults and reports to the authorities. In addition, we also store the subsystems state/config periodically to `SubsysStateStorage` so that in case of fault, the authorities can either decide to redeploy the subsystem or revert it to the previous working state by fetching the state/config from the storage.

`PMSComChannel` implements exponential back-off to send sensor reading to PMS. Also as mentioned before, in degraded mode, the data will be temporary store in `GatewayStorage`

Considered Alternatives

Hierarchical monitoring As mentioned before, since we overload the component `PMSComHealthMonitor` which is responsible for monitor the status of the whole system which can create overheads that decrease the performance of the system. We can introduce a hierarchical Monitoring system such as splitting the responsibility to com channels, storage, etc and then they will report to a central channel which makes the data processing decrease. On the other hand, if the component is not overloaded then we create lots of running processes which waste resources as it runs.

Employ replication for even more availability In all the cases as above, especially regarding storage, we used a passive scheme which involves administration when faults occur but we can employ active/pass replication for (introduce some redundancy) to the system. We can deploy instances of a component based on their load. For example, 3 instances for `GatewayComChannel`, etc. So that in case of failure, the system that fails will automatically switch to the instance that is not failed (passive replication). But this approach introduces a heavy cost on resources.

Sensitivity points and trade-offs

2.3 P1: Data exchange with physicians

Key decisions

- Deploys multiple PMSRequestProcessor and communication channels to increase performance.
- Having a Request RequestScheduler with a polling scheme so that processor and actively poll for operations that need heavy computation. Also this component manage the priority of the request.
- Caching shared storage with SensorDataCache.
- Separating the communication channels between HISComChannel and GatewayComChannel.

Employed tactics and patterns: Prioritize events, Reduce overhead, Increase resources, Schedule resources.

Rationale

The system is able to process all changes to a patient's risk level in a timely fashion/The system is able to process all requests for patient information in a timely fashion We increase the number of PMSRequestProcessor so that the PMSRequestProcessor will poll scheduled operations in the priority queue. Since the scheme is polling we reduce the overhead of the RequestScheduler pushing and managing the PMSRequestProcessor. But not all operation request scheduling since there are some small weight operations that can be processed directly by calling the PMSRequestProcessor directly from the communication channel. An example of this distinction can be found in diagram D.14 where the getting available risk level is simple (only 3 every time) so that no need to increase the overhead with RequestScheduler while the updating risk level require accessing to the user storage and save it so it require the need of RequestScheduler in case a large number of the same requests coming simultaneously. This approach however seems to be having some similarity with indirect communication which should not be used in this scheme due to its nature of lacking immediate feedback. To preserve the polling characteristic while having immediate feedback (reply), we maintains threads (physical and virtual) to support the RequestScheduler. When a request comes in to the RequestScheduler, it opens a light-weight threads that has only one purpose of holding/waiting for the result from the PMSRequestProcessor for the result of the processed request comes back. Since the RequestScheduler is only to distribute the requests, the strains on the threads should not be large enough to be posed as an overhead to the performance of the system.

Any risk assessment triggered by an on-demand consultation or risk assessment configuration is initiated within a short time frame of the request. In the case of remote on-demand consultations, the results of the risk assessment are provided to the physician in a timely fashion According to the process of on-demand consultation in diagram D.12, this type of operation requires scheduling so that other operations would not get in the way of its process so that the process can happen in a timely fashion. Also the scheduler acts as a priority queue where it would assign a different priority to requests based on the patient risk level.

Operations related to notification is dealt with a certain level of performance For the current system, notifications are dealt with as same as other heavy-weight operation which involves the RequestScheduler which also mean that the notification of high-risk patients is prioritized to be sent over the one with lower priority in batch then the batches are sent sequentially. However this approach may become a problem if batches with different priorities come to the queue while the other is being sent since PMSRequestProcessor is used for notification sending should be limited due to the nature of centralised processor (the processors also have to process other jobs as well). So in extreme cases where low priority batches are being sent by the processors and the new high-risk batches come in. One solution for this problem is discuss in the below section.

Considered Alternatives

Concurrent notification sending when there is numerous notification to be sent For our current design, we assign a batch of notification of a patient and send them to all interested party but in case there is a lot of notification to be sent. We can design a Notification Sending Queue where it ensures two criteria: the queue is a priority queue based on the color of the notification, and the queue allows concurrent fetching from communication channels. This way the process of sending notifications will be a separate process that has its own queue. However this approach will introduce new processes which also consume resources so this approach should not be deployed at the trials of the system but only deploy after necessary data is collected after observing the system for some time.

Pub-sub scheme for sending notification In our design, we sent the notification in batches which have the potential to become an overhead as explained above. One solution for this problem is to separate the notification sending system from the main processors. We introduce the Notification Publisher and a few Notification Subscriber to the system where the `PMSRequestProcessor` can send the notification request order to it and the worker/subscriber can use either polling or pushing scheme to get the orders from the Publisher. The Publisher should also maintain a priority queue bases solely on the risk level of the notification. Due to the nature of indirect communication, the process of sending notification would not occupy the processors or hinder other requests. Therefore instead of sending requests in batches, the queue receives batch of notification and add them to the queue so that the requests are sending one by one based on its order which solves the problem of sending notification on batches mentioned above.

Balance between strength and number with records In our design, we deploy multi-instances of the components that we think more resources to ensure the predefined performance agreement but the process of ensuring the reliability of the multi-instances also requires other components such as scheduler or monitor. For an alternative, we can experiment with different configurations and maintain records of the results from this experiment so that we can adjust the computation power of the component and the number of instances or whether to drop multi-instances according to the observations of the systems after a few months.

Sensitivity points and trade-offs

2.4 Av1: Internal PMS database failure

Key decisions

- Splitting the database into 3 shard horizontally based on their level of risk as shown in node SensorDataStorageNode in diagram C.2.
- Implementing data replication (passive replication) so that we can improve the availability of the database. The number of replications based on the level of risk: 1 replication for low, 2 for medium and 3 for high as shown in node SensorDataStorageReplicaNode in diagram C.2.
- In rare case (when the replication fail), we have component DegradedSensorDataStorage so that it can temporary store data.
- Introducing SensorDataCache to compensate for the decrease in performance (the request exchange overheads) when using passive replication on 3 different database shard.

Employed tactics and patterns: Database sharding, Data replication.

Rationale

Since there are 3 levels of availability for the database, we implement sharding on the database so that we can divide the database into 3 shards based on the level of risk and have a SensorDataDistributer to balance the load as well as find where to find the necessary data. Since we have a separation of data we can have different configurations for the patients. For example, to have SensorDataDistributer maintain a priority queue for querying or have different hardware or replication scheme for the shards. The SensorDataDistributer is also responsible for monitoring the status of the database in case of crashes so that it can replace the database with its replication or in case there is no replication then switch the shard to degraded mode and store the temporary data in DegradedSensorDataStorage. This component will also inform the authority in case of fault. Also, introduce Exceptions appropriately in case of fault.

We also introduce data replication for the shards based on their availability level (1 replication for low, 2 for medium, and 3 for high). The replication employed a passive replication scheme: read operations are called on the secondary replication while the writes are implemented on the primary one and then distributed to the primary. The advantage of this method is that the database will have high availability since the replication will act as the primary one in case of an emergency. The downside of this method is that the overhead of the replication scheme so that it works perfectly is a bit large. Therefore, to compensate for this we can introduce SensorDataCache. The cache will temporarily store recent requests to reduce the need of retrieving one.

We also introduce a stand-by SensorDataDistributer to take over when a crash happens on the SensorDataDistributer and the stand-by replace the primary one. Based on the replication scheme and also the support of DegradedSensorDataStorage. A crash does not lead to any loss of already stored (raw) sensor data readings, to data inconsistency or lack of integrity.

Considered Alternatives

Implementation of caching queried SensorDataPackage While we introduce caching to boost up the overhead, we will not implement the details in the design. In the future if this become an issue then we can implement it.

Geographic redundancy We can also implement Geographic redundancy. The idea is to have multiple servers are deployed at geographical distinct sites. The locations should be globally distributed and not localized in a specific area. It is crucial to run independent application stacks in each of the locations, so that in case there is a failure in one location, the other can continue running. Ideally, these locations should be completely independent of each other.

Predictive Model If we plan to use the service for a long time, we can also collect data and use Predictive Model to keep failure or resource consumption data that can be used to isolate problems and analyze trends. This data can only be gathered through continuous monitoring of operational workload.

A recovery help desk can be put in place to gather problem information, establish problem history, and begin immediate problem resolutions. A recovery plan should not only be well documented but also tested regularly to ensure its practicality when dealing with unplanned interrupts. Staff training on availability engineering will improve their skills in designing, deploying, and maintaining high availability architectures. Security policies should also be put in place to curb incidences of system outages due to security breaches.

Sensitivity points and trade-offs

In this section we introduced passive replication which has a very heavy time complexity and it will decrease the performance of the system. On the other hand, we can argue that the down-time of the database would cause a larger performance loss for the PMS system than the needed time for implementing passive replication. Furthermore, we also have caching and sharding which provide some improvement in the performance aspect of the system.

3. Team approach and time spent

We originally worked in a team of three for this assignment but due to the conflicts arose in the last two assignments, we decided to work on this project individually in part 3. In total we spent roughly 50 hours on the this assignment which are distributed as below:

- Approximately 15 hours for brainstorming the approach and solutions for the 4 scenarios in addition to reading the supported materials.
- Approximately 25 hours for modelling and adding adjustments after a few iterations of the solutions.
- Approximately 10 hours for writing the report.

4. Discussion

After the architecture decisions we made above, our system ensures the following key points:

- The communication among HIS, PMS and Gateways ensures security constraints (confidentiality, integrity, authentication, access control).
- The key components of the system have high availability: `ShardedSensorStorage`, `HISComChannel`, `GatewayComChannel`, etc.
- Have good performance overall by having multi-instances of processor and scheduler.

However there are a few weak points that have not been solved by the design:

- The high workload and potential failure of `UserStorage` where the information of the user (from their security info, to their personal information and their notification subscription etc.).
- There are still improvements that we discussed in the alternatives of the QAS.

A. Client-Server View

Figures

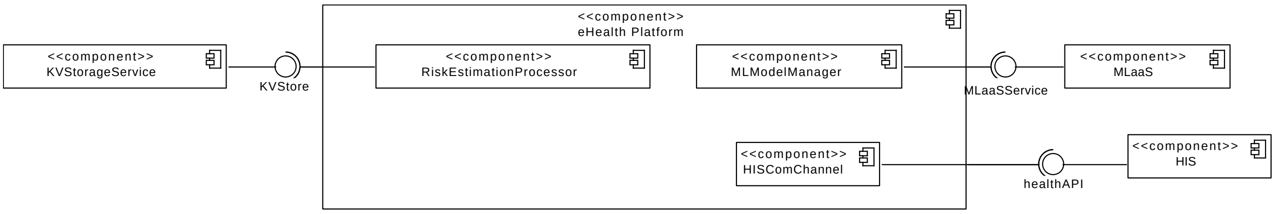

Figure A.1: Context diagram for the client-server view

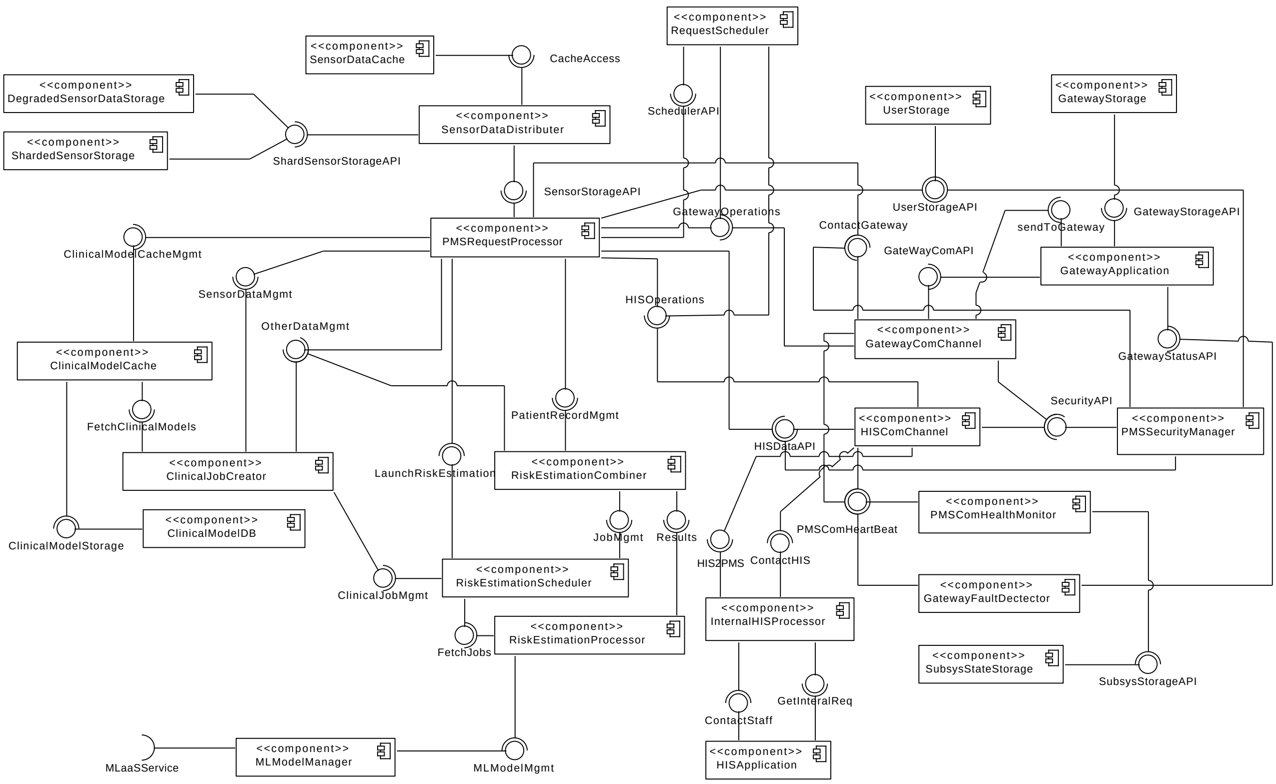

Figure A.2: Primary diagram of the client-server view

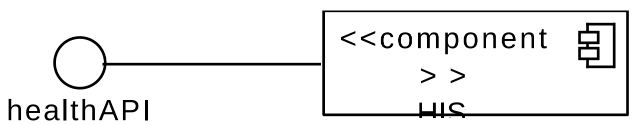

Figure A.3: Component Diagram1

B. Decomposition View

Figures

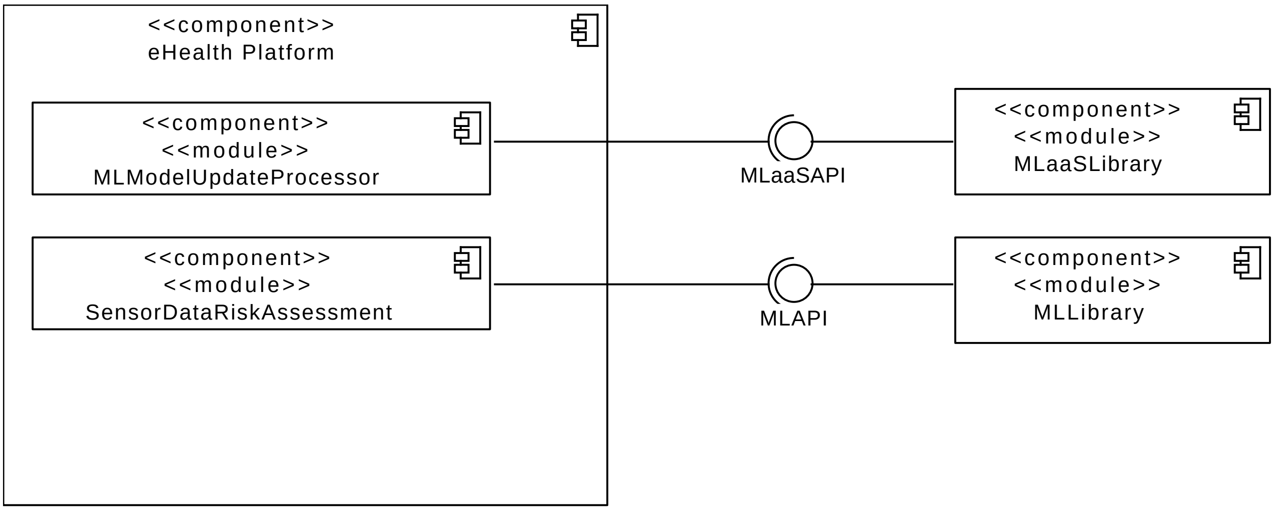

Figure B.1: Context diagram for the decomposition view

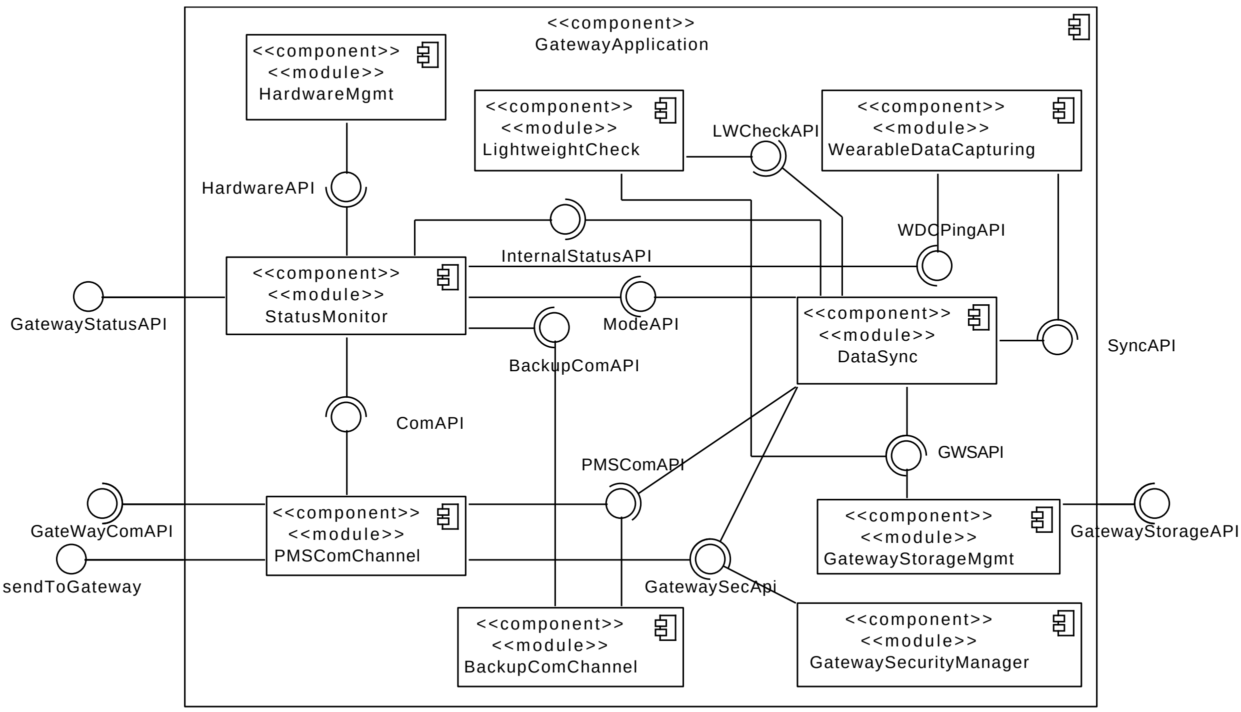

Figure B.2: Decomposition for the GatewayApplication

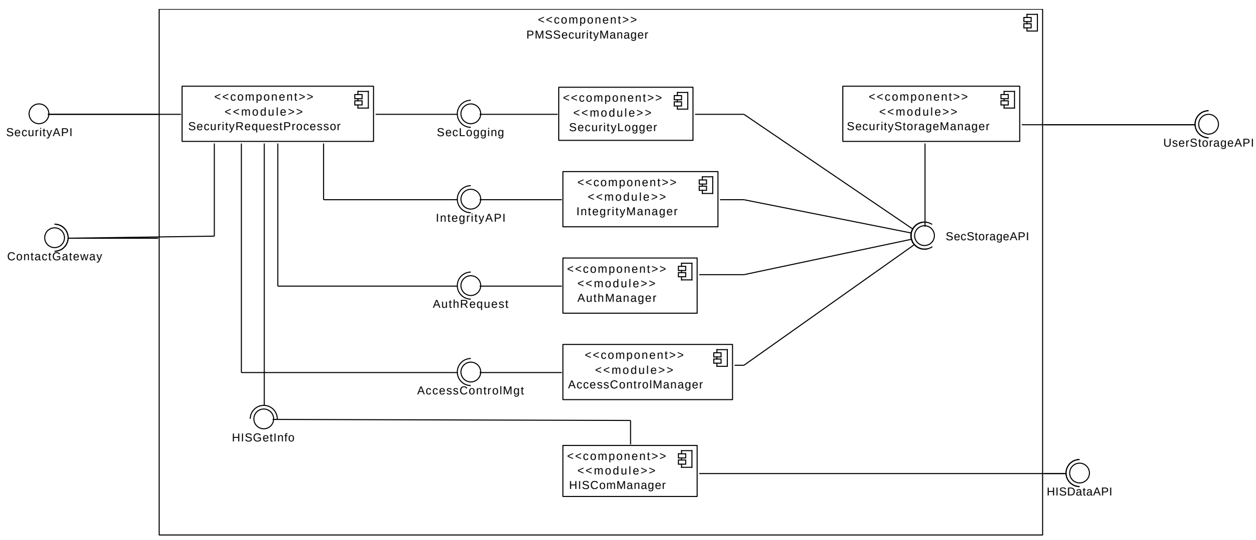

Figure B.3: Decomposition of SecurityManager

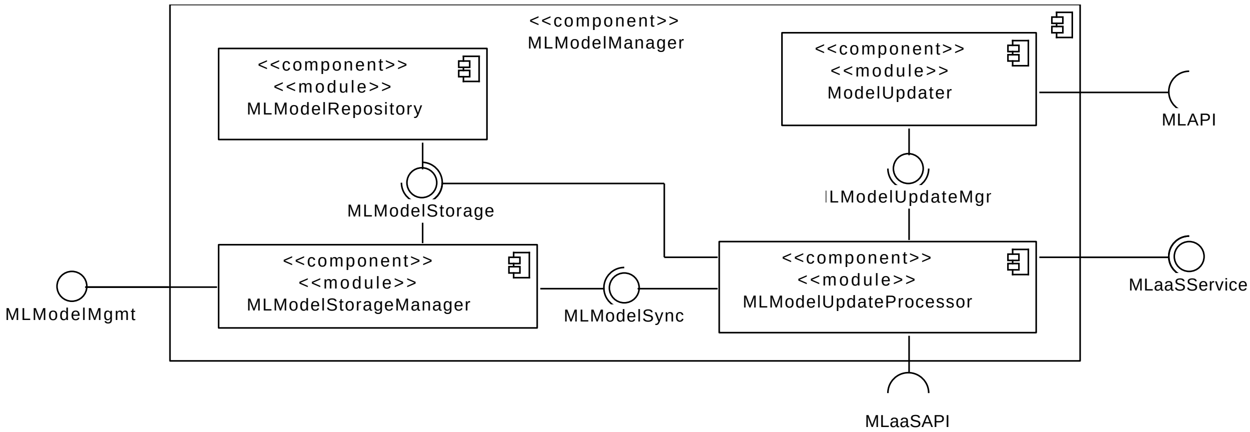

Figure B.4: Decomposition of the MLModelManager

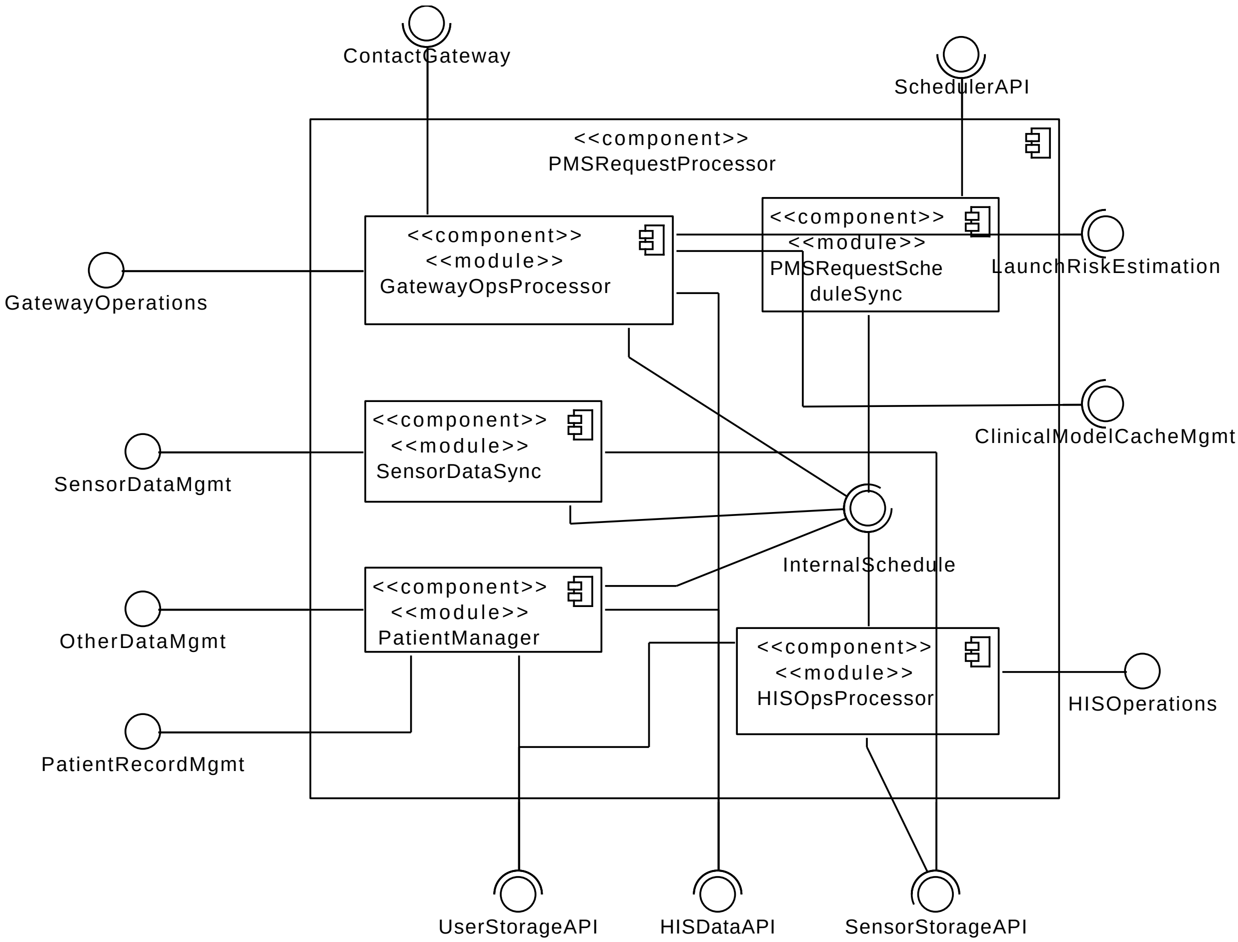

Figure B.5: Decomposition of the PMSRequestProcessor

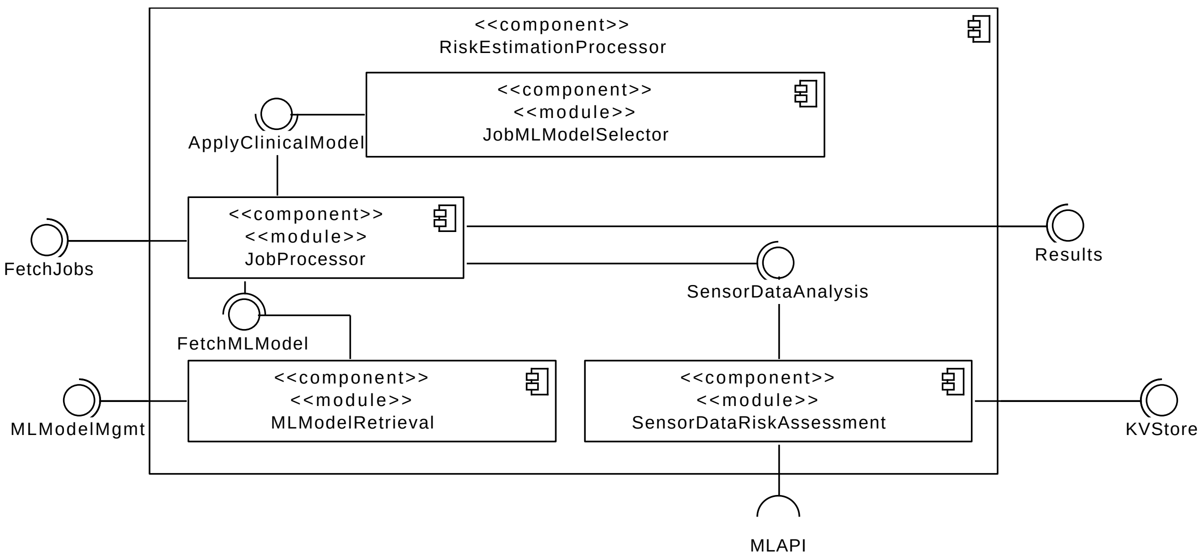

Figure B.6: Decomposition of the RiskEstimationProcessor

C. Deployment view

Figures

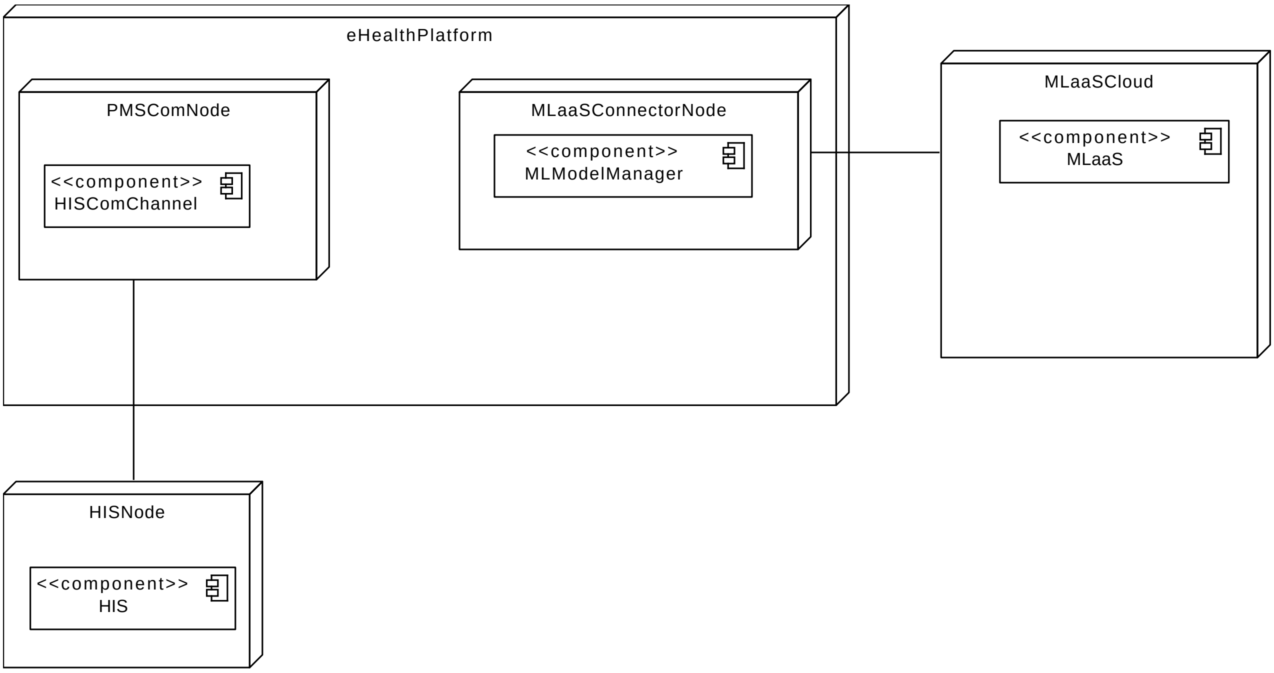

Figure C.1: Context diagram for the deployment view

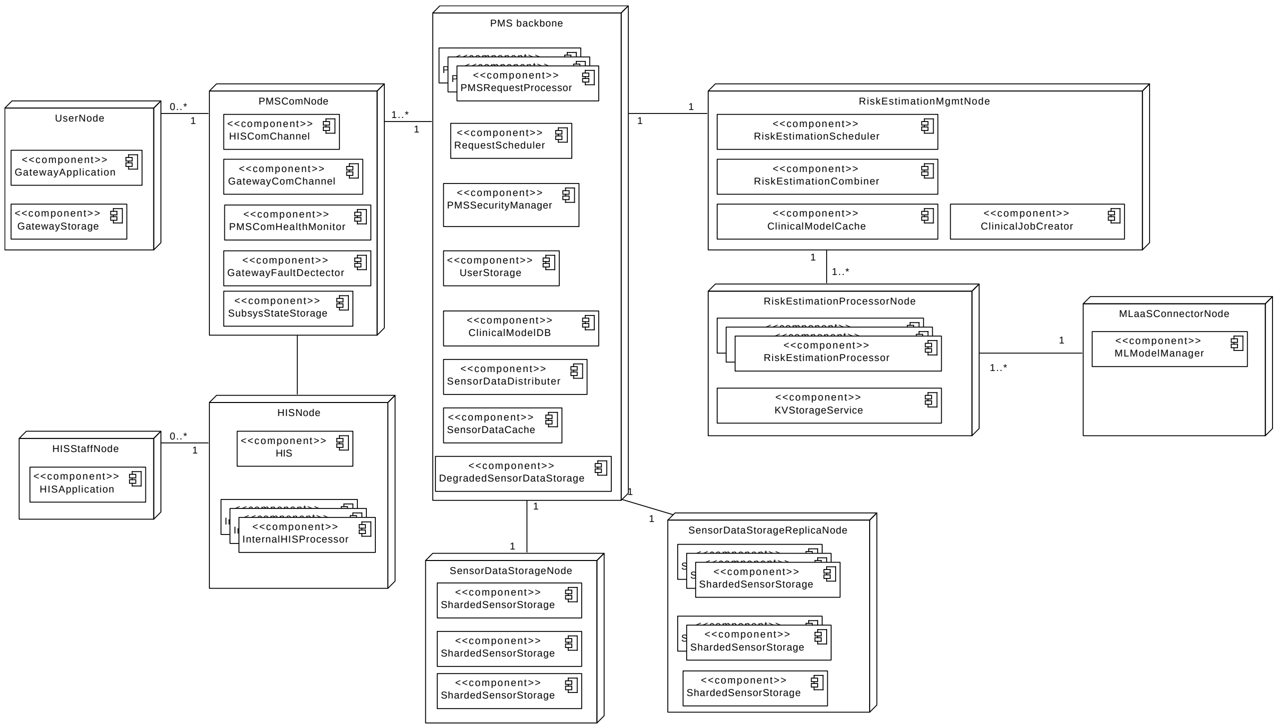

Figure C.2: Primary deployment diagram.

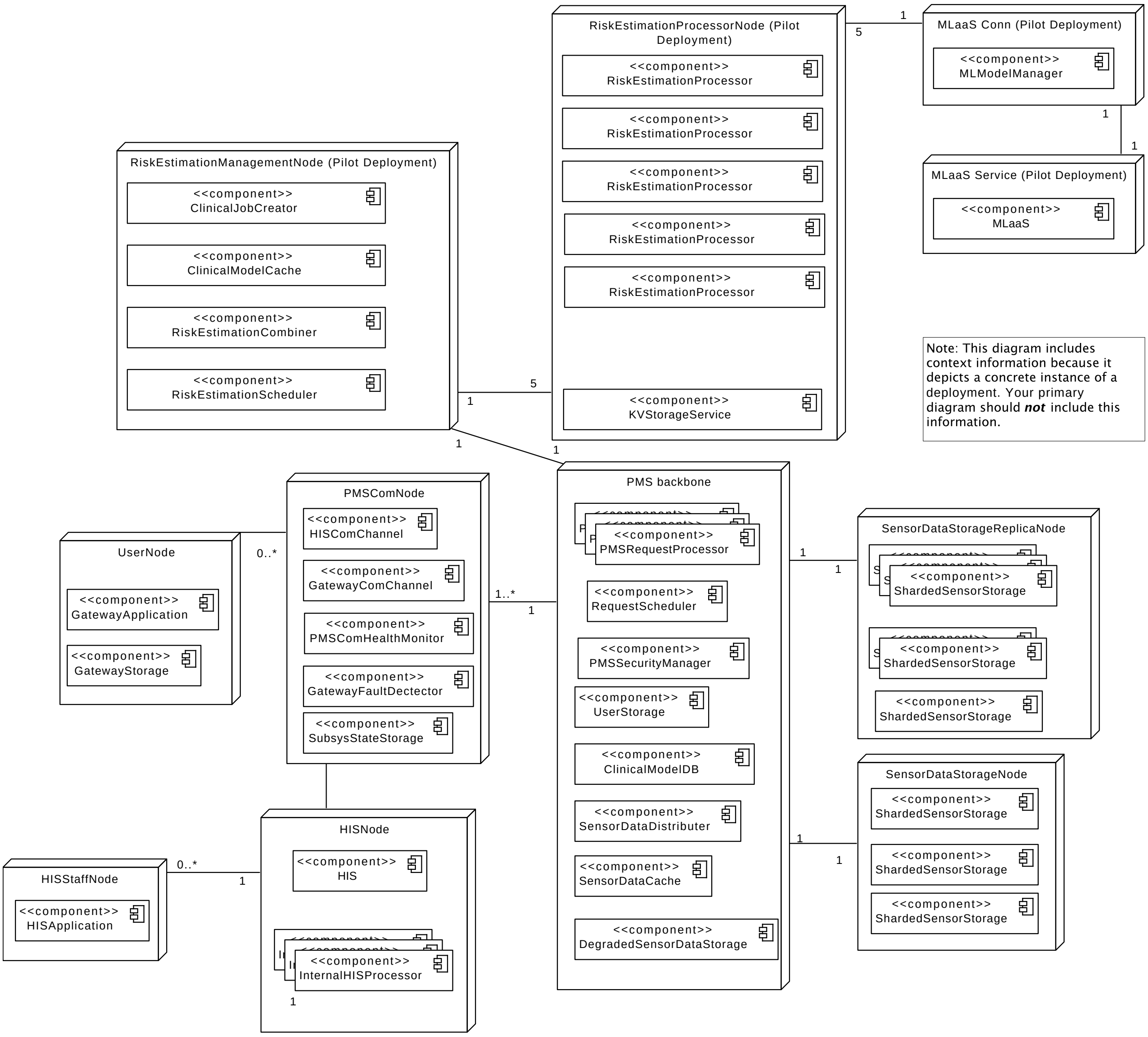

Figure C.3: Pilot Deployment (200 patients, 5 clin models)
This diagram shows a concrete initial pilot deployment for 200 patients with an average of 5 clinical models per patient. This diagram is a concrete instance of a deployment for the specified number of patients and clinical models, therefore it also includes information from the context diagram. Note that it is not correct to include this type of information in your primary deployment diagram.

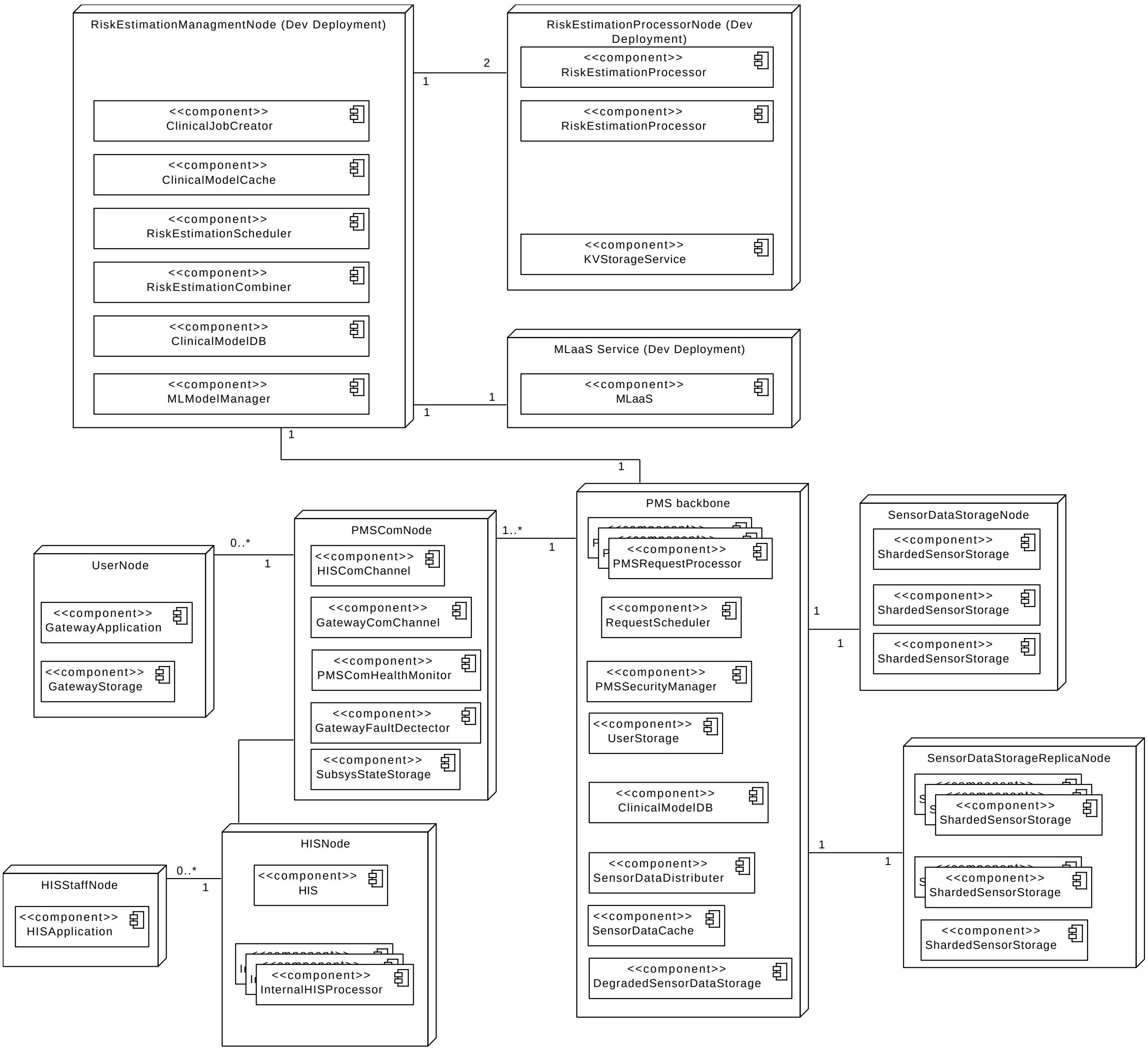

Figure C.4: Development Test Deployment (20 patients, 3 clin models)
This diagram shows a simplified deployment for local development. This diagram is a concrete instance of a deployment for the specified number of patients and clinical models, therefore it also includes information from the context diagram. Note that it is not correct to include this type of information in your primary deployment diagram.

D. Process View

Figures

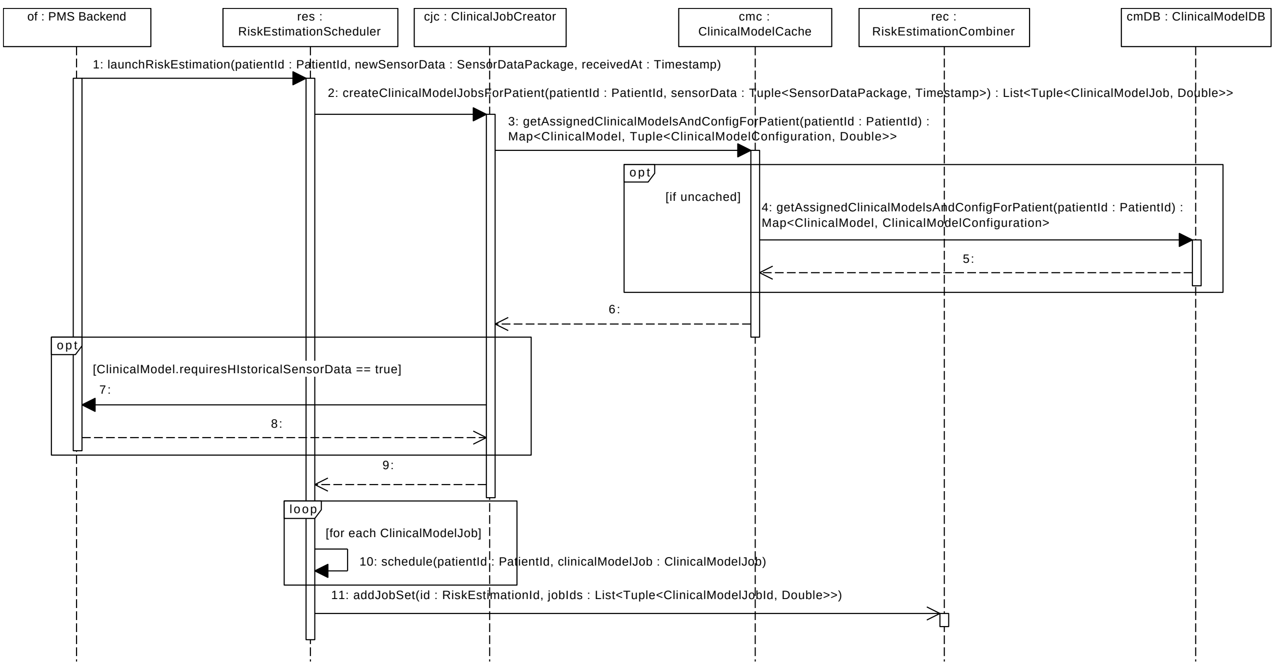

Figure D.1: Risk estimation process
The scheduling and execution of a risk estimation triggered by the arrival of new sensor data. Note that if one or more clinical models in the `ClinicalModelCache` have been invalidated, all relevant clinical models are renewed.

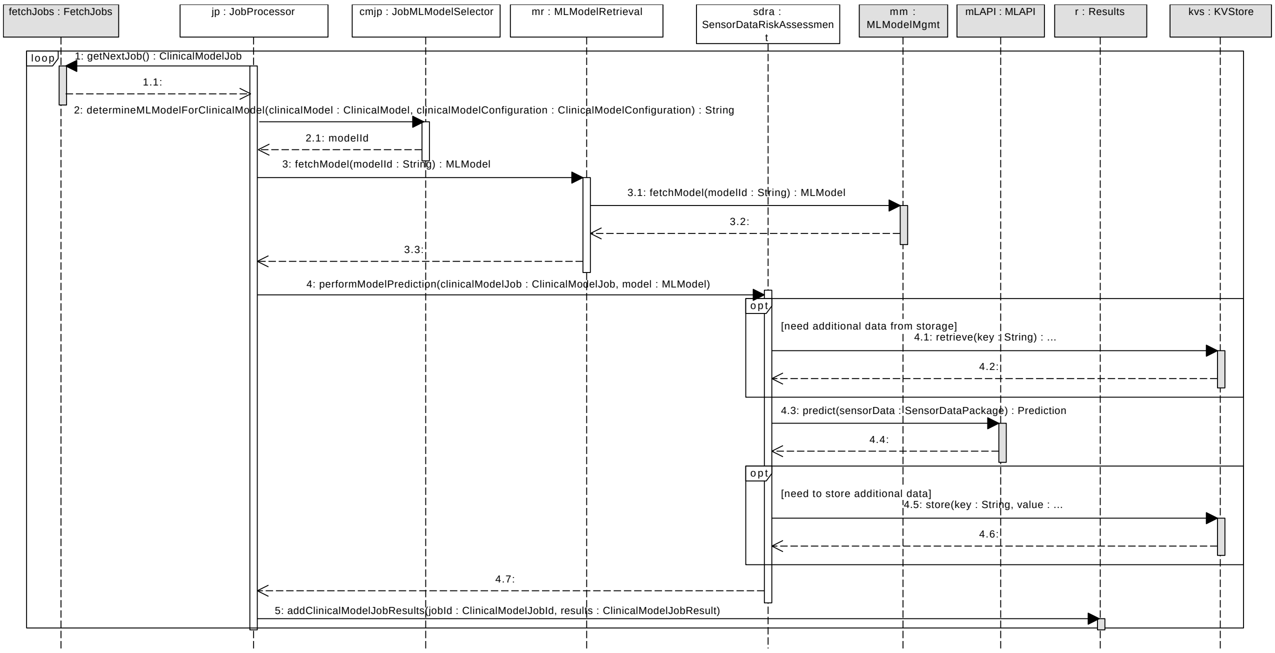

Figure D.2: Compute clinical model result

This sequence diagram shows the detailed application and computation of the clinical model. The first step involves the retrieval of the appropriate **MLModel** (specified by the **ClinicalModel**). The second step involves the calculation of the prediction using the retrieved **MLModel**s. Note that, as this diagram depicts the internal flow between modules within the `RiskEstimationProcessor`, the gray lifelines depict its (external) interfaces.

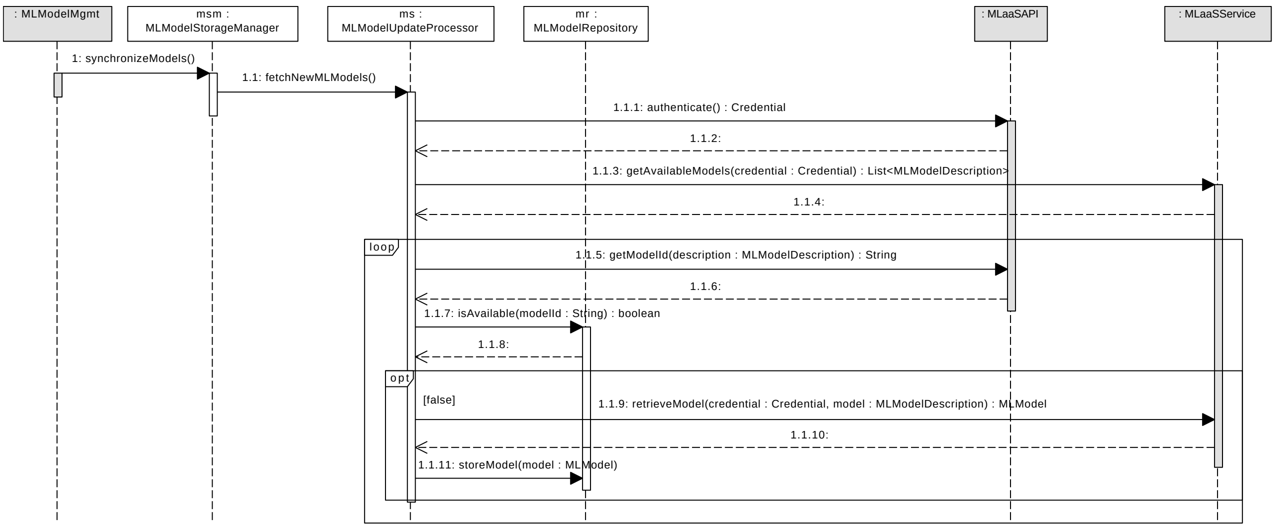

Figure D.3: MLModel Synchronization

As the system operations new **MLModel**s may become available for use by the **ClinicalModel**s. This diagram illustrates the synchronization logic to fetch any unavailable **MLModel**s and to store a local copy of the **MLModel**s in the `MLModelRepository` for faster retrieval and use in the sensor data risk assessment. Note that, as this diagram depicts the internal flow between modules within the `MLModelManager`, the gray lifelines depict its (external) interfaces.

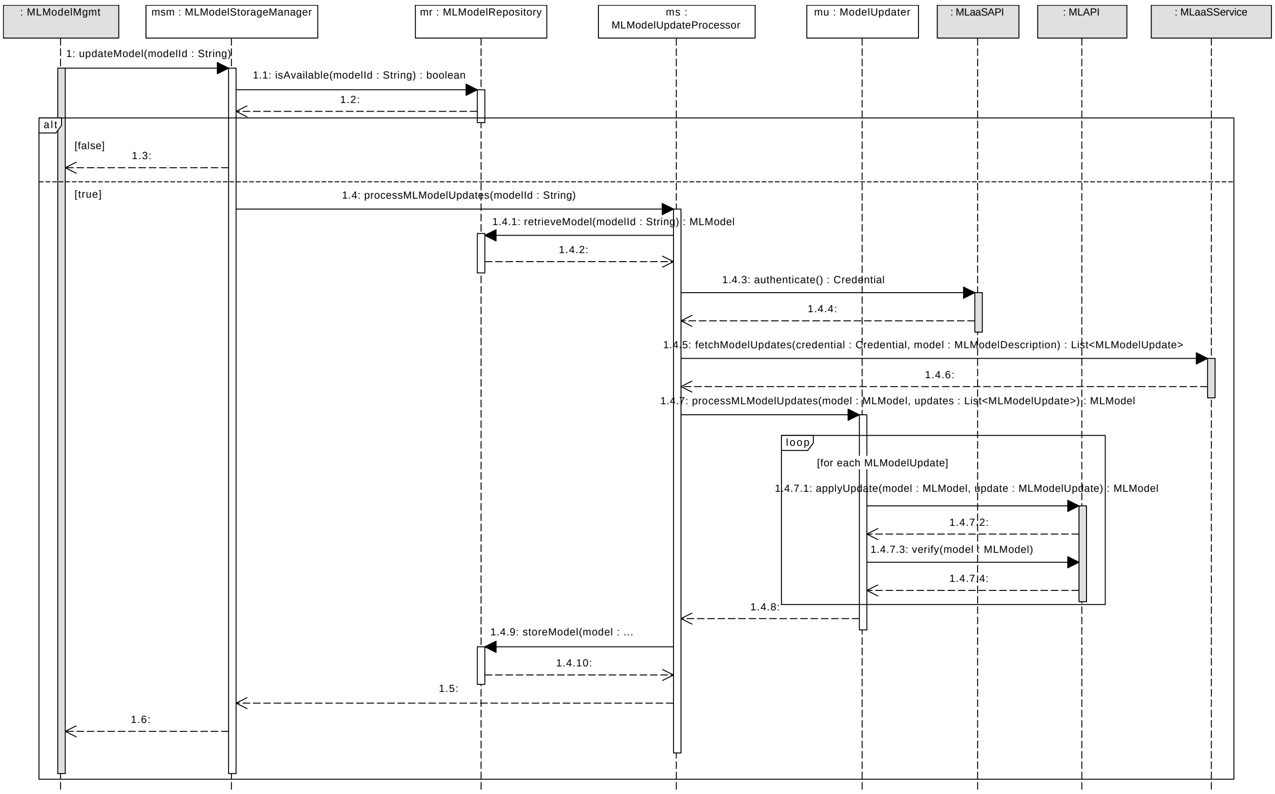

Figure D.4: Processing MLModel updates

MLModelUpdates are made available to improve the existing **MLModel**s. This diagram illustrates the retrieval of these **MLModelUpdate**s and their application on the local **MLModel**. As updates to these models influence the risk assessments, they need to be triggered for each individual **MLModel** that requires an update. Note that, as this diagram depicts the internal flow between modules within the `MLModelManager`, the gray lifelines depict its (external) interfaces.

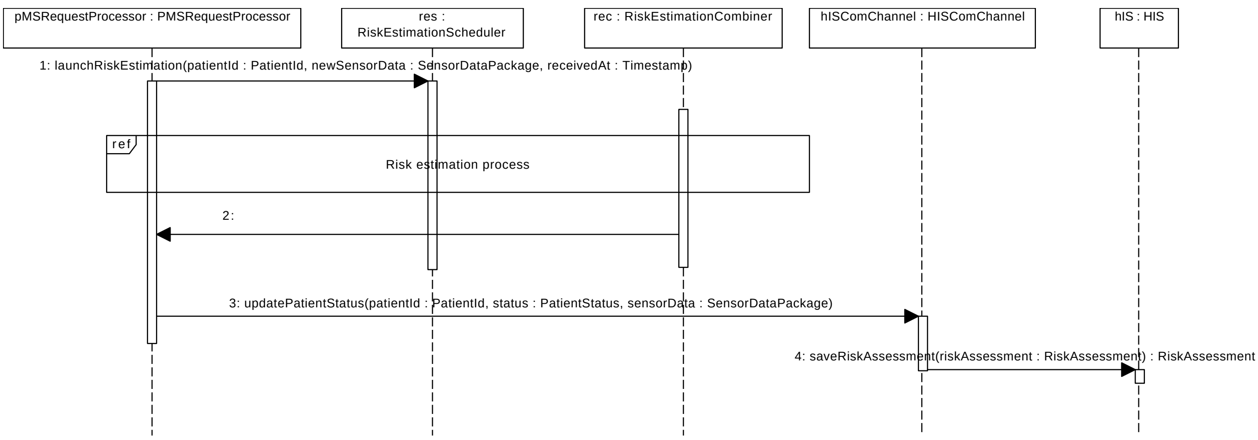

Figure D.5: LaunchRiskEstimation

This diagram shows a small part of the processing of incoming sensor data. It does not yet depict the arrival of the sensor data, storage in a database, or any other processing, as this functionality is not yet worked out.

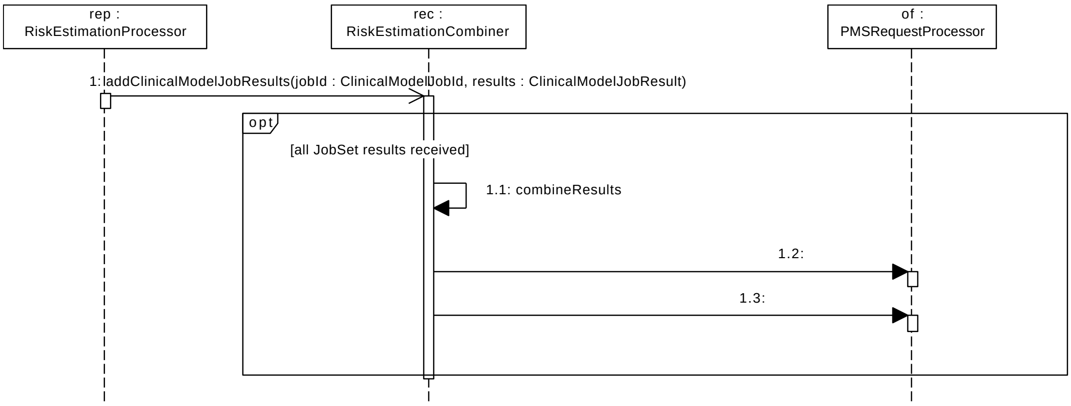

Figure D.6: Combining ClinicalModelJob results

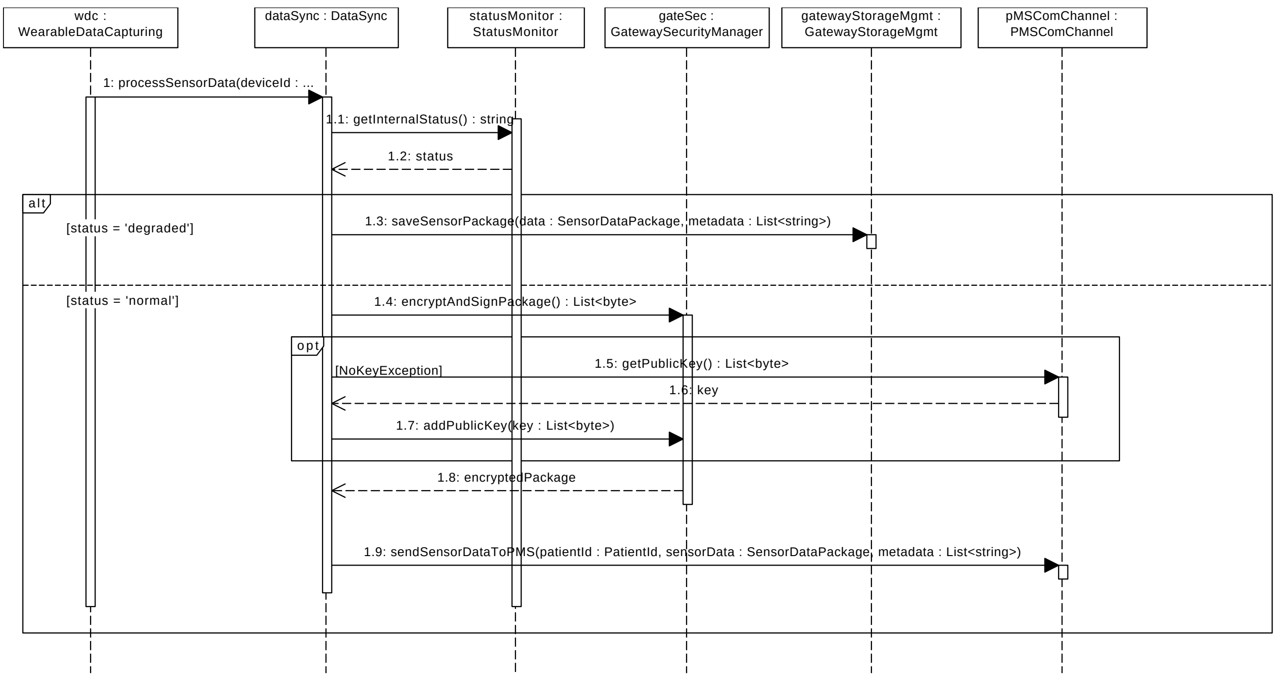

Figure D.7: GatewayInternalDataProcessing

The process of the gateway module get sensor data then either send or store the data. The sensor data will be encrypted by the symmetric key and the request will be sign by the public key

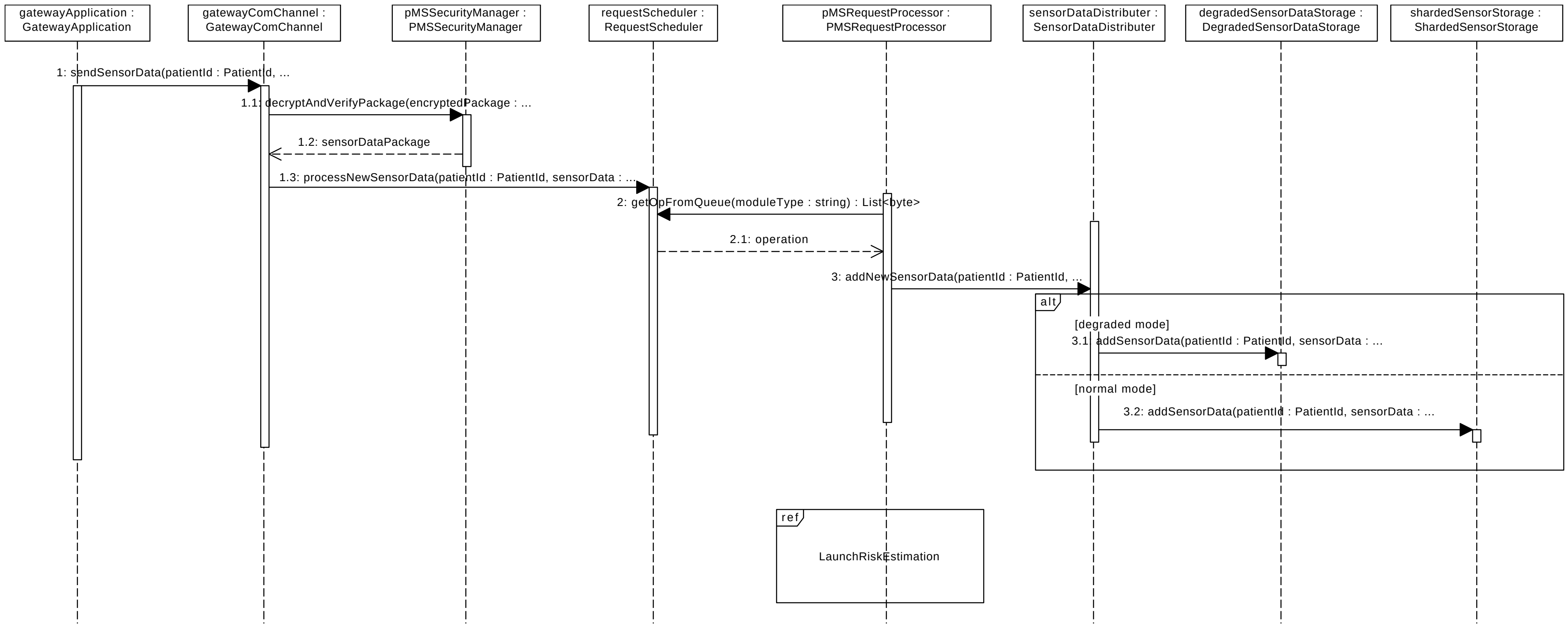

Figure D.8: IncomingSensorData
Upaded version on how incoming sensor data should be processed

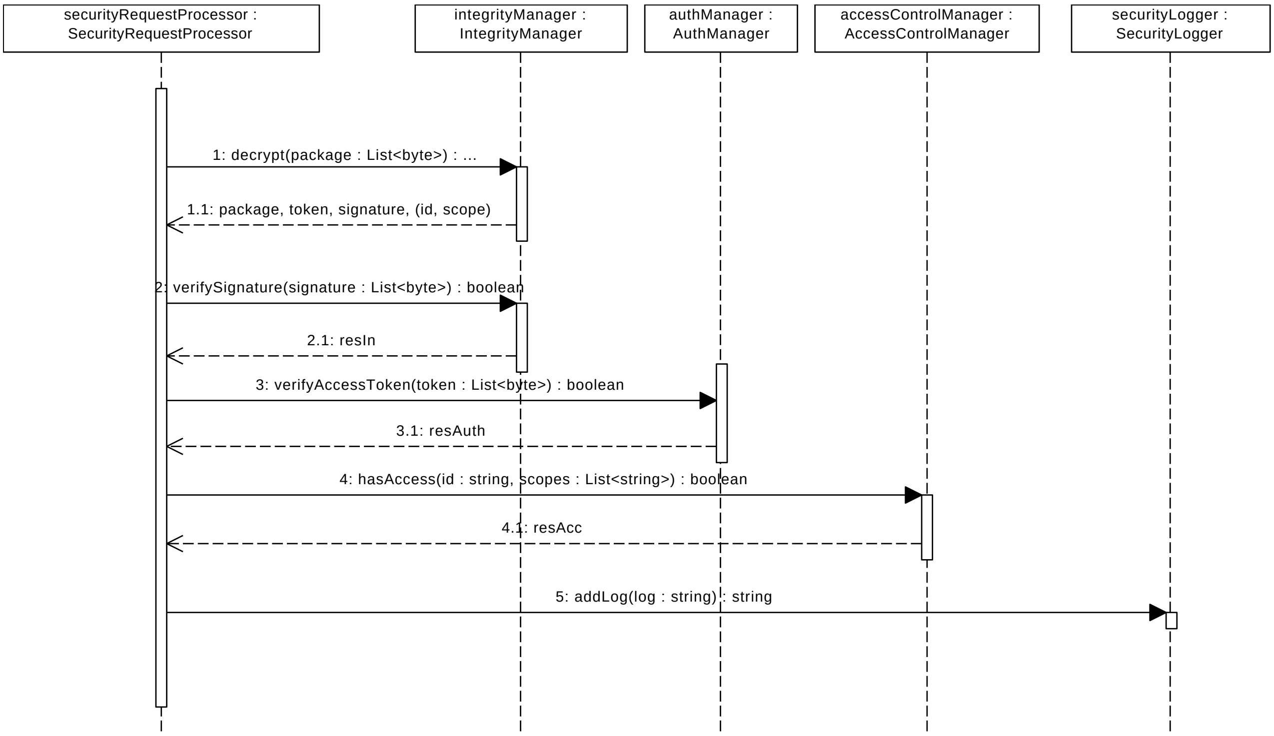

Figure D.9: ApplySecurityCheck
The process of checking a request. The request will be checked for its authenticity, confidentiality, integrity and acees control

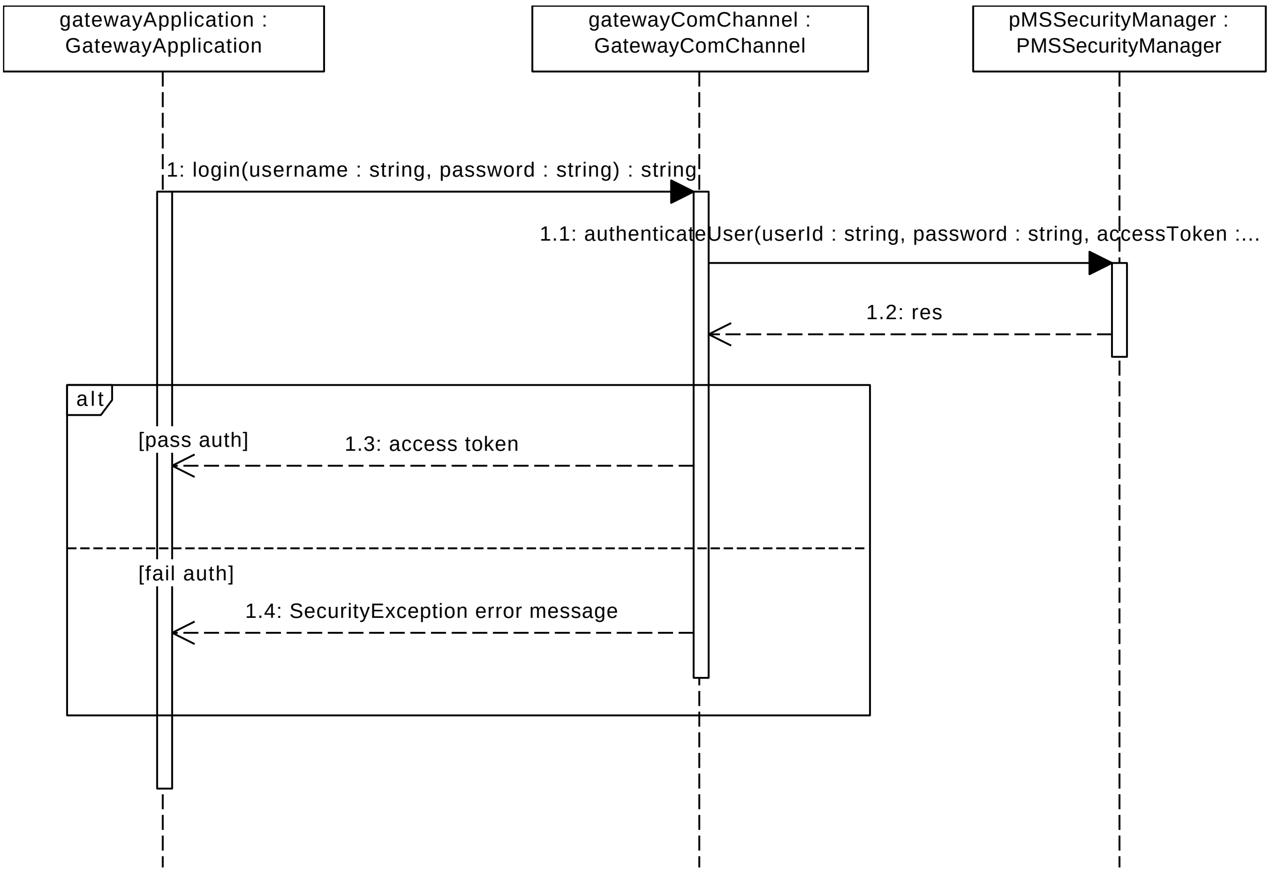

Figure D.10: GatewayLogin
Simple login process for gateway. The same applies for HIS staff user

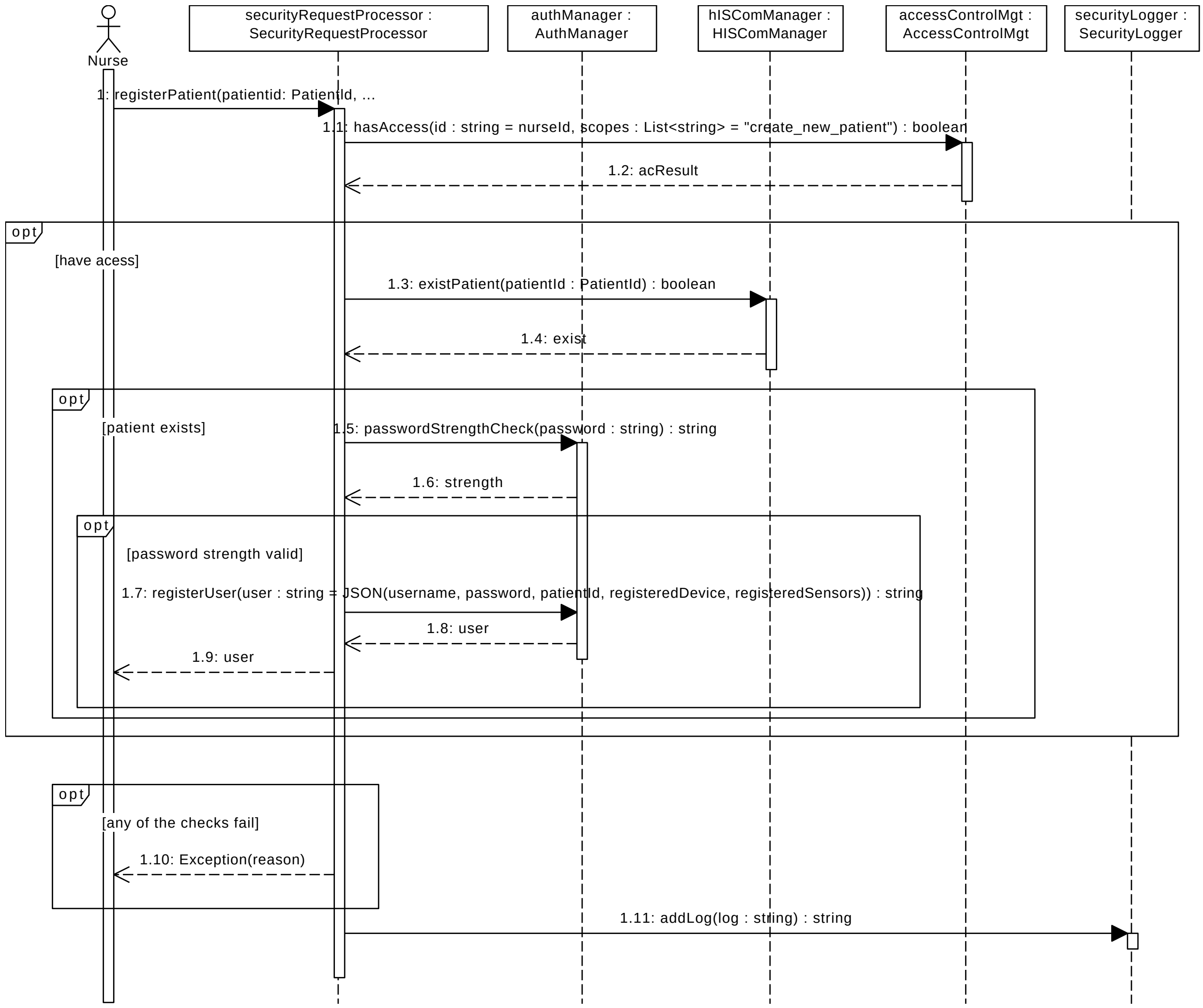

Figure D.11: RegisterPatient

The diagrams describe the process of registering a new patient. Here the nurse will pair the devices and register the gateway as well as the sensor for the patient.

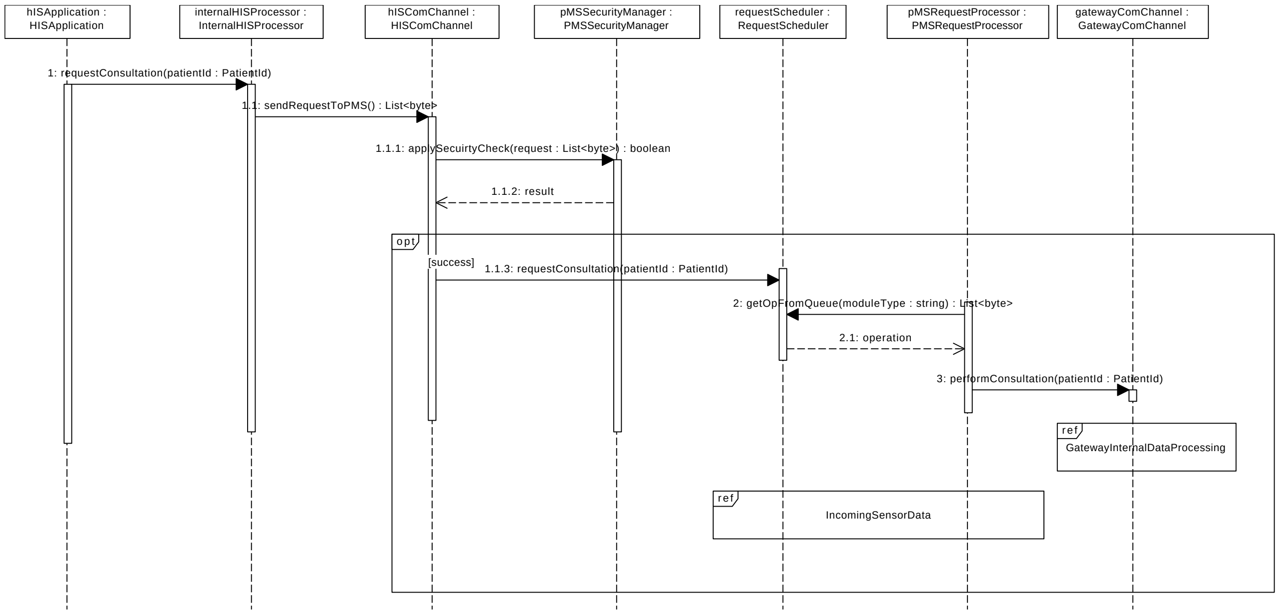

Figure D.12: OnDemandConsultation

The diagrams describes the process of a HIS staff ask for a constulation of a patient. PMS process this request and ask gateway to send a new package of data

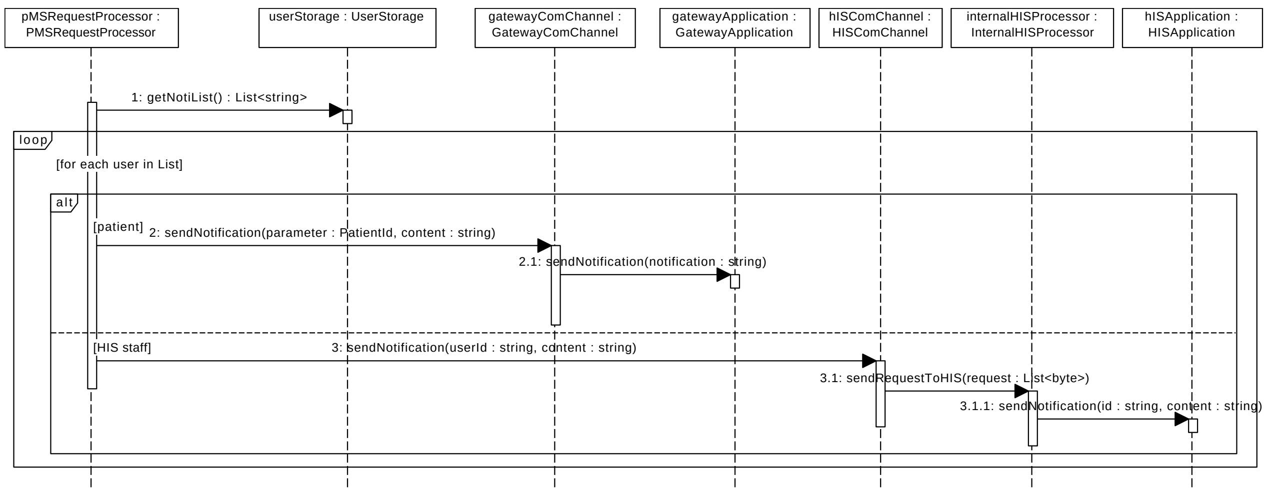

Figure D.13: SendNotification

The diagram describes the process of sending notifications to all relavant parties. The HIS staff can register to the noti list of a patient and notification will be send according to the list

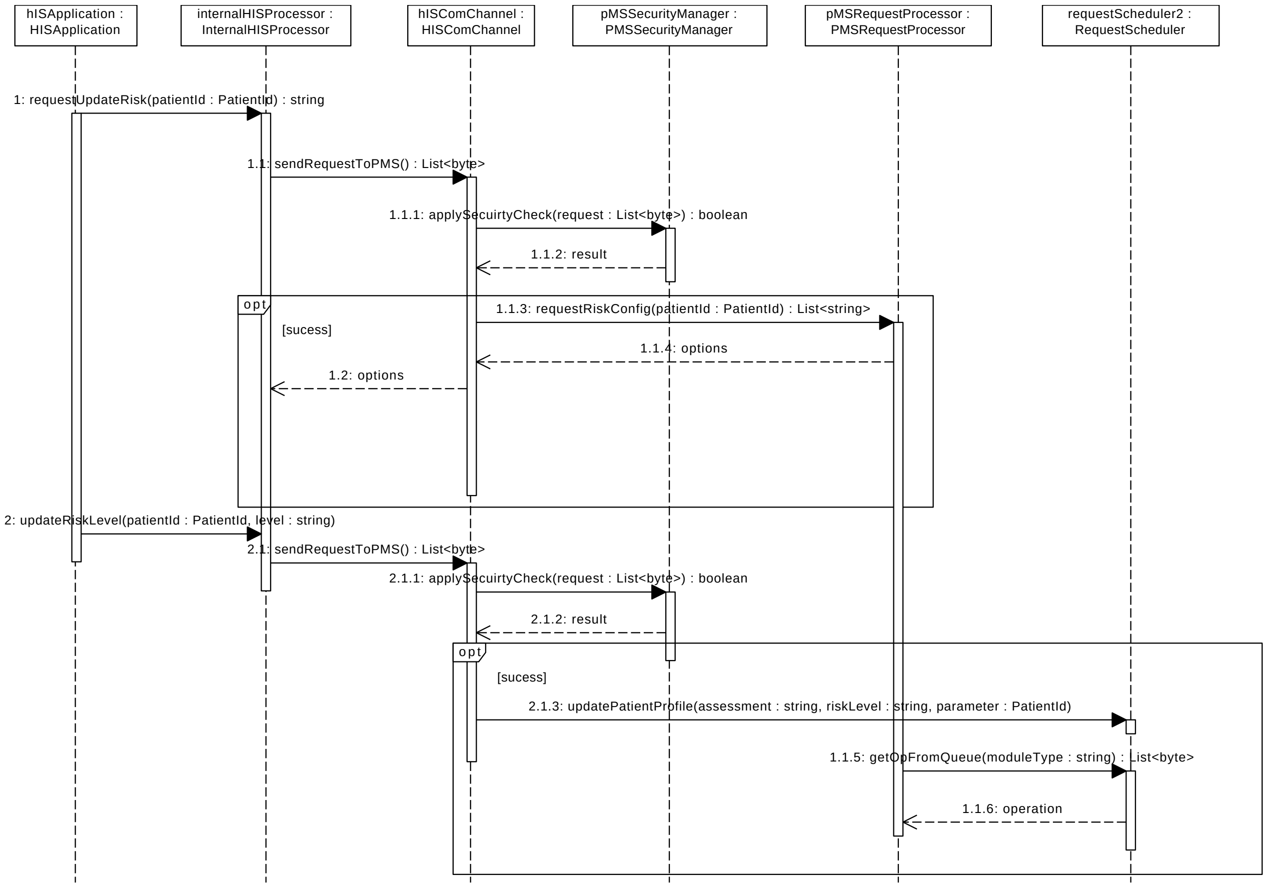

Figure D.14: UpdateRiskLevel

The diagram describes the process of updating a patient risk levels. First the staff must request for a config list and then can perform an update. Note that every requests are subjected to security check. Also not all requests require a scheduler

E. Element catalog

E.1 Components

E.1.1 ClinicalJobCreator

Responsibility: The ClinicalJobCreator is responsible for constructing the **ClinicalModelJob** objects and populating them with the necessary patient health data for execution by the RiskEstimationProcessors.
Super-components: eHealth Platform
Sub-components: None
Provided interfaces: ClinicalJobMgmt
Required interfaces: FetchClinicalModels, OtherDataMgmt, SensorDataMgmt
Deployed on: RiskEstimationManagementNode (Pilot Deployment), RiskEstimationManagmentNode (Dev Deployment), RiskEstimationMgmtNode
Visible on diagrams: figs. A.2, C.2, C.3, C.4 and D.1

E.1.2 ClinicalModelCache

Responsibility: The ClinicalModelCache is a read-through cache responsible for caching the **ClinicalModel**s that should be evaluated for a certain patient and their configurations. This cache is located close to the RiskEstimationProcessor and RiskEstimationCombiner in order to improve the latency of the whole risk estimation flow. The items in the ClinicalModelCache do not expire over time, but should be invalided explicitly if needed. Cached models remain available for the RiskEstimationProcessor while updates are made in the ClinicalModelDB.
Super-components: eHealth Platform
Sub-components: None
Provided interfaces: ClinicalModelCacheMgmt, FetchClinicalModels
Required interfaces: ClinicalModelStorage, OtherDataMgmt
Deployed on: RiskEstimationManagementNode (Pilot Deployment), RiskEstimationManagmentNode (Dev Deployment), RiskEstimationMgmtNode
Visible on diagrams: figs. A.2, C.2, C.3, C.4 and D.1

E.1.3 ClinicalModelDB

Responsibility: The ClinicalModelDB stores all the **ClinicalModel**s and the configurations of these **ClinicalModel**s for the different patients separately from other data. It only allows appending new **ClinicalModel**s.
Super-components: eHealth Platform
Sub-components: None
Provided interfaces: ClinicalModelStorage
Required interfaces: None
Deployed on: RiskEstimationManagmentNode (Dev Deployment), PMS backbone
Visible on diagrams: figs. A.2, C.2, C.3, C.4 and D.1

E.1.4 DegradedSensorDataStorage

Responsibility: degraded sensor data storage when the main storage and its replica failed
Super-components: eHealth Platform
Sub-components: None
Provided interfaces: ShardSensorStorageAPI
Required interfaces: None
Deployed on: PMS backbone
Visible on diagrams: figs. A.2, C.2, C.3, C.4 and D.8

E.1.5 eHealth Platform

Responsibility: The parent eHealth Platform component that represents the context boundary in the client-server view.
Super-components: None
Sub-components: MLModelManager, ClinicalModelCache, RiskEstimationCombiner, RiskEstimationProcessor, RiskEstimationScheduler, ClinicalJobCreator, ClinicalModelDB, GatewayApplication, GatewayComChannel, HISComChannel, SensorDataDistributer, PMSRequestProcessor, PMSSecurityManager, UserStorage, GatewayStorage, GatewayFaultDectector, PMSComHealthMonitor, SubsysStateStorage, DegradedSensorDataStorage, ShardedSensorStorage, SensorDataCache, HISApplication, InternalHISProcessor, RequestScheduler
Provided interfaces: None
Required interfaces: healthAPI, KVStore, MLaaSService
Deployed on: RiskEstimationManagementNode (Pilot Deployment), RiskEstimationProcessorNode (Pilot Deployment), MLaaS Conn (Pilot Deployment), RiskEstimationManagmentNode (Dev Deployment), RiskEstimationProcessorNode (Dev Deployment), PMSComNode, HISNode, UserNode, SensorDataStorageNode, SensorDataStorageReplicaNode, HISStaffNode, MLaaSConnectorNode, RiskEstimationMgmtNode, RiskEstimationProcessorNode, PMS backbone
Visible on diagrams: figs. A.1 and B.1

E.1.6 GatewayApplication

Responsibility: Component represent the gate application where PMS user can interact with
Super-components: eHealth Platform
Sub-components: None
Provided interfaces: GatewayStatusAPI, sendToGateway
Required interfaces: GateWayComAPI, GatewayStorageAPI
Deployed on: UserNode
Visible on diagrams: figs. A.2, B.2, C.2, C.3, C.4, D.8, D.10 and D.13

E.1.7 GatewayComChannel

Responsibility: Gateway communication channel between PMS backend and gateway
Super-components: eHealth Platform
Sub-components: None
Provided interfaces: ContactGateway, GateWayComAPI
Required interfaces: GatewayOperations, PMSComHeartBeat, SecurityAPI, sendToGateway
Deployed on: PMSComNode
Visible on diagrams: figs. A.2, C.2, C.3, C.4, D.8, D.10, D.12 and D.13

E.1.8 GatewayFaultDectector

Responsibility: A process to monitor and report if a gateway is offline for too long
Super-components: eHealth Platform
Sub-components: None
Provided interfaces: None
Required interfaces: GatewayStatusAPI, PMSComHeartBeat
Deployed on: PMSComNode
Visible on diagrams: figs. A.2, C.2, C.3 and C.4

E.1.9 GatewayStorage

Responsibility: Physical storage on gateway device
Super-components: eHealth Platform
Sub-components: None
Provided interfaces: GatewayStorageAPI
Required interfaces: None

Deployed on: ▢ UserNode
Visible on diagrams: figs. A.2, C.2, C.3 and C.4

E.1.10 HIS

Responsibility: Hospital Information System
Super-components: None
Sub-components: None
Provided interfaces: ⊸ healthAPI
Required interfaces: None
Deployed on: ▢ HISNode
Visible on diagrams: figs. A.1, A.3, C.1, C.2, C.3, C.4 and D.5

E.1.11 HISApplication

Responsibility: Application that runs on HIS staff devices
Super-components: eHealth Platform
Sub-components: None
Provided interfaces: ⊸ ContactStaff
Required interfaces: ≺ GetInteralReq
Deployed on: ▢ HISStaffNode
Visible on diagrams: figs. A.2, C.2, C.3, C.4, D.12, D.13 and D.14

E.1.12 HISComChannel

Responsibility: A communication channel between PMS backend and HIS
Super-components: eHealth Platform
Sub-components: None
Provided interfaces: ⊸ HIS2PMS, ⊸ HISDataAPI
Required interfaces: ≺ ContactHIS, ≺ GatewayOperations, ≺ healthAPI, ≺ HISOperations, ≺ PMSComHeartBeat, ≺ SecurityAPI
Deployed on: ▢ PMSComNode
Visible on diagrams: figs. A.1, A.2, C.1, C.2, C.3, C.4, D.5, D.12, D.13 and D.14

E.1.13 InternalHISProcessor

Responsibility: Processing HIS staff request
Super-components: eHealth Platform
Sub-components: None
Provided interfaces: ⊸ ContactHIS, ⊸ GetInteralReq
Required interfaces: ≺ ContactStaff, ≺ HIS2PMS
Deployed on: ▢ HISNode
Visible on diagrams: figs. A.2, C.2, C.3, C.4, D.12, D.13 and D.14

E.1.14 KVStorageService

Responsibility: Service for key-value storage of data.
Super-components: None
Sub-components: None
Provided interfaces: ⊸ KVStore
Required interfaces: None
Deployed on: ▢ RiskEstimationProcessorNode (Pilot Deployment), ▢ RiskEstimationProcessorNode (Dev Deployment), ▢ RiskEstimationProcessorNode
Visible on diagrams: figs. A.1, C.2, C.3 and C.4

E.1.15 MLaaS

Responsibility: MachineLearning-as-a-Service cloud provider. This cloud service provides APIs for managing different **MLModel**s and exchanging **MLModel** updates with other parties.

Super-components: None
Sub-components: None
Provided interfaces: ⊸ MLaaSService
Required interfaces: None
Deployed on: ▢ MLaaSCloud, ▢ MLaaS Service (Pilot Deployment), ▢ MLaaS Service (Dev Deployment)
Visible on diagrams: figs. A.1, C.1, C.3 and C.4

E.1.16 MLModelManager

Responsibility: This is component is responsible for managing the local **MLModel**s for making predictions for malignant events based on the incoming sensor data. It keeps track of the available **MLModel**s that can be used and supports exchanging **MLModel** updates with other systems using the APIs of MLaaS providers.
Super-components: eHealth Platform
Sub-components: None
Provided interfaces: ⊸ MLModelMgmt
Required interfaces: ≺ MLaaSService
Deployed on: ▢ MLaaS Conn (Pilot Deployment), ▢ RiskEstimationManagmentNode (Dev Deployment), ▢ MLaaSConnectorNode
Visible on diagrams: figs. A.1, A.2, B.4, C.1, C.2, C.3 and C.4

E.1.17 PMSComHealthMonitor

Responsibility: A communication channel health monitor process. To ensure to check whether a process is offline for too long
Super-components: eHealth Platform
Sub-components: None
Provided interfaces: ⊸ PMSComHeartBeat
Required interfaces: ≺ SubsysStorageAPI
Deployed on: ▢ PMSComNode
Visible on diagrams: figs. A.2, C.2, C.3 and C.4

E.1.18 PMSRequestProcessor

Responsibility: PMS request processor which is a central processor/distributor after the request get pass the communication (incl. sercuity)
Super-components: eHealth Platform
Sub-components: None
Provided interfaces: ⊸ GatewayOperations, ⊸ HISOperations, ⊸ OtherDataMgmt, ⊸ PatientRecordMgmt, ⊸ SensorDataMgmt
Required interfaces: ≺ ClinicalModelCacheMgmt, ≺ ContactGateway, ≺ HISDataAPI, ≺ LaunchRiskEstimation, ≺ SchedulerAPI, ≺ SensorStorageAPI, ≺ UserStorageAPI
Deployed on: ▢ PMS backbone
Visible on diagrams: figs. A.2, B.5, C.2, C.3, C.4, D.5, D.8, D.12, D.13 and D.14

E.1.19 PMSSecurityManager

Responsibility: A security management process that is responsible for various security measures such as authentication, integrity checking, ensure confidentiality, etc.
Super-components: eHealth Platform
Sub-components: None
Provided interfaces: ⊸ SecurityAPI
Required interfaces: ≺ ContactGateway, ≺ HISDataAPI, ≺ UserStorageAPI
Deployed on: ▢ PMS backbone
Visible on diagrams: figs. A.2, B.3, C.2, C.3, C.4, D.8, D.10, D.12 and D.14

E.1.20 RequestScheduler

Responsibility: A component running as a priority queue that received forward operations/request from PMSRequestProcessor and send it back to process in an order
Super-components: eHealth Platform
Sub-components: None
Provided interfaces: GatewayOperations, HISOperations, SchedulerAPI
Required interfaces: None
Deployed on: PMS backbone
Visible on diagrams: figs. A.2, C.2, C.3, C.4, D.8, D.12 and D.14

E.1.21 RiskEstimationCombiner

Responsibility: The RiskEstimationCombiner is responsible for combining the results of the **ClinicalModelJob**s belonging to a patient risk estimation. More precisely, the RiskEstimationScheduler passes the set of the scheduled jobs for a risk estimation to the RiskEstimationCombiner before scheduling them. The RiskEstimationCombiner then waits for all results to arrive, combines them, and propagates the new sensor data and the results of the risk estimation to the patient record if needed.
Super-components: eHealth Platform
Sub-components: None
Provided interfaces: JobMgmt, Results
Required interfaces: OtherDataMgmt, PatientRecordMgmt
Deployed on: RiskEstimationManagementNode (Pilot Deployment), RiskEstimationManagementNode (Dev Deployment), RiskEstimationMgmtNode
Visible on diagrams: figs. A.2, C.2, C.3, C.4, D.1, D.5 and D.6

E.1.22 RiskEstimationProcessor

Responsibility: The RiskEstimationProcessor is responsible for computing **ClinicalModelJob**s. The RiskEstimationProcessor fetches new **ClinicalModelJob**s from the RiskEstimationScheduler, uses the specified **MLModel** to calculate the risk prediction, and passes the result to the RiskEstimationCombiner. Multiple instances of the RiskEstimationProcessor run in parallel to improve the throughput.
Super-components: eHealth Platform
Sub-components: None
Provided interfaces: None
Required interfaces: FetchJobs, KVStore, MLModelMgmt, Results
Deployed on: RiskEstimationProcessorNode (Pilot Deployment), RiskEstimationProcessorNode (Dev Deployment), RiskEstimationProcessorNode
Visible on diagrams: figs. A.1, A.2, B.6, C.2, C.3, C.4 and D.6

E.1.23 RiskEstimationScheduler

Responsibility: The RiskEstimationScheduler is responsible for taking in new and scheduling requests for risk estimations. It keeps track of the throughput of incoming jobs and changes the scheduling from first-in/first-out to dynamic priority earliest deadline first when going into overload modus.
Super-components: eHealth Platform
Sub-components: None
Provided interfaces: FetchJobs, LaunchRiskEstimation
Required interfaces: ClinicalJobMgmt, FetchClinicalModels, JobMgmt, OtherDataMgmt, SensorDataMgmt
Deployed on: RiskEstimationManagementNode (Pilot Deployment), RiskEstimationManagementNode (Dev Deployment), RiskEstimationMgmtNode
Visible on diagrams: figs. A.2, C.2, C.3, C.4, D.1 and D.5

E.1.24 SensorDataCache

Responsibility: Store Cache of sensor data so that the queries have increasing performance

Super-components: eHealth Platform
Sub-components: None
Provided interfaces: CacheAccess
Required interfaces: None
Deployed on: PMS backbone
Visible on diagrams: figs. A.2, C.2, C.3 and C.4

E.1.25 SensorDataDistributer

Responsibility: internal sensor data storage for PMS
Super-components: eHealth Platform
Sub-components: None
Provided interfaces: SensorStorageAPI
Required interfaces: CacheAccess, ShardSensorStorageAPI
Deployed on: PMS backbone
Visible on diagrams: figs. A.2, C.2, C.3, C.4 and D.8

E.1.26 ShardedSensorStorage

Responsibility: A Sharded database which contain the database of one type of patient (based on risk level)
Super-components: eHealth Platform
Sub-components: None
Provided interfaces: ShardSensorStorageAPI
Required interfaces: None
Deployed on: SensorDataStorageNode, SensorDataStorageReplicaNode
Visible on diagrams: figs. A.2, C.2, C.3, C.4 and D.8

E.1.27 SubsysStateStorage

Responsibility: storage for the states of the subsystem in the communication node. The states will be periodically store and retreive if there is a fault to return the subsystem to its nearest functionning state
Super-components: eHealth Platform
Sub-components: None
Provided interfaces: SubsysStorageAPI
Required interfaces: None
Deployed on: PMSComNode
Visible on diagrams: figs. A.2, C.2, C.3 and C.4

E.1.28 UserStorage

Responsibility: User storage where information regarding the user is stored, information such as user id, patient id, device id, sensors id, etc.
Super-components: eHealth Platform
Sub-components: None
Provided interfaces: UserStorageAPI
Required interfaces: None
Deployed on: PMS backbone
Visible on diagrams: figs. A.2, C.2, C.3, C.4 and D.13

E.2 Modules

E.2.1 AccessControlManager

Responsibility: Access control manager manage the access scope of users
Super-components: eHealth Platform ▷ PMSSecurityManager
Super-modules: None

Sub-modules: None
Provided interfaces: -○ AccessControlMgt
Required interfaces: ≺ SecStorageAPI
Visible on diagrams: figs. B.3 and D.9

E.2.2 AuthManager

Responsibility: Authentication manager module to authorize user
Super-components: eHealth Platform ▷ PMSSecurityManager
Super-modules: None
Sub-modules: None
Provided interfaces: -○ AuthRequest
Required interfaces: ≺ SecStorageAPI
Visible on diagrams: figs. B.3, D.9 and D.11

E.2.3 BackupComChannel

Responsibility: a backup channel that overtake the main channel if there is a fault
Super-components: eHealth Platform ▷ GatewayApplication
Super-modules: None
Sub-modules: None
Provided interfaces: -○ BackupComAPI, -○ PMSComAPI
Required interfaces: None
Visible on diagrams: fig. B.2

E.2.4 DataSync

Responsibility: a module that process the incoming sensor data from wearable unit. If the gateway in degraded mode then the module will temporary send the data to the physical storage. Otherwise forward it to PMS backend
Super-components: eHealth Platform ▷ GatewayApplication
Super-modules: None
Sub-modules: None
Provided interfaces: -○ ModeAPI, -○ SyncAPI
Required interfaces: ≺ GatewaySecApi, ≺ GWSAPI, ≺ InternalStatusAPI, ≺ LWCheckAPI, ≺ PMSComAPI
Visible on diagrams: figs. B.2 and D.7

E.2.5 GatewayOpsProcessor

Responsibility: a module manage the operations related to gateway (either invoke from gateway or it is to call gateway)
Super-components: eHealth Platform ▷ PMSRequestProcessor
Super-modules: None
Sub-modules: None
Provided interfaces: -○ GatewayOperations
Required interfaces: ≺ ClinicalModelCacheMgmt, ≺ ContactGateway, ≺ HISDataAPI, ≺ InternalSchedule, ≺ LaunchRiskEstimation
Visible on diagrams: fig. B.5

E.2.6 GatewaySecurityManager

Responsibility: A module manage the security aspect of the gateway
Super-components: eHealth Platform ▷ GatewayApplication
Super-modules: None
Sub-modules: None
Provided interfaces: -○ GatewaySecApi
Required interfaces: None
Visible on diagrams: figs. B.2 and D.7

E.2.7 GatewayStorageMgmt

Responsibility: a module manage the gateway storage
Super-components: eHealth Platform ▷ GatewayApplication
Super-modules: None
Sub-modules: None
Provided interfaces: -○ GWSAPI
Required interfaces: -≺ GatewayStorageAPI
Visible on diagrams: figs. B.2 and D.7

E.2.8 HardwareMgmt

Responsibility: Wearable hardware management. To get physcial data from the hardware such as baterry , etc.
Super-components: eHealth Platform ▷ GatewayApplication
Super-modules: None
Sub-modules: None
Provided interfaces: -○ HardwareAPI
Required interfaces: None
Visible on diagrams: fig. B.2

E.2.9 HISComManager

Responsibility: a module responsible for fetching data from HIS
Super-components: eHealth Platform ▷ PMSSecurityManager
Super-modules: None
Sub-modules: None
Provided interfaces: -○ HISGetInfo
Required interfaces: -≺ HISDataAPI
Visible on diagrams: figs. B.3 and D.11

E.2.10 HISOpsProcessor

Responsibility: process operations from HIS
Super-components: eHealth Platform ▷ PMSRequestProcessor
Super-modules: None
Sub-modules: None
Provided interfaces: -○ HISOperations
Required interfaces: -≺ InternalSchedule, -≺ SensorStorageAPI, -≺ UserStorageAPI
Visible on diagrams: fig. B.5

E.2.11 IntegrityManager

Responsibility: Integrity manager module that is responsible for encrypt/decrypt message as well as request integrity verification
Super-components: eHealth Platform ▷ PMSSecurityManager
Super-modules: None
Sub-modules: None
Provided interfaces: -○ IntegrityAPI
Required interfaces: -≺ SecStorageAPI
Visible on diagrams: figs. B.3 and D.9

E.2.12 JobMLModelSelector

Responsibility: This module is responsible for determining the appropriate **MLModel** to use for making the prediction on the provided sensor data, as there could be multiple **MLModel** candidates for making predictions.
Super-components: RiskEstimationProcessor ▷ eHealth Platform
Super-modules: None

Sub-modules: None
Provided interfaces: ⊸ ApplyClinicalModel
Required interfaces: None
Visible on diagrams: figs. B.6 and D.2

E.2.13 JobProcessor

Responsibility: The JobProcessor module coordinates the execution of the **ClinicalModelJob**s to ensure that the required **MLModel**s are fetched, and the prediction is performed with the appropriate **MLModel**.
Super-components: RiskEstimationProcessor ▷ eHealth Platform
Super-modules: None
Sub-modules: None
Provided interfaces: None
Required interfaces: ≺ ApplyClinicalModel, ≺ FetchJobs, ≺ FetchMLModel, ≺ Results, ≺ SensorDataAnalysis
Visible on diagrams: figs. B.6 and D.2

E.2.14 LightweightCheck

Responsibility: A module contain light weight ML check for estimate risk level
Super-components: eHealth Platform ▷ GatewayApplication
Super-modules: None
Sub-modules: None
Provided interfaces: ⊸ LWCheckAPI
Required interfaces: ≺ GWSAPI
Visible on diagrams: fig. B.2

E.2.15 MLaaSLibrary

Responsibility: MLaaS development library for working with the MLaaS providers. It supports storage & retrieval of **MLModel**s, and exchanging **MLModelUpdate**s with MLaaS providers.
Super-components: None
Super-modules: None
Sub-modules: None
Provided interfaces: ⊸ MLaaSAPI
Required interfaces: None
Visible on diagrams: fig. B.1

E.2.16 MLLibrary

Responsibility: Library for working with **MLModel**s locally and using them to make predictions based on the incoming sensor data. The library can also process external updates to apply them to a specific **MLModel**.
Super-components: None
Super-modules: None
Sub-modules: None
Provided interfaces: ⊸ MLAPI
Required interfaces: None
Visible on diagrams: fig. B.1

E.2.17 MLModelRepository

Responsibility: This module stores **MLModel**s locally and makes them available for use by the RiskEstimationProcessor when processing **ClinicalModelJob**s.
Super-components: eHealth Platform ▷ MLModelManager
Super-modules: None
Sub-modules: None
Provided interfaces: ⊸ MLModelStorage

Required interfaces: None
Visible on diagrams: figs. B.4, D.3 and D.4

E.2.18 MLModelRetrieval

Responsibility: This module is responsible for retrieving the requested **MLModel**s from the MLModelManager.
Super-components: RiskEstimationProcessor ▷ eHealth Platform
Super-modules: None
Sub-modules: None
Provided interfaces: FetchMLModel
Required interfaces: MLModelMgmt
Visible on diagrams: figs. B.6 and D.2

E.2.19 MLModelStorageManager

Responsibility: This module is responsible for managing the locally stored **MLModel**s, triggering the synchronization and retrieval of new **MLModel**s, and coordinating the updates to the existing **MLModel**s.
Super-components: eHealth Platform ▷ MLModelManager
Super-modules: None
Sub-modules: None
Provided interfaces: MLModelMgmt
Required interfaces: MLModelStorage, MLModelSync
Visible on diagrams: figs. B.4, D.3 and D.4

E.2.20 MLModelUpdateProcessor

Responsibility: The MLModelUpdater processes and applies updates to the local **MLModel**s to prevent updates to these models from requiring to retrieve the full **MLModel**s.
Super-components: eHealth Platform ▷ MLModelManager
Super-modules: None
Sub-modules: None
Provided interfaces: MLModelSync
Required interfaces: MLaaSAPI, MLaaSService, MLModelStorage, MLModelUpdateMgmt
Visible on diagrams: figs. B.1, B.4, D.3 and D.4

E.2.21 ModelUpdater

Responsibility: This module is responsible for processing **MLModelUpdate**s and applying them to the **MLModel**.
Super-components: eHealth Platform ▷ MLModelManager
Super-modules: None
Sub-modules: None
Provided interfaces: MLModelUpdateMgmt
Required interfaces: MLAPI
Visible on diagrams: figs. B.4 and D.4

E.2.22 PatientManager

Responsibility: a module manage patient operations
Super-components: eHealth Platform ▷ PMSRequestProcessor
Super-modules: None
Sub-modules: None
Provided interfaces: OtherDataMgmt, PatientRecordMgmt
Required interfaces: HISDataAPI, InternalSchedule, UserStorageAPI
Visible on diagrams: fig. B.5

E.2.23 PMSComChannel

Responsibility: PMS comunication channel to call for operations from PMS
Super-components: eHealth Platform ▷ GatewayApplication
Super-modules: None
Sub-modules: None
Provided interfaces: ComAPI, PMSComAPI, sendToGateway
Required interfaces: GateWayComAPI, GatewaySecApi
Visible on diagrams: figs. B.2 and D.7

E.2.24 PMSRequestScheduleSync

Responsibility: forward/distribute requests
Super-components: eHealth Platform ▷ PMSRequestProcessor
Super-modules: None
Sub-modules: None
Provided interfaces: InternalSchedule
Required interfaces: SchedulerAPI
Visible on diagrams: fig. B.5

E.2.25 SecurityLogger

Responsibility: Security logging module which add log every time an action happens inside the SecurityManager component
Super-components: eHealth Platform ▷ PMSSecurityManager
Super-modules: None
Sub-modules: None
Provided interfaces: SecLogging
Required interfaces: SecStorageAPI
Visible on diagrams: figs. B.3, D.9 and D.11

E.2.26 SecurityRequestProcessor

Responsibility: A module that is a processor to distribute jobs among the sub modulesbased on the quest
Super-components: eHealth Platform ▷ PMSSecurityManager
Super-modules: None
Sub-modules: None
Provided interfaces: SecurityAPI
Required interfaces: AccessControlMgt, AuthRequest, ContactGateway, HISGetInfo, IntegrityAPI, SecLogging
Visible on diagrams: figs. B.3, D.9 and D.11

E.2.27 SecurityStorageManager

Responsibility: A gateway to get/store data to the key-value storage
Super-components: eHealth Platform ▷ PMSSecurityManager
Super-modules: None
Sub-modules: None
Provided interfaces: SecStorageAPI
Required interfaces: KVStore, UserStorageAPI
Visible on diagrams: fig. B.3

E.2.28 SensorDataRiskAssessment

Responsibility: This module performs the actual risk assessment by using the **MLModel** to make a prediction on the sensor data.
Super-components: RiskEstimationProcessor ▷ eHealth Platform
Super-modules: None

Sub-modules: None
Provided interfaces: -○ SensorDataAnalysis
Required interfaces: ≺ KVStore, ≺ MLAPI
Visible on diagrams: figs. B.1, B.6 and D.2

E.2.29 SensorDataSync

Responsibility: a module that help adding/storing and sync the raw sensor data with the internal PMS storage
Super-components: eHealth Platform ▷ PMSRequestProcessor
Super-modules: None
Sub-modules: None
Provided interfaces: -○ SensorDataMgmt
Required interfaces: ≺ InternalSchedule, ≺ SensorStorageAPI
Visible on diagrams: fig. B.5

E.2.30 StatusMonitor

Responsibility: A central health monitor to monitor the sub modules in gateway to ensure everything is running smoothly
Super-components: eHealth Platform ▷ GatewayApplication
Super-modules: None
Sub-modules: None
Provided interfaces: -○ GatewayStatusAPI, -○ InternalStatusAPI
Required interfaces: ≺ BackupComAPI, ≺ ComAPI, ≺ HardwareAPI, ≺ ModeAPI, ≺ WDCPingAPI
Visible on diagrams: figs. B.2 and D.7

E.2.31 WearableDataCapturing

Responsibility: a module responisble for comminication with wearable unit and receive sensor data
Super-components: eHealth Platform ▷ GatewayApplication
Super-modules: None
Sub-modules: None
Provided interfaces: -○ WDCPingAPI
Required interfaces: ≺ SyncAPI
Visible on diagrams: figs. B.2 and D.7

E.3 Interfaces

E.3.1 AccessControlMgt

Provided by: AccessControlManager
Required by: SecurityRequestProcessor
Operations:

- boolean deviceCheck(List<string> metadata)
  - Effect: verify whether the data user sent is from their devices
  - Returns: whether it passes the checking process or not
  - Sequence Diagrams: None
- boolean hasAccess(string id, List<string> scopes)
  - Effect: check whether the user have the access to the scope of data
  - Returns: whether the user have the necessary access scope for the request
  - Sequence Diagrams: figs. D.9 and D.11

Diagrams: None

E.3.2 ApplyClinicalModel

Provided by: JobMLModelSelector
Required by: JobProcessor

Operations:

- String determineMLModelForClinicalModel(**ClinicalModel** clinicalModel, **ClinicalModelConfiguration** clinicalModelConfiguration)
  - Effect: Apply the **ClinicalModel** to obtain the most appropriate **MLModel** to use for the calculation.
  - Returns: **Identifier** of the **MLModel** to use.
  - Sequence Diagrams: fig. D.2

Diagrams: None

E.3.3 AuthRequest

Provided by: AuthManager
Required by: SecurityRequestProcessor
Operations:

- string authenticateUser(string username, string password)
  - Effect: Authenticate user
  - Returns: Results of authentication in JSON format
  - Sequence Diagrams: fig. D.9
- string passwordStrengthCheck(string password)
  - Effect: Since the password constraint require User passwords are enforced to be minimally 8 characters and maximally 64 characters long, and should not be commonly used passwords. This method will check this contraints
  - Returns: either "ok" or a reason why the check fail.
  - Sequence Diagrams: fig. D.11
- string registerUser(string user)
  - Effect: register user and their devices. Input will take a form of a json string so that it fits with the storage
  - Returns: JSON user from storage
  - Sequence Diagrams: fig. D.11
- boolean verifyAccessToken(List<byte> token)
  - Effect: Check the authenticity of the access token
  - Returns: whether the access token is valid
  - Sequence Diagrams: fig. D.9

Diagrams: None

E.3.4 BackupComAPI

Provided by: BackupComChannel
Required by: StatusMonitor
Operations:

- string ping()
  - Effect: ping to check for issue
  - Returns: return a status string
  - Sequence Diagrams: None

Diagrams: None

E.3.5 CacheAccess

Provided by: SensorDataCache
Required by: SensorDataDistributer
Operations:

- void addToCache(**PatientId** patientId, **SensorDataPackage** sensorData)
  - Effect: add sensorDataToCache
  - Sequence Diagrams: None
- **SensorDataPackage** getFromCache(**PatientId** patientId)
  - Effect: get from cache for faster query time
  - Returns: return a data package if there is a cached version, otherwise throw IOException
  - Sequence Diagrams: None

Diagrams: None

E.3.6 ClinicalJobMgmt

Provided by: ClinicalJobCreator
Required by: RiskEstimationScheduler
Operations:

- List<Tuple<**ClinicalModelJob**, Double>> createClinicalModelJobsForPatient(**PatientId** patientId, Tuple<**SensorDataPac** **Timestamp**> sensorData)
  - Effect: Construct the list of **ClinicalModelJob**s for a specified patient. These jobs are populated with the necessary data for their execution by the RiskEstimationProcessor.
  - Returns: A list of tuples containing the **ClinicalModelJob** and its weight.
  - Sequence Diagrams: fig. D.1

Diagrams: None

E.3.7 ClinicalModelCacheMgmt

Provided by: ClinicalModelCache
Required by: GatewayOpsProcessor, PMSRequestProcessor
Operations:

- void invalidateCacheEntries(**PatientId** patientId)
  - Effect: The ClinicalModelCache will invalidate (i.e., remove) all items in its cache for the patient identified by patientId. If the cache does not contain any items for this patient, nothing is changed. After invalidating the cached items for a certain patient, the next request for them will lead to fetching them from the database and storing them in the cache again.
  - Sequence Diagrams: None

Diagrams: None

E.3.8 ClinicalModelStorage

Provided by: ClinicalModelDB
Required by: ClinicalModelCache
Operations:

- Map<**ClinicalModel**, **ClinicalModelConfiguration**> getAssignedClinicalModelsAndConfigForPatient(**PatientId** patientId) throws *NoSuchPatientException*
  - Effect: Fetch and return the ids of the **ClinicalModel**s assigned to the patient identified by patientId and their configurations for this patient. The configurations are returned as a **ClinicalModelConfiguration** containing a map of configuration parameters and their value.
  - Returns: A map with for every applicable **ClinicalModel** the corresponding **ClinicalModelConfiguration**.
  - Sequence Diagrams: fig. D.1

Diagrams: None

E.3.9 ComAPI

Provided by: PMSComChannel
Required by: StatusMonitor
Operations:

- string ping()
  - Effect: ping to check for issue
  - Returns: return a status string
  - Sequence Diagrams: None

Diagrams: None

E.3.10 ContactGateway

Provided by: GatewayComChannel
Required by: GatewayOpsProcessor, PMSRequestProcessor, PMSSecurityManager, SecurityRequestProcessor

Operations:

- void failRequest(string log)
  - Effect: contact gateway and inform that the security failed, usually leads to re-login
  - Sequence Diagrams: fig. D.9
- void performConsultation(**PatientId** patientId)
  - Effect: perform consultation
  - Sequence Diagrams: fig. D.12
- void sendNotification(**PatientId** parameter, string content)
  - Effect: Send notification to **Patient**'s gateway
  - Sequence Diagrams: fig. D.13

Diagrams: None

E.3.11 ContactHIS

Provided by: InternalHISProcessor
Required by: HISComChannel
Operations:

- void sendRequestToHIS(List<byte> request)
  - Effect: send request to HIS
  - Sequence Diagrams: fig. D.13

Diagrams: None

E.3.12 ContactStaff

Provided by: HISApplication
Required by: InternalHISProcessor
Operations:

- void sendNotification(string id, string content)
  - Effect: send notification to HIS staff
  - Sequence Diagrams: fig. D.13

Diagrams: None

E.3.13 FetchClinicalModels

Provided by: ClinicalModelCache
Required by: ClinicalJobCreator, RiskEstimationScheduler
Operations:

- Map<**ClinicalModel**, Tuple<**ClinicalModelConfiguration**, Double>> getAssignedClinicalModelsAndConfigForPatient(**PatientId** patientId)
  - Effect: The ClinicalModelCache will return the clinical models assigned to the patient identified by patientId and their configurations for this patient. The configurations are a map of configuration parameters and their value. If the cache contains data for the patient with given id, the data from the cache is returned. If not, the ClinicalModelCache will fetch all clinical models assigned to the patient with given patientId and their configurations for this patient, store these in the cache and return them.
  - Returns: A map with for every applicable **ClinicalModel**, a tuple containing the corresponding **ClinicalModelConfiguration** for the specified patient and the corresponding weight factors used for combining the results.
  - Sequence Diagrams: fig. D.1

Diagrams: None

E.3.14 FetchJobs

Provided by: RiskEstimationScheduler
Required by: JobProcessor, RiskEstimationProcessor
Operations:

- **ClinicalModelJob** getNextJob()

– Effect: Returns the next clinical model computation job that must be performed (i.e., the first job in the queue).
– Returns: The next clinical model computation job that must be performed.
– Sequence Diagrams: fig. D.2

Diagrams: None

E.3.15 FetchMLModel

Provided by: MLModelRetrieval
Required by: JobProcessor
Operations:

- **MLModel** fetchModel(String modelId) throws *NoSuchMLModelException*
  - Effect: Fetch the **MLModel** corresponding with the provided modelId.
  - Returns: The requested **MLModel**.
  - Sequence Diagrams: fig. D.2

Diagrams: None

E.3.16 GateWayComAPI

Provided by: GatewayComChannel
Required by: GatewayApplication, PMSComChannel
Operations:

- List<byte> getSecurityKeys()
  - Effect: a method to get the public key and a symetric key that can be used for signing and encryption
  - Returns: return the public key and a symetric key that can be used for signing and encryption
  - Sequence Diagrams: None
- string login(string username, string password)
  - Effect: login using gateway
  - Returns: access token
  - Sequence Diagrams: fig. D.10
- void sendSensorData(**PatientId** patientId, List<byte> encrptedSensorData, string metadata)
  - Effect: send request/data to PMS
  - Sequence Diagrams: fig. D.8

Diagrams: None

E.3.17 GatewayOperations

Provided by: GatewayOpsProcessor, PMSRequestProcessor, RequestScheduler
Required by: GatewayComChannel, HISComChannel
Operations:

- void processNewSensorData(**PatientId** patientId, **SensorDataPackage** sensorData, string additionalInfo)
  - Effect: forward the sensor data to the next step in PMS backend after passing the security component
  - Sequence Diagrams: fig. D.8

Diagrams: None

E.3.18 GatewaySecApi

Provided by: GatewaySecurityManager
Required by: DataSync, PMSComChannel
Operations:

- void addPublicKey(List<byte> key)
  - Effect: assign public key to the security manager
  - Sequence Diagrams: fig. D.7
- List<byte> encryptAndSignPackage()
  - Effect: a method to encrypt and sign the data package to ensure the confidentiality and integrity of the package

– Returns: encypted data package or throw a **NoKeyException** in case of there are no key
– Sequence Diagrams: fig. D.7

Diagrams: None

E.3.19 GatewayStatusAPI

Provided by: GatewayApplication, StatusMonitor
Required by: GatewayFaultDectector
Operations:

- Pair <int, string> getGatewayStatus()
  - Effect: get the status of gateway
  - Returns: a status string represent the status of the gateway with their gateway id
  - Sequence Diagrams: None

Diagrams: None

E.3.20 GatewayStorageAPI

Provided by: GatewayStorage
Required by: GatewayApplication, GatewayStorageMgmt
Operations:

- void cleanOldData()
  - Effect: invoke to remove data that is stored over 24h
  - Sequence Diagrams: None
- List< Pair <**SensorDataPackage**>> getStoredSensorPackages()
  - Effect: retrived all stored sensor package with their meta-data
  - Returns: tupple of metadata, data
  - Sequence Diagrams: None
- void saveSensorData1(**SensorDataPackage** sensorData, List<string> metadata)
  - Effect: save temporary sensor data in physical storage
  - Sequence Diagrams: None

Diagrams: None

E.3.21 GetInteralReq

Provided by: InternalHISProcessor
Required by: HISApplication
Operations:

- **PatientStatus** getPatientStatus(**PatientId** patientId)
  - Effect: get patient status
  - Returns: patient stauts
  - Sequence Diagrams: None
- string login(string username, string password)
  - Effect: HIS staff login in PMS system
  - Returns: access token
  - Sequence Diagrams: None
- void registerNotiForPatient(**PatientId** patientId)
  - Effect: register an intesterest in receiving notification for a patient
  - Sequence Diagrams: None
- List<string> requestConfigRisk(**PatientId** patientId)
  - Effect: HIS staff request for a configuration options
  - Returns: config options
  - Sequence Diagrams: None
- void requestConsultation(**PatientId** patientId)
  - Effect: request on-demand consultation
  - Sequence Diagrams: fig. D.12
- string requestUpdateRisk(**PatientId** patientId)
  - Effect: request to perform an update risk level
  - Returns: risk options

– Sequence Diagrams: fig. D.14

- void updateRiskConfig(**PatientId** id, string config)
  - Effect: update a patient risk assessment config
  - Sequence Diagrams: None
- void updateRiskLevel(**PatientId** patientId, string level)
  - Effect: update patient risk level
  - Sequence Diagrams: fig. D.14

Diagrams: None

E.3.22 GWSAPI

Provided by: GatewayStorageMgmt
Required by: DataSync, LightweightCheck
Operations:

- List< Pair <**SensorDataPackage**>> getStoredData()
  - Effect: retrieve all temporary packages
  - Returns: tupple of data, metadata
  - Sequence Diagrams: None
- void saveSensorPackage(**SensorDataPackage** data, List<string> metadata)
  - Effect: temporary store sensor package when the app is in degraded mode for 24 hours
  - Sequence Diagrams: fig. D.7

Diagrams: None

E.3.23 HardwareAPI

Provided by: HardwareMgmt
Required by: StatusMonitor
Operations:

- int getBatterPercentage()
  - Effect: get the remaining battery level
  - Returns: between 0 and 100 represent the percentage of the remainning battery
  - Sequence Diagrams: None
- string getNetworkStatus()
  - Effect: get the device network status
  - Returns: a string represent the status of the device
  - Sequence Diagrams: None

Diagrams: None

E.3.24 healthAPI

Provided by: HIS
Required by: HISComChannel, eHealth Platform
Description : This interface provides a reduced set of simplified operations that are sufficient in the context of the patient monitoring service that is being developed. It is derived from the HL7 FHIR standard for healthcare interoperability, based on the HAPI FHIR implementation of the FHIR specification in Java. The relevant OpenAPIs are described at: https://hapi.fhir.org/baseR4/swagger-ui/, while the FHIR standard is available at: https://hl7.org/fhir/. The descriptions from the data types used in this interface follow directly from the API documentation (https://hapifhir.io/hapi-fhir/apidocs/hapi-fhir-structures-r5/org/hl7/fhir/r5/model/package-summary.html).
Operations:

- void deleteObservation(string id) throws *AuthorizationException2*, *AuthenticationException2*
  - Effect: Delete an **Observation** instance with the specified id.
  - Sequence Diagrams: None
- void deletePatient(string id) throws *AuthorizationException2*, *AuthenticationException2*
  - Effect: Delete a **Patient** instance with the specfied id.
  - Sequence Diagrams: None
- void deleteRiskAssessment(string id) throws *AuthorizationException2*, *AuthenticationException2*

- Effect: Delete a **RiskAssessment** instance with the specified id.
- Sequence Diagrams: None

- **Observation** getObservation(string id) throws *AuthorizationException2*, *AuthenticationException2*
  - Effect: Retrieve an observation instance
  - Returns: The **Observation** with the specified identifier.
  - Sequence Diagrams: None
- **Patient** getPatient(string id) throws *AuthorizationException2*, *AuthenticationException2*
  - Effect: Retrieve a patient record with the specified identifier.
  - Returns: The **Patient** with the specified identifier.
  - Sequence Diagrams: None
- **RiskAssessment** getRiskAssessment(string id) throws *AuthorizationException2*
  - Effect: Retrieve a **RiskAssessment** instance with the specified id.
  - Returns: The **RiskAssessment** with the specified identifier.
  - Sequence Diagrams: None
- **Observation** saveObservation(**Observation** observation) throws *AuthorizationException2*, *AuthenticationException2*
  - Effect: Create a new **Observation** instance or in case an existing **Observation** instance exists with the same identifier, update this **Observation** instance.
  - Returns: The **Observation** object.
  - Sequence Diagrams: None
- **Patient** savePatient(**Patient** patient) throws *AuthorizationException2*
  - Effect: Create a new **Patient** instance or update an existing **Patient** instance.
  - Returns: The updated **Patient** object.
  - Sequence Diagrams: None
- **RiskAssessment** saveRiskAssessment(**RiskAssessment** riskAssessment) throws *AuthorizationException2*, *AuthenticationException2*
  - Effect: Create a new **RiskAssessment** instance or update an existing **RiskAssessment** instance.
  - Returns: The updated **RiskAssessment** object.
  - Sequence Diagrams: fig. D.5
- List<**Observation**> searchObservation(**Query**<**Date**> date, **Query**<String> dataAbsentReason, **Query**<**Patient**> subject, **Query**<String> valueConcept, **Query**<**Date**> valueDate, **Query**<**Observation**> derivedFrom, **Query**<**Patient**> patient, **Query**<**Quantity**> valueQuantity, **Query**<String> identifier, **Query**<String> performer, **Query**<String> method, **Query**<String> category, **Query**<String> device, **Query**<**ObservationStatus**> status) throws *AuthorizationException2*
  - Effect: Search for **Observation** instances matching the specified query criteria.
  - Returns: The list of **Observation**s matching the specified search criteria.
  - Sequence Diagrams: None
- List<**Patient**> searchPatient(**Query**<**Date**> birthDate, **Query**<Boolean> deceased, **Query**<String> addressState, **Query**<String> administrativeGender, **Query**<String> link, **Query**<String> language, **Query**<String> addressCountry, **Query**<**Date**> deathDate, **Query**<String> phonetic, **Query**<String> telecom, **Query**<String> addressCity, **Query**<String> email, **Query**<String> given, **Query**<String> identifier, **Query**<String> address, **Query**<String> generalPractitioner, **Query**<Boolean> active, **Query**<String> addressPostalCode, **Query**<String> phone, **Query**<String> organizationCustodian, **Query**<String> name, **Query**<String> family) throws *AuthorizationException2*, *AuthenticationException2*
  - Effect: Search for **Patient** instances matching the specified query criteria.
  - Returns: The list of **Patient**s matching the specified search criteria.
  - Sequence Diagrams: None
- List<**RiskAssessment**> searchRiskAssessment(**Query**<**Date**> date, **Query**<String> identifier, **Query**<String> performer, **Query**<String> method, **Query**<**Range**> probability, **Query**<**Patient**> subject, **Query**<String> condition, **Query**<**Patient**> patient, **Query**<**BigDecimal**> risk) throws *AuthenticationException2*, *AuthorizationException2*
  - Effect: Search for a **RiskAssessment** instance matching the provided query criteria.
  - Returns: The list of **RiskAssessment**s matching the specified search criteria.
  - Sequence Diagrams: None

Diagrams: None

E.3.25 HIS2PMS

Provided by: HISComChannel
Required by: InternalHISProcessor
Operations:

- List<byte> sendRequestToPMS()
  - Effect: method used to request something from PMS
  - Returns: encrypted request from HIS
  - Sequence Diagrams: figs. D.12 and D.14

Diagrams: None

E.3.26 HISDataAPI

Provided by: HISComChannel
Required by: GatewayOpsProcessor, HISComManager, PMSRequestProcessor, PMSSecurityManager, PatientManager
Operations:

- **Patient** getPatient(**PatientId** patientId)
  - Effect: Retrieve the instance of patient from HIS
  - Returns: patient instance from HIS
  - Sequence Diagrams: None
- **PatientRecord** getPatientRecord(**PatientId** patientId)
  - Effect: get patient record from HIS
  - Returns: **Patient** record
  - Sequence Diagrams: None
- void sendNotification(string userId, string content)
  - Effect: Send notification to HIS staff
  - Sequence Diagrams: fig. D.13
- void updatePatientStatus(**PatientId** patientId, **PatientStatus** status, **SensorDataPackage** sensorData)
  - Effect: update patient status. if available then also update the sensor data for infer the status
  - Sequence Diagrams: fig. D.5

Diagrams: None

E.3.27 HISGetInfo

Provided by: HISComManager
Required by: SecurityRequestProcessor
Operations:

- boolean existPatient(**PatientId** patientId)
  - Effect: During registration a patient to PMS, it is important to verify the patient existence in HIS.
  - Returns: whether the patient exist in HIS
  - Sequence Diagrams: fig. D.11

Diagrams: None

E.3.28 HISOperations

Provided by: HISOpsProcessor, PMSRequestProcessor, RequestScheduler
Required by: HISComChannel
Description : update patient info
Operations:

- **PatientStatus** getPatientStatus(**PatientId** patientId, string reqUserId)
  - Effect: Get the details of the patient status
  - Returns: patient's status
  - Sequence Diagrams: None
- void registerNoti(**PatientId** patientId, string requester)
  - Effect: add to noti list of a patient
  - Sequence Diagrams: None
- void requestConsultation(**PatientId** patientId)

  - Effect: request gateway to perform a consultation
  - Sequence Diagrams: fig. D.12
- List<string> requestRiskConfig(**PatientId** patientId)
  - Effect: request risk config options for a patient
  - Returns: config options
  - Sequence Diagrams: fig. D.14
- void updatePatientProfile(string assessment, string riskLevel, **PatientId** parameter)
  - Effect: update a patient assessment config or risk level
  - Sequence Diagrams: fig. D.14

Diagrams: None

E.3.29 IntegrityAPI

Provided by: IntegrityManager
Required by: SecurityRequestProcessor
Operations:

- Tupple <**SensorDataPackage**, List<byte>, List<byte>, List<string>> decrypt(List<byte> package)
  - Effect: decrypt the message using RSA
  - Returns: Tupple contains sensor data package, access token, signature, meta-data
  - Sequence Diagrams: fig. D.9
- List<byte> encrypt(List<byte> data)
  - Effect: Encryption using RSA
  - Returns: encrypted message in bytes
  - Sequence Diagrams: None
- boolean verifySignature(List<byte> signature)
  - Effect: verify the signature of the request
  - Returns: whether the request is not tamperred
  - Sequence Diagrams: fig. D.9

Diagrams: None

E.3.30 InternalSchedule

Provided by: PMSRequestScheduleSync
Required by: GatewayOpsProcessor, HISOpsProcessor, PatientManager, SensorDataSync
Operations:

- List<byte> getOpFromQueue()
  - Effect: request an operation from queue if exist
  - Returns: operation in bytes
  - Sequence Diagrams: None

Diagrams: None

E.3.31 InternalStatusAPI

Provided by: StatusMonitor
Required by: DataSync
Operations:

- string getInternalStatus()
  - Effect: Retrieve the status of the gateway subsystem
  - Returns: return the status of the gateway subsystem. Can be either normal or degraded
  - Sequence Diagrams: fig. D.7

Diagrams: None

E.3.32 JobMgmt

Provided by: RiskEstimationCombiner
Required by: RiskEstimationScheduler
Operations:

- void addJobSet(**RiskEstimationId** id, List<Tuple<**ClinicalModelJobId**, Double>> jobIds)

- Effect: Adds a set of identifiers for jobs belonging to a single risk estimation identified by id. The partial results of each clinical model computation job identified by an element in jobIds have to be combined in order to find the final result of the risk estimation as a whole.
- Sequence Diagrams: fig. D.1

Diagrams: None

E.3.33 KVStore

Provided by: KVStorageService
Required by: RiskEstimationProcessor, SecurityStorageManager, SensorDataRiskAssessment, eHealth Platform
Operations:

- **Object** retrieve(String key) throws *IOException*
  - Effect: Retrieve the object with the provided key from the Key-Value storage.
  - Returns: The object retrieved from storage.
  - Sequence Diagrams: fig. D.2
- void store(String key, **Object** value) throws *IOException*
  - Effect: Store the provided object in the Key-Value storage under the provided key.
  - Sequence Diagrams: fig. D.2

Diagrams: None

E.3.34 LaunchRiskEstimation

Provided by: RiskEstimationScheduler
Required by: GatewayOpsProcessor, PMSRequestProcessor
Operations:

- void launchRiskEstimation(**PatientId** patientId, **SensorDataPackage** newSensorData, **Timestamp** receivedAt)
  - Effect: The RiskEstimationScheduler will fetch the clinical models and their configurations associated to the patient identified by patientId from storage using the ClinicalModelCache, notify the RiskEstimationCombiner of the different jobs that will be performed for a single risk estimation and schedule the individual jobs in its queue. In normal modus, queued jobs are returned in FIFO order. In overload modus, the system switches to dynamic priority: earliest deadline first and enqueues jobs of patient with a high risk level with an earlier deadline (2 instead of 5 minutes) to prioritize them over patients with lower risk levels. The **SensorDataPackage** newSensorData is passed because its arrival triggered the risk estimation. A risk level is estimated based on the computation of these clinical models. For the computation of the clinical models, the given sensor data newSensorData (which is the new sensor data that was received) is used and other required data is fetched from the respective databases if needed. The given time-stamp receivedAt is used in order to avoid fetching the new sensor data from the database. This time-stamp represents the time at which the new sensor data was received.
  - Sequence Diagrams: figs. D.1 and D.5

Diagrams: None

E.3.35 LWCheckAPI

Provided by: LightweightCheck
Required by: DataSync
Operations:

- **SensorDataPackage** isEmergency()
  - Effect: The method apply lightweight ML model to check whether the sensor data indicate an Emergency
  - Returns: whether the data indicates an emergency
  - Sequence Diagrams: None

Diagrams: None

E.3.36 MLaaSAPI

Provided by: MLaaSLibrary

Required by: MLModelUpdateProcessor
Description : This API provides a number of operations for managing the models deployed on the MLaaS cloud.
Operations:

- **Credential** authenticate() throws *InvalidAPIKeyException*
  - Effect: Construct a **Credential** object for authenticating requests with the MLaaS service provider. This operation uses the previously configured API key to construct the **Credential** object. If such an API key is not configured, an exception will be thrown.
  - Returns: Authentication token for subsequent requests.
  - Sequence Diagrams: figs. D.3 and D.4
- void delete(**MLModel** model, **Credential** token) throws *AuthorizationException*
  - Effect: Delete a model from the online MLaaS service.
  - Sequence Diagrams: None
- **MLModelDescription** getModelDescription(List<**MLModelDescription**> descriptions, String modelId)
  - Effect: This method retrieves the correct **MLModelDescription** that corresponds with the **MLModel** identified by the provided modelId.
  - Returns: The **MLModelDescription** corresponding with the **MLModel** identified by the modelId.
  - Sequence Diagrams: None
- String getModelId(**MLModelDescription** description)
  - Effect: Extract the modelId from the **MLModelDescription**.
  - Returns: String with the modelId of the **MLModel** specified in the **MLModelDescription**.
  - Sequence Diagrams: fig. D.3
- void register(**MLModel** model, **Credential** token) throws *AuthorizationException*
  - Effect: Register a new model with the MLaaS service.
  - Sequence Diagrams: None
- void setAPIKey(String apiKey) throws *AuthenticationException*
  - Effect: Set the credentials for authenticating with the ML-as-a-Service provider.
  - Sequence Diagrams: None

Diagrams: None

E.3.37 MLaaSService

Provided by: MLaaS
Required by: MLModelManager, MLModelUpdateProcessor, eHealth Platform
Description : This interface offiers operations for interacting with MLaaS service providers.
Operations:

- List<**MLModelUpdate**> fetchModelUpdates(**Credential** credential, **MLModelDescription** model)
  - Effect: Fetch a list of model updates to apply to the local **MLModel**.
  - Returns: The list of MLModelUpdates to apply locally.
  - Sequence Diagrams: fig. D.4
- List<**MLModelDescription**> getAvailableModels(**Credential** credential)
  - Effect: Get the list of available ML models to use for making predictions on incoming sensor data.
  - Returns: The list of available MLModels.
  - Sequence Diagrams: fig. D.3
- void pushModelUpdates(**Credential** credential, **MLModelDescription** model, List<**MLModelUpdate**> updates)
  - Effect: Send a list of MLModelUpdates to the online service.
  - Sequence Diagrams: None
- **MLModel** retrieveModel(**Credential** credential, **MLModelDescription** model)
  - Effect: Retrieve a specific **MLModel** corresponding with the provided **MLModelDescription**.
  - Returns: The requested **MLModel**.
  - Sequence Diagrams: fig. D.3

Diagrams: None

E.3.38 MLAPI

Provided by: MLLibrary

Required by: ModelUpdater, SensorDataRiskAssessment
Description : This is the API provided by the MLLibrary to use **MLModel**s locally for making predictions and processing updates to these models.
Operations:

- **MLModel** applyUpdate(**MLModel** model, **MLModelUpdate** update)
  - Effect: Applies the provided **MLModelUpdate** on the provided **MLModel** and returns the resulting **MLModel**.
  - Returns: The resulting **MLModel** after applying the requested **MLModelUpdate**.
  - Sequence Diagrams: fig. D.4
- **Prediction** predict(**SensorDataPackage** sensorData)
  - Effect: Make a prediction with the active **MLModel** on the provided sensordata.
  - Returns: The prediction based on the provided sensor data.
  - Sequence Diagrams: fig. D.2
- void verify(**MLModel** model) throws *InvalidMLModelException*
  - Effect: Verifies that a provided **MLModel** is working correctly after applying **MLModelUpdate**s.
  - Sequence Diagrams: fig. D.4

Diagrams: None

E.3.39 MLModelMgmt

Provided by: MLModelManager, MLModelStorageManager
Required by: MLModelRetrieval, RiskEstimationProcessor
Description : The interface provides operations for managing different **MLModel**s locally, querying available models for making predictions on newly incoming sensor data.
Operations:

- **MLModel** fetchModel(String modelId) throws *NoSuchMLModelException*
  - Effect: Fetch the **MLModel** corresponding with the provided modelId.
  - Returns: The requested **MLModel**.
  - Sequence Diagrams: fig. D.2
- List<String> getAvailableModels()
  - Effect: Get the list of available **MLModel**s that can be used for making **Prediction**s locally.
  - Returns: The list of identifiers of the available **MLModel**s that can be used.
  - Sequence Diagrams: None
- void synchronizeModels()
  - Effect: Synchronize the list of locally-stored **MLModel**s by checking for newly-available **MLModel**s that are not stored in the local MLModelRepository and retrieving these **MLModel**s from the MLaaS provider.
  - Sequence Diagrams: fig. D.3
- void updateModel(String modelId) throws *NoSuchMLModelException*
  - Effect: Update the model with the specified id by fetching **MLModelUpdate**s and applying them to the model.
  - Sequence Diagrams: fig. D.4

Diagrams: None

E.3.40 MLModelStorage

Provided by: MLModelRepository
Required by: MLModelStorageManager, MLModelUpdateProcessor
Operations:

- boolean isAvailable(String modelId)
  - Effect: Check whether an **MLModel** (identified by the modelId) is available in the local MLModelRepository.
  - Returns: Boolean indicating if the specified model is present in the repository.
  - Sequence Diagrams: figs. D.3 and D.4
- **MLModel** retrieveModel(String modelId)
  - Effect: Retrieve the **MLModel** with the specified id.
  - Returns: The requested **MLModel**.
  - Sequence Diagrams: fig. D.4

- void storeModel(**MLModel** model)
  - Effect: Store an **MLModel** in the local MLModelRepository
  - Sequence Diagrams: figs. D.3 and D.4

Diagrams: None

E.3.41 MLModelSync

Provided by: MLModelUpdateProcessor
Required by: MLModelStorageManager
Operations:

- void fetchNewMLModels()
  - Effect: Fetches newly available **MLModel**s and store them in the MLModelRepository.
  - Sequence Diagrams: fig. D.3
- void processMLModelUpdates(String modelId) throws *NoSuchMLModelException*
  - Effect: Retrieve and process the updates for the specified **MLModel** and store the updated **MLModel** in the MLModelRepository.
  - Sequence Diagrams: fig. D.4

Diagrams: None

E.3.42 MLModelUpdateMgmt

Provided by: ModelUpdater
Required by: MLModelUpdateProcessor
Operations:

- **MLModel** processMLModelUpdates(**MLModel** model, List<**MLModelUpdate**> updates)
  - Effect: Requests the ModelUpdater to apply the list of **MLModelUpdate**s on the provided **MLModel**.
  - Returns: The **MLModel** resulting from the application of the list of **MLModelUpdate**s on the initial **MLModel**.
  - Sequence Diagrams: fig. D.4

Diagrams: None

E.3.43 ModeAPI

Provided by: DataSync
Required by: StatusMonitor
Operations:

- string ping()
  - Effect: ping to check for issue
  - Returns: return a status string
  - Sequence Diagrams: None
- void switchMode(string mode)
  - Effect: switch between normal mode or degraded mode. When switch back to normal mode from degraded, perform a mass-update from the sensor data in the internal storage
  - Sequence Diagrams: None

Diagrams: None

E.3.44 OtherDataMgmt

Provided by: PMSRequestProcessor, PatientManager
Required by: ClinicalJobCreator, ClinicalModelCache, RiskEstimationCombiner, RiskEstimationScheduler
Operations:

- **PatientStatus** getPatientStatus(**PatientId** patientId) throws *NoSuchPatientException*
  - Effect: Fetch and return the status of the patient identified by the patientId.
  - Returns: The status of the patient.
  - Sequence Diagrams: None

- void setEstimatedPatientStatus(**PatientId** patientId, **PatientStatus** estimatedStatus, **Timestamp** estimationTime) throws *NoSuchPatientException*
  - Effect: Update the patient status estimation of the patient identified by patientId to the given value estimatedStatus and update the time of estimation to estimationTime. The time of estimation is the time at which the corresponding estimation job for this patient was completed.
  - Sequence Diagrams: None

Diagrams: None

E.3.45 PatientRecordMgmt

Provided by: PMSRequestProcessor, PatientManager
Required by: RiskEstimationCombiner
Operations:

- **PatientRecord** getPatientRecord(**PatientId** patientId) throws *NoSuchPatientRecordException*
  - Effect: This will fetch the EHR record of the patient with given id from the HIS and return it. If the HIS is not available, an older (cached) copy of the patient record is returned if possible.
  - Returns: The requested patient record.
  - Sequence Diagrams: None

Diagrams: None

E.3.46 PMSComAPI

Provided by: BackupComChannel, PMSComChannel
Required by: DataSync
Operations:

- List<byte> getPublicKey()
  - Effect: retrieve public key from PMS
  - Returns: public key
  - Sequence Diagrams: fig. D.7
- void sendSensorDataToPMS(**PatientId** patientId, **SensorDataPackage** sensorData, List<string> metadata)
  - Effect: in normal mode, forward the sensor data to pms backend
  - Sequence Diagrams: fig. D.7

Diagrams: None

E.3.47 PMSComHeartBeat

Provided by: PMSComHealthMonitor
Required by: GatewayComChannel, GatewayFaultDectector, HISComChannel
Operations:

- void reportHeartBeat(void id)
  - Effect: report heartbeat to the monitor component
  - Sequence Diagrams: None

Diagrams: None

E.3.48 Results

Provided by: RiskEstimationCombiner
Required by: JobProcessor, RiskEstimationProcessor
Operations:

- void addClinicalModelJobResults(**ClinicalModelJobId** jobId, **ClinicalModelJobResult** results)
  - Effect: Sends the result of the performed clinical model computation (identified by the jobId) to the RiskEstimationCombiner for combining with the other partial results belonging to the same risk estimation.
  - Sequence Diagrams: figs. D.2 and D.6

Diagrams: None

E.3.49 SchedulerAPI

Provided by: RequestScheduler
Required by: PMSRequestProcessor, PMSRequestScheduleSync
Operations:

- List<byte> getOpFromQueue(string moduleType)
  - Effect: get operation from Queue providing the component id so that the queue know which component can process which request
  - Returns: operation in bytes
  - Sequence Diagrams: figs. D.8, D.12 and D.14

Diagrams: None

E.3.50 SecLogging

Provided by: SecurityLogger
Required by: SecurityRequestProcessor
Operations:

- string addLog(string log)
  - Effect: adding security log for all request containing user id, scope, etc.
  - Returns: JSON format of the log
  - Sequence Diagrams: figs. D.9 and D.11

Diagrams: None

E.3.51 SecStorageAPI

Provided by: SecurityStorageManager
Required by: AccessControlManager, AuthManager, IntegrityManager, SecurityLogger
Operations:

- void addOrUpdateUserInfo(string user)
  - Effect: register or update user info to the user storage
  - Sequence Diagrams: None
- string getUserInfo()
  - Effect: get user information (username, password, role, access scope).
  - Returns: JSON type of user info
  - Sequence Diagrams: None

Diagrams: None

E.3.52 SecurityAPI

Provided by: PMSSecurityManager, SecurityRequestProcessor
Required by: GatewayComChannel, HISComChannel
Operations:

- boolean accessControlCheck(string userId, List<String> accessScope)
  - Effect: Verify whether the request have the right to use the data it requires
  - Returns: return whether the access control is valid
  - Sequence Diagrams: None
- boolean applySecuirtyCheck(List<byte> request)
  - Effect: Apply security check to request incl. verify signature, access token, etc.
  - Returns: whether the check pass all contraints
  - Sequence Diagrams: figs. D.12 and D.14
- string authenticateUser(string userId, string password, string accessToken)
  - Effect: Authenticate user request
  - Returns: Can be either the access token or a error message incase of failling authentication
  - Sequence Diagrams: fig. D.10
- **SensorDataPackage** decryptAndVerifyPackage(List<byte> encryptedPackage)
  - Effect: decrypt the sensor data package and verify its signature (as well as check the sensor data and gateway id to see whether they registered for the same user
  - Returns: decrypted package if it passes the security check. Otherwise throw SecurityException
  - Sequence Diagrams: fig. D.8

- List<byte> getSecurityKeys()
  - Effect: A method to get the public key and the symetric key for signing and ecrypting messages
  - Returns: return the public key and the symetric key that can be used for signing and encryption
  - Sequence Diagrams: None
- string registerPatient(**PatientId** patientId, string username, string password, string deviceId, List<String> sensorsId)
  - Effect: Register patient as user for PMS app. This process can only be done by PMS admin or PMS user with access control "register_new_patient"
  - Returns: JSON user from storage or Exception
  - Sequence Diagrams: None
- void registerStaff(string username, string staffId, string password, List<String> accessScope)
  - Effect: A method for registering a new staff (either from PMS or HIS). The registration is done by PMS admin or the user with access control "register_new_staff"
  - Sequence Diagrams: None
- boolean userDeviceVerification(string userId, List<string> sensorsId, string deviceId)
  - Effect: Verify whether the user use the correct gateway, wearable unit or not
  - Returns: whether the user use the correct gateway, wearable unit or not
  - Sequence Diagrams: None

Diagrams: None

E.3.53 sendToGateway

Provided by: GatewayApplication, PMSComChannel
Required by: GatewayComChannel
Operations:

- void sendNotification(string notification)
  - Effect: send notification to gateway
  - Sequence Diagrams: fig. D.13

Diagrams: None

E.3.54 SensorDataAnalysis

Provided by: SensorDataRiskAssessment
Required by: JobProcessor
Operations:

- void performModelPrediction(**ClinicalModelJob** clinicalModelJob, **MLModel** model)
  - Effect: Perform a prediction using the provided **MLModel** and the data provided in the clinicalModelJob definition. Afterwards the results are submitted for combination by the RiskEstimationCombiner.
  - Sequence Diagrams: fig. D.2

Diagrams: None

E.3.55 SensorDataMgmt

Provided by: PMSRequestProcessor, SensorDataSync
Required by: ClinicalJobCreator, RiskEstimationScheduler
Operations:

- void addSensorData(**PatientId** patientId, **SensorDataPackage** sensorData, **Timestamp** receivedAt)
  - Effect: Store the given sensor data and meta-data.
  - Sequence Diagrams: None
- Map<**Timestamp**, **SensorDataPackage**> getAllSensorDataOfPatient(**PatientId** patientId)
  - Effect: This will fetch and return all sensor data belonging to the patient identified by patientId
  - Returns: The sensor data and the timestamp of their arrival in the system.
  - Sequence Diagrams: None
- Map<**Timestamp**, **SensorDataPackage**> getAllSensorDataOfPatientBefore(**PatientId** patientId, **Timestamp** stopTime)
  - Effect: Fetch and return all sensor data belonging to the patient identified by patientId which was received before the specified time stopTime.

- Returns: The sensor data and the timestamp of their arrival in the system.
- Sequence Diagrams: None

Diagrams: None

E.3.56 SensorStorageAPI

Provided by: SensorDataDistributer
Required by: HISOpsProcessor, PMSRequestProcessor, SensorDataSync
Operations:

- void addNewSensorData(**PatientId** patientId, **SensorDataPackage** sensorData, List<string> metadata)
  - Effect: adding new sensor data to the storage
  - Sequence Diagrams: fig. D.8
- List<**SensorDataPackage**> getSensorDataForPatient(**PatientId** patientId, string filter)
  - Effect: get sensor data for a patient with additional filter
  - Returns: Return a list of sensor data
  - Sequence Diagrams: None

Diagrams: None

E.3.57 ShardSensorStorageAPI

Provided by: DegradedSensorDataStorage, ShardedSensorStorage
Required by: SensorDataDistributer
Operations:

- void addSensorData(**PatientId** patientId, **SensorDataPackage** sensorData, List<string> metadata)
  - Effect: add sensor data to the appropriate datashard (based on risk level)
  - Sequence Diagrams: fig. D.8

Diagrams: None

E.3.58 SubsysStorageAPI

Provided by: SubsysStateStorage
Required by: PMSComHealthMonitor
Operations:

- void addSubsystemState(string id, long timestamp, List<byte> state)
  - Effect: adding the functional state of a subsystem in the communication channel for for later revertion in case of failure
  - Sequence Diagrams: None
- List<byte> getLatestSubsystemState(string id)
  - Effect: retreive the latest functional state of a subsystem for revertion
  - Returns: bytes encoded subsystem state
  - Sequence Diagrams: None

Diagrams: None

E.3.59 SyncAPI

Provided by: DataSync
Required by: WearableDataCapturing
Operations:

- void processSensorData(string deviceId, List<string> sensorsId, **SensorDataPackage** sensorData)
  - Effect: receive and process the sensor data
  - Sequence Diagrams: fig. D.7

Diagrams: None

E.3.60 UserStorageAPI

Provided by: UserStorage
Required by: HISOpsProcessor, PMSRequestProcessor, PMSSecurityManager, PatientManager, SecurityStorageManager
Operations:

- List<string> getNotiList()
  - Effect: get a list of user id that interested in this patient
  - Returns: return a list of user id that interested in this patient
  - Sequence Diagrams: fig. D.13
- string getUserInfo(string userId)
  - Effect: a general function to get all information regarding a user in PMS system (excluding heavy weight info such as records, sensor data, etc.).
  - Returns: Json string of user info including their name, hash password, id, registered devices, registered sensors, etc.
  - Sequence Diagrams: None
- void updateUserInfo(string userId, string userInfo)
  - Effect: update a user infomation
  - Sequence Diagrams: None

Diagrams: None

E.3.61 WDCPingAPI

Provided by: WearableDataCapturing
Required by: StatusMonitor
Operations:

- string ping()
  - Effect: ping to check for issue
  - Returns: return a status string
  - Sequence Diagrams: None

Diagrams: None

E.4 Nodes

E.4.1 eHealthPlatform

Responsibility: Container node representing the deployment context boundary for the eHealth Platform.
Visible on diagrams: fig. C.1

E.4.2 HISNode

Responsibility: HIS server
Visible on diagrams: figs. C.1, C.2, C.3 and C.4

E.4.3 HISStaffNode

Responsibility: HIS staff application such as phone, laptop inside the hospital
Visible on diagrams: figs. C.2, C.3 and C.4

E.4.4 MLaaS Conn (Pilot Deployment)

Responsibility: This node is responsible for the connection with the MLaaS Service.
Visible on diagrams: fig. C.3

E.4.5 MLaaS Service (Dev Deployment)

Responsibility: This represents the MLaaS cloud service.
Visible on diagrams: fig. C.4

E.4.6 MLaaS Service (Pilot Deployment)

Responsibility: This represents the cloud MLaaS service.
Visible on diagrams: fig. C.3

E.4.7 MLaaSCloud

Responsibility: Cloud infrastructure of the `MLaaS` Service provider.
Visible on diagrams: fig. C.1

E.4.8 MLaaSConnectorNode

Responsibility: Node responsible for handling connections with external `MLaaS` providers.
Visible on diagrams: figs. C.1 and C.2

E.4.9 PMS backbone

Responsibility: Temporary node that requires further decomposition.
Visible on diagrams: figs. C.2, C.3 and C.4

E.4.10 PMSComNode

Responsibility: A communication node, for availability issues, there can be more than 1 channel (the other act as stand-by) and actively replace the main channel if there is a fault
Visible on diagrams: figs. C.1, C.2, C.3 and C.4

E.4.11 RiskEstimationManagementNode (Pilot Deployment)

Responsibility: The RiskEstimationManagement node hosts the creation, scheduling and combining of the clinical jobs.
Visible on diagrams: fig. C.3

E.4.12 RiskEstimationManagmentNode (Dev Deployment)

Responsibility: The development deployment is a single RiskEstimationManagement node that contains all other functionality, job creation, scheduling, and combination.
Visible on diagrams: fig. C.4

E.4.13 RiskEstimationMgmtNode

Responsibility: The RiskEstimationMgmtNode keeps track of the risk estimation jobs, and schedules and combines the results.
Visible on diagrams: fig. C.2

E.4.14 RiskEstimationProcessorNode

Responsibility: Multiple `RiskEstimationProcessor` nodes enable the processing of multiple sensorData inputs in parallel.
Visible on diagrams: fig. C.2

E.4.15 RiskEstimationProcessorNode (Dev Deployment)

Responsibility: The development deployment only uses two RiskEstimationProcessorNodes to run **ClinicalModelJob**s in parallel.
Visible on diagrams: fig. C.4

E.4.16 RiskEstimationProcessorNode (Pilot Deployment)

Responsibility: There are five instances of the RiskEstimationProcessorNode to enable the processing of multiple **ClinicalModelJob**s in parallel.
Visible on diagrams: fig. C.3

E.4.17 SensorDataStorageNode

Responsibility: Sharded SensorData stored contain 3 shared storage that represent base on the risk level of the storage
Visible on diagrams: figs. C.2, C.3 and C.4

E.4.18 SensorDataStorageReplicaNode

Responsibility: Node deploys passive replication of the sharded sensor storage which will act as the primary storage in case of failure
Visible on diagrams: figs. C.2, C.3 and C.4

E.4.19 TODO Node (Pilot Deployment)

Responsibility: This TODO node for the pilot deployment contains the OtherFunctionality of the system that has not been worked out in detail yet.
Visible on diagrams: None

E.4.20 UserNode

Responsibility: User node containing the running processes of gateways
Visible on diagrams: figs. C.2, C.3 and C.4

E.5 Exceptions

- *AuthenticationException* (△*Exception*):
  Signals a problem with the credentials used for authentication. These may be invalid or expired.
- *AuthenticationException2* (△*SecurityException*):
  Missing or invalid credentials.
- *AuthorizationException* (△*Exception*):
  Signals that the principal has insufficient access rights to perform the operation.
- *AuthorizationException2* (△*SecurityException*):
  The requestor is not authorized to perform the requested action.
- *Exception*:
  Subtypes: △*AuthenticationException*, △*AuthorizationException*, △*InvalidAPIKeyException*, △*InvalidMLModelException*, △*IOException*, △**??**, △*NoSuchMLModelException*, △*NoSuchPatientException*, △*NoSuchPatientRecordException*
  Base exception class.
- *InvalidAPIKeyException* (△*Exception*):
  Thrown when no correctly formatted API key is configured for constructing authentication Credential|s.
- *InvalidMLModelException* (△*Exception*):
  Thrown when a provided MLModel is invalid.
- *IOException* (△*Exception*):
  Thrown when an IO operation fails.
- *NoSuchMLModelException* (△*Exception*):
  Thrown when the requested MLModel cannot be found.
- *NoSuchPatientException* (△*Exception*):
  Thrown if no patient with the specified identifier exists.
- *NoSuchPatientRecordException* (△*Exception*):
  Thrown if no patient record for the patient with the given identifier exists in the HIS.
- *SecurityException*:
  Subtypes: △*AuthenticationException2*, △*AuthorizationException2*
  Parent class for security exceptions.

E.6 Data types

- **Address**:
  Attributes: **AddressUse** use, **AddressType** type, String text, List<String> line, String city, String district, String state, String postalCode, String country, org.hl7.fhir.r5.model.Period period, long serialVersionUID
  Base StructureDefinition for Address Type: An address expressed using postal conventions (as opposed to GPS or other location definition formats). This data type may be used to convey addresses for use in delivering mail as well as for visiting locations which might not be valid for mail delivery. There are a variety of postal address formats defined around the world. The line attribute contains contains the house number, apartment number, street name, street direction, P.O. The text attribute specifies the entire address as it should be displayed e.g. on a postal label. This may be provided instead of or as well as the specific parts.
- **AddressType**:
  Enum: POSTAL, PHYSICAL, BOTH, NULL.
  Distinguishes between physical addresses (those you can visit) and mailing addresses (e.g. PO Boxes and care-of addresses). Most addresses are both.
- **AddressUse**:
  Enum: HOME, WORK, TEMP, OLD, BILLING, NULL.
  The purpose of the address.
- **Annotation**:
  Attributes: **Date** time, String text, long serialVersionUID, String author
  Base StructureDefinition for Annotation Type: A text note which also contains information about who made the statement and when.
- **BigDecimal**:
  Immutable, arbitrary-precision signed decimal numbers. A BigDecimal consists of an arbitrary precision integer unscaled value and a 32-bit integer scale
- **ClinicalModel**:
  Attributes: String id, String name, String description, List<String> requiredParameters, List<String> optionalParameters, List<String> applicableMLModelIds, boolean requiresHistoricalSensorData
  A clinical model. This is model has an id, name, description. It contains a list of parameters for using the model and a set of MLModelIds to specify which types of MLModels can be used in the context of the ClinicalModel.
- **ClinicalModelConfiguration**:
  Attributes: Map<String, String> modelParameters
  The configuration for a clinical model. Represented as a set key/value pair for the parameters of the clinical model.
- **ClinicalModelJob**:
  Attributes: **ClinicalModel** clinicalModel, **ClinicalModelJobId** jobId, **SensorDataPackage** sensorData, **PatientId** patientId, **ClinicalModelConfiguration** clinicalModelConfiguration, **PatientRecord** patientRecord, Map<**Timestamp**, **SensorDataPackage**> historicalSensorData
  A single job for the computation of a clinical model. This contains a ClinicalModelJobId, ClinicalModelId and PatientId respectively identifying the job itself, the corresponding clinical model and the patient. If the corresponding risk estimation the job belongs to is triggered by the arrival of a sensor data package, this sensor data is also contained.
- **ClinicalModelJobId**:
  A piece of data uniquely identifying a certain clinical model job in the system. This architecture does not specify the exact format of this identifier, but possibilities are a long integer, a string, a URI etc.
- **ClinicalModelJobResult**:
  The result of the computation of a clinical model containing the estimated risk level for the patient. The resulting risk score for ClinicalModels are on the same unit scale so the the results of multiple ClinicalModels can be combined.
- **CodeableConcept**:
  Attributes: List<**Coding**> coding, String text, long serialVersionUID
  Base StructureDefinition for CodeableConcept Type: A concept that may be defined by a formal reference to a terminology or ontology or may be provided by text.

- **CodeType**:
  Attributes: long serialVersionUID, String system, String value
  Primitive type "code" in FHIR, when not bound to an enumerated list of codes
- **Coding**:
  Attributes: String system, String version, org.hl7.fhir.r5.model.CodeType code, String display, boolean userSelected, long serialVersionUID
  Base StructureDefinition for Coding Type: A reference to a code defined by a terminology system.
- **ContactComponent**:
  Attributes: List<**CodeableConcept**> relationship, String name, List< org.hl7.fhir.r5.model.ContactPoint > telecom, org.hl7.fhir.r5.model.Address address, String administrativeGender, String organization, org.hl7.fhir.r5.model.Period period, long serialVersionUID
  A contact party (e.g. guardian, partner, friend) for the patient. The relationship attribute species the nature of the relationship between the patient and the contact person. It is a CodeableConcept (which can be formal reference to terminology or an ontology). For example, in the system `http://hl7.org/fhir/ValueSet/relatedperson-relationshiptype`, it can be coded as CHILD for a child, or MGRMTH for maternal grandmother, etc.
- **ContactPoint**:
  Attributes: **ContactPointSystem** system, String value, **ContactPointUse** use, int rank, long serialVersionUID, **Period** period
  Base StructureDefinition for ContactPoint Type: Details for all kinds of technology mediated contact points for a person or organization, including telephone, email, etc. The value attribute contains the actual contact point details, in a form that is meaningful to the designated communication system (i.e. phone number or email address). Rank specifies a preferred order in which to use a set of contacts. ContactPoints with lower rank values are more preferred than those with higher rank values. Period specifies the tiime period when the contact point was/is in use.
- **ContactPointSystem**:
  Enum: PHONE, FAX, EMAIL, PAGER, URL, SMS, OTHER, NULL.
  Enum denotying the type of communication channel for the ContactPoint.
- **ContactPointUse**:
  Enum: HOME, WORK, TEMP, OLD, MOBILE, NULL.
  Enum denoting the usage type of the ContactPoint.
- **Credential**:
  Authentication object for making authenticated requests to the MLaaS service provider.
- **Date**:
  The class Date represents a specific instant in time, with millisecond precision.
- **Identifier**:
  Attributes: **IdentifierUse** use, String type, String system, String value, org.hl7.fhir.r5.model.Period period, String assigner, long serialVersionUID
  Base StructureDefinition for Identifier Type: An identifier - identifies some entity uniquely and unambiguously. Typically this is used for business identifiers. The type is a a coded type for the identifier that can be used to determine which identifier to use for a specific purpose. The system attribute establishes the namespace for the value - that is, a URL that describes a set values that are unique. The value attribute specifies the portion of the identifier typically relevant to the user and which is unique within the context of the system. The period specifies the time period during which identifier is/was valid for use. The assigner specifies the organization that issued/manages the identifier.
- **IdentifierUse**:
  Enum: USUAL, OFFICIAL, TEMP, SECONDARY, OLD, NULL.
  Enum denoting the usage type of a particular Identifier.
- **MLModel**:
  Attributes: String modelId, **MLModelDescription** modelDescription
  Machine learning model for making predictions based on sensordata. The actual internal encoding of the model itself is left open.
- **MLModelDescription**:

  Attributes: String uuid, String name, String description, String providerName
  Object providing the MLModel meta data. It does not contain the model itself.

- **MLModelUpdate**:
  Attributes: **MLModelDescription** modelDescription
  Represents a model update that can be applied on an MLModel. These model updates enable exchanging changes to the MLModel between different parties without sending the actual MLModel to the other parties.
- **NoKeyException** (⇧*Exception*):
  Throw when gateway doesn't have the key to do security operations
- **Object**:
  Root class for objects that can be stored in the Key-Value storage service. Data types that need to be saved need to inherit from this class.
- **Observation**:
  Attributes: List<**Identifier**> identifier, List<**Identifier**> partOf, **ObservationStatus** status, List<**CodeableConcept**> category, **CodeableConcept** code, **Patient** subject, **Period** effective, **Date** issued, List<**Identifier**> performer, String value, **CodeableConcept** dataAbsentReason, **CodeableConcept** interpretation, List<**Annotation**> note, **CodeableConcept** bodySite, **CodeableConcept** method, String device, List<**Observation**> hasMember, List<**Observation**> derivedFrom, long serialVersionUID, **ObservationReferenceRangeComponent** referenceRange, **ObservationComponentComponent** component
  Measurements and simple assertions made about a patient, device or other subject. The partOf attribute specifies the larger event of which this particular Observation is a component or step. For example, an observation as part of a procedure.
  The category specifies a code that classifies the general type of observation being made. For example, in the system `http://terminology.hl7.org/CodeSystem/observation-category`, vital-signs is used for clinical observations such as boold pressure, heart rate, etc.
  The code attribute describes what was observed. Sometimes this is called the observation 'name'. For example, in the system `http://loinc.org`, code '8867-4' is used for expressing the heart rate, '85354-9' for a blood pressure panel (which must have components '8480-6' and '8462-4' for, respectively, systolic and diastolic pressure.
  The performer is who was responsible for asserting the observed value as 'true'. This can refer to a Patient, Practitioner, organization, etc. The value is the information determined as a result of making the observation, if the information has a simple value.
  The dataAbsentReason provides a reason why the expected value in the element Observation.value[x] is missing. For example, in the system `http://terminology.hl7.org/CodeSystem/data-absent-reason`, 'not-applicable'.
  The interpretation is a categorical assessment of an observation value. For example, high, low, normal. For example, in the system `http://terminology.hl7.org/CodeSystem/v3-ObservationInterpretation`, 'H' for high, or 'ND' for not detected.
  The bodySite indicates the site on the subject's body where the observation was made (i.e. the target site). For example, in the system `http://snomed.info/sct`, code '344001' indicates the ankle.
  The method indicates the mechanism used to perform the observation. For example, in the system `http://snomed.info/sct`, code '46973005' indicates blood pressure taking procedure.
  The device attribute specifies an identifier of the device used to generate the observation data.
- **ObservationComponentComponent**:
  Attributes: String code, **Range** value, String dataAbsentReason, List<String> interpretation, long serialVersionUID, **ObservationReferenceRangeComponent** referenceRange
  Some observations have multiple component observations. These component observations are expressed as separate code value pairs that share the same attributes. Examples include systolic and diastolic component observations for blood pressure measurement and multiple component observations for genetics observations. The code describes what was observed. Sometimes this is called the observation 'code'. For example, in the system `http://loinc.org`, '8480-6' is used for the systolic component of the blood pressure.,
  The dataAbsentReason provides a reason why the expected value in the element Observation.component.value[x] is missing. For example, in the system `http://terminology.hl7.org/CodeSystem/data-absent-reason`, 'not-applicable'.
  The interpretation provides a categorical assessment of an observation value. For example,

high, low, normal. For example, in the system `http://terminology.hl7.org/CodeSystem/v3-ObservationInterpretation`, 'H' for high, or 'ND' for not detected.

- **ObservationReferenceRangeComponent**:
  Attributes: org.hl7.fhir.r5.model.Quantity low, org.hl7.fhir.r5.model.Quantity high, **CodeableConcept** type, List<**CodeableConcept**> appliesTo, org.hl7.fhir.r5.model.Range age, String text, long serialVersionUID
  Guidance on how to interpret the value by comparison to a normal or recommended range. Multiple reference ranges are interpreted as an "OR". In other words, to represent two distinct target populations, two 'referenceRange' elements would be used.
  The appliesTo specifies codes to indicate the target population this reference range applies to. For example, in the system `http://snomed.info/sct`, code '248152002' indicates female.
  The type specifies codes to indicate the what part of the targeted reference population it applies to. For example, in the system `http://terminology.hl7.org/CodeSystem/referencerange-meaning`, 'normal' for the 95% normal range, or 'recommended' for the recommended range by a relevant professional body.
  The text attribute contains a text based reference range in an observation which may be used when a quantitative range is not appropriate for an observation.
- **ObservationStatus**:
  Enum: REGISTERED, PRELIMINARY, FINAL, AMENDED, CORRECTED, CANCELLED, ENTEREDINERROR, UNKNOWN, NULL.
  The status of the observation result value or the RiskAssessment.
- **Patient**:
  Attributes: List<**Identifier**> identifier, boolean active, List<String> name, String administrativeGender, **Date** birthDate, **Date** deceased, List<**Address**> address, boolean multipleBirth, List<**Identifier**> generalPractitioner, **Identifier** managingOrganization, long serialVersionUID, **ContactComponent** contact
  Demographics and other administrative information about an individual or animal receiving care or other health-related services.
- **PatientId**:
  A piece of data uniquely identifying a patient in the system. This architecture does not specify the exact format of this identifier, but possibilities are a long integer, a string, a URI, etc.
- **PatientRecord**:
  The structured contents of the EHR record of a certain patient, including medication history, recent treatments, allergies, etc.
- **PatientStatus**:
  The status of a patient, i.e., their risk level.
- **Period**:
  Attributes: **Date** start, **Date** end, long serialVersionUID
  Base StructureDefinition for Period Type: A time period defined by a start and end date and optionally time.
- **Prediction**:
  A prediction from the MLModel.
- **Quantity**:
  Attributes: **BigDecimal** value, String unit, String system, **CodeType** code, long serialVersionUID
  Base StructureDefinition for Quantity Type: A measured amount (or an amount that can potentially be measured). Note that measured amounts include amounts that are not precisely quantified, including amounts involving arbitrary units and floating currencies. Unit specifies a human-readable form of the unit. System contains the identification of the system that provides the coded form of the unit. The code contains a computer processable form of the unit in some unit representation system. For example, blood pressure can be expressed in the system `http://unitsofmeasure.org` and the unit mm[Hg].
- **Query<T>**:
  Attributes: List<T> values, List<String> qualifier
  The Query object is used to express search restrictions in a string-based format for interacting with the REST-based health API. For example, before a specified date is encoded as 'ltyyyy-MM-dd'. The list of qualifiers specify how the corresponding value should be used

(e.g., lt for less than a date, matches if it should be present in a string, exact for an exact string match).

- **Range**:
  Attributes: long serialVersionUID, **Quantity** low, **Quantity** high
  Base StructureDefinition for Range Type: A set of ordered Quantities defined by a low and high limit.
- **RiskAssessment**:
  Attributes: List<**Identifier**> identifier, **ObservationStatus** status, **CodeableConcept** method, **CodeableConcept** code, **Patient** subject, **Date** occurrence, String condition, **Identifier** performer, List<**CodeableConcept**> reason, List<**Observation**> basis, String mitigation, List< org.hl7.fhir.r5.model.Annotation > note, long serialVersionUID, **RiskAssessmentPredictionComponent** prediction
  An assessment of the likely outcome(s) for a patient or other subject as well as the likelihood of each outcome. Method specifies the algorithm, process or mechanism used to evaluate the risk. Code specifies the type of the risk assessment performed. For example, in the system `http://snomed.info/sct`, '709510001' represents Assessment of risk for disease (procedure). For assessments or prognosis specific to a particular condition, the condition attribute indicates the condition being assessed. It should be a reference to an existing condition but is a simplified as a string here. Reason specifies the reason the risk assessment was performed. Mitigation contains a description of the steps that might be taken to reduce the identified risk(s).
- **RiskAssessmentPredictionComponent**:
  Attributes: String outcome, **Range** probability, String qualitativeRisk, **BigDecimal** relativeRisk, **Date** when, String rationale, long serialVersionUID
  Describes the expected outcome for the subject. Outcome speciifes one of the potential outcomes for the patient (e.g. remission, death, a particular condition). QualitativeRisk indicates how likely the outcome is (in the specified timeframe), expressed as a qualitative value (e.g. low, medium, or high). Rationale contains additional information explaining the basis for the prediction. Both outcome and qualitativeRisk are CodeableConcepts (which can be formal reference to terminology or an ontology), but simplified as a string.
- **RiskEstimationId**:
  A piece of data uniquely identifying a risk estimation performed by the PMS. This architecture does not specify the exact format of this identifier, but possibilities are a long integer, a string, a URL etc.
- **SensorDataElement**:
  Attributes: String sensorId, String sensorType, String measurementType, String measurementUnit, **BigDecimal** measurementValue, String measurementValueString
  Each sub-package lists the id of the sensor, the type of the sensor, the type of the measurements and the measurements itself. These measurements range from a single value (e.g., in case of the blood pressure) to a complex data structure (e.g., in case of an ECG). Non-numerical types of measurements can be captured in the measurementValueString attribute.
- **SensorDataPackage**:
  Attributes: List<**SensorDataElement**> sensorDataElements
  A package of sensor data, i.e., a list of sensorDataElements.
- **string2**:
  Return JSON string user info
- **Timestamp**:
  The representation of a time (i.e., date and time of the day) in the system.

E.7 Unresolved issues

SA Plugin v6.0.8 (VP OpenAPI v16.3)
No failed checks