CALL H2020-SC1-FA-DTS-2018-2020
Trusted digital solutions and Cybersecurity in Health and Care
TOPIC DT-TDS-01-2019
Smart and healthy living at home

SMART BEAR

"Smart Big Data Platform to Offer Evidence-based Personalised Support for Healthy and Independent Living at Home"

D27 – Report on integrated SMART BEAR platform v1

Due date of deliverable: 28/02/2022
Actual submission date: 28/02/2022

Grant agreement number: 857172
Start date of project: 01/09/2019

Lead contractor: CNR
Duration: 60 months

Project funded by the European Commission within the EU Framework Programme for Research and Innovation HORIZON 2020	
Dissemination Level	
PU = Public, fully open, e.g. web	✓
CO = Confidential, restricted under conditions set out in Model Grant Agreement	
CI = Classified, information as referred to in Commission Decision 2001/844/EC.	
Int = Internal Working Document	

D6.2 (D27) – Report on integrated SMART BEAR platform v1

Editors
Manuel M Perez Perez - ATOS
Alberto Acebes - ATOS

Contributors
Adalberto Sangalli - ARIA
Gianluca Carletti - ARIA

Reviewers
Stéphane Railhet - SV
Ioannis Kouris - ICCS

The work described in this document has been conducted within the project SMART BEAR, started in September 2019. This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 857172

©Copyright by the SMART BEAR Consortium.

Executive Summary

This document describes the activities performed within the scope of the WP6 related to the integration of the SB@Cloud components to implement the functionalities for the first prototype used during the Pilot of Pilots (PoP).

Everything described here has a direct and complementary relationship with the documented content in deliverables D6.1 (I*ntegrated SMART BEAR platform*) as well as previous deliverables already submitted at month 19, D6.8 (*Report on integrated SMART BEAR platform)* and D6.7 *(Integrated SMART BEAR platform).* In this regard, this deliverable is a direct evolution of D6.8. The union of all the information gathered in those deliverable complement D6.2 and provides a complete vision, practical, and demonstrative information about the integration of the components for the prototype. For this reason, and in order not to be repetitive in terms of content, frequent cross references are made, whereas in some cases and for clarity shake, some tables are repeated. In that case, the reference to correspondent deliverable is provided.

As we stated in previous report D6.8, the scope of this integration was adapted from the original definition for the Pilot and Pilot, since at the current stage, the whole infrastructure is still not available and therefore a simplified integration was accomplished, sufficiently for demonstration and pilot execution.

Due to the above, this document takes particular care to not create confusion between the advanced work for the integration of the final platform (CI/CD, Kubernetes related configuration, extended security aspects) and the work done and temporally alternatives applied in this integration for the PoP.

Contents

1 Introduction 6
1.1 Goals of the deliverable 6
1.2 CI/CD integration 6
1.3 Kubernetes 6
2 GitLab CI/CD 7
2.1 GitLab CI/CD concepts 7
2.1.1 Pipelines 7
2.1.2 Variables 8
2.1.3 Environments 8
2.1.4 GitLab Runner 9
3 Kubernetes 10
3.1 Data storage 11
3.2 Cloud oriented and load balance service – Ingress controller(nginx) 11
3.3 HAProxy and NodePort service type 12
3.4 DNS Resolution in the cluster (CoreDNS) 13
3.5 Monitoring the cluster (Prometheus) 13
3.6 Automatic renewal of certificates in the cluster (cert-manager) 13
3.7 Kubernetes descriptor files 13
4 Security 15
5 Assistance and Testing 16
6 Concluding remarks 17

List of acronyms

Acronym	Description
ACME	Automatic Certificate Management Environment
API	Application programming interface
BDA	Big Data Analytics
CA	Certificate Authority
DSS	Decision Support System
GDPR	General Data Protection Regulation
PoP	Pilot of Pilots
REST API	(Also, RESTful API) is API that conforms to the constraints of REST architectural style and allows for interaction with RESTful web services
SB	Smart Bear
SB@App	Smart Bear smartphone app

1 Introduction

As we stated in D6.7 "*The SMART BEAR platform is formed of a complex set of components so that is why the issue of integration is an important aspect to consider. Therefore, the platform integration was discussed from the beginning of the project when it was decided to follow CI/CD methodologies to facilitate this aspect and for the integration to be carried out in the most efficient viable way*."
In view of the necessity of having an operating platform ready for starting the PoP it was decided to integrate the different components and services developed in the scope of the WP4 and WP3 by using an environment based on a Virtual Machine (VM) and docker compose instead of the more advanced CI/CD integration under the framework of Kubernetes.
As the PoP have been executed satisfactorily and important lesson have been learnt during the two months of the pilot: we can conclude that our alternative integration was plenty of advantages, and let us to start the PoP, maintain the platform and upgrade it when it was necessary in a very feasible way.
Certainly, as the project is evolving, new functionalities are included, and the pilots will escalating the original CI/CD based on Kubernetes approach will be used.

1.1 Goals of the deliverable

The goal of this deliverable is to describe the activities performed during the past eleven months regarding the integration and testing work done. This deliverable also provides clarification regarding the current achievements and deviation from the initial integration plan. The second release of the project's components is due by the end of 2022. Therefore, at this stage of the project, most but not all the services are integrated in the production environment. The integration framework is, however, prepared for future integrations using Kubernetes.
Along with the document, there are references related to the work carried out for the integration performed for the PoP and the original plans based on CI/CD integration using Kubernetes, explaining how we are working in parallel with both alternatives.

1.2 CI/CD integration

The main activity within task 6.1 is the continuous integration/continuous deployment of the different components of the SMART BEAR platform. The Gitlab repository (https://scm.atosresearch.eu/ari/smartbear) has been maintenance by Atos. During this last period, the integration has evolved from a Docker compose based integration to a more automatic framework based on Kubernetes.

1.3 Kubernetes

Kubernetes[5] is a portable, extensible, open-source platform for managing containerised workloads and services (containers orchestration), that facilitates both declarative configuration and automation. In a separate section is described in details what Kubernetes consists in, as well as the steps followed for its configuration and its practical application in this project.

2 itLab CI/CD

Among of the objectives of D6.1 activities is to build a development environment based on Continuous Integration and Continuous Delivery (or Deployment), abbreviated as CI/CD, in which the changes in the components are carried out in the most agile and controlled way. Therefore, each component modification is identified, tracked, tested and automatically deployed in production

2.1 GitLab CI/CD concepts

Continuous integration is a software development agile practice where the work done by all the team members are integrated frequently and verified by an automated build and testing process, to detect integration errors as quickly as possible, so there will be notable savings in maintenance and support costs. The most significant advantages of using continuous integration are:

- *Reducing the complexity of long projects*, when software construction tasks are automated and iterations are used, it is easier and faster to maintain the projects, thus improving the quality and ease of maintenance.
- *Reduce errors*, since continuous testing when building provides early alert and identifies potential errors.
- *Facilitate deployment*, because unitary functionalities are continuously tested and validated whereas new ones are intensively tested to ensure they work properly before being integrated
- *Deliver faster*, automatisation allows developers to focus on features and improve costs and productivity.

Although mostly common to any continuous integration environment, the following concepts are framed and specified in the context of the GitLab related functionalities.

2.1.1 Pipelines

The *pipelines* are the base concept of the CI/CD process and consists of *jobs*, which define what to do (building, testing, deploying…) and *stages*, which define when to run the jobs (test after building, deploy after success tests).

The jobs computing needs are met by processes denominated *runners*, which can conceptually execute in parallel the jobs that are related to the same stage as far as the resources allow it. The pipeline will move to the next stage only if all current stage's jobs succeed.

One of the main ideas of the pipelines is that they can be executed automatically when source code modification are pushed although they can be forced on demand.

In the SMART BEAR GitLab repository, the CI/CD functionalities are activated and for most of the SMART BEAR components will consist of three stages:

- a **build** stage, in which a single job builds the component,
- a **test** stage, in which a single job runs unit tests and
- a **deploy** stage, in which the component's docker image will be pushed to a registry repository and deployed to the Kubernetes cluster's test environment applying the descriptor files.

.

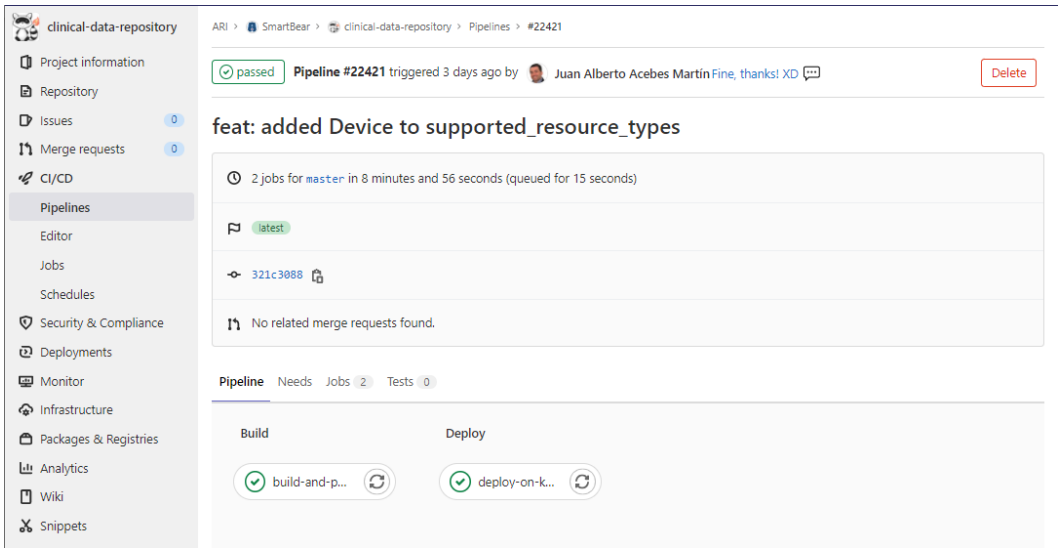

Figure 1: GitLab pipelines

2.1.2 Variables

The CI/CD variables are environmental variables available to be used when running pipelines' jobs. The stored values can be reused and to avoid hard-coding values in the configuration files like in the default CI/CD configuration file named `.gitlab-ci.yml`.

The variables can be defined at:

- CI/CD configuration file, as lowest level describing the pipeline execution,
- Project level, as its name suggests, with project scope,
- Group of Projects level, common to the group of projects,
- GitLab Instance level, that are auxiliary global defined ones.

There is also a set of *default predefined variables*[2], that are general purpose variables, and they can be printed with the "`script`" command.

The SMART BEAR GitLab repository has the following variables set:

- At Project level, depending on the kind of project, PROJECT_DOCKER_IMAGE_NAME and PROJECT_K8S_CONFIG, this last one containing resources with sensitive data to avoid having it in files.
- at Group level: CI_REGISTRY, CI_REGISTRY_PASSWORD, CI_REGISTRY_USER, K8S_SERVER, K8S_TOKEN and PROJECT_K8S_NAMESPACE ready and overwriting default GitLab ones coming from the instance or specific Kubernetes integration functionality.

2.1.3 Environments

In GitLab, the concept of *environment* describes where the code will be deployed. In complex projects such as SMART BEAR is important as different environments are set up, and therefore is needed to create distinctions between different ones

Defining environments allow GitLab to provide a full history of deployments for each of them and track them in temporary terms, so the systems administrator can always be aware of what is deployed on the servers or, in this case, in the SMART BEAR's Kubernetes cluster.

The configuration of the SMART BEAR GitLab repository is focused on the development and testing phases, so for now, only a default test environment is set up, and future promotion to production environments policy is not defined until now. Depending on frequency, risks and real interests, it may be manual or automated.

2.1.4 GitLab Runner

GitLab is, in terms of storage, basically a code repository, but it also provides functionality for storing artifacts[3] and docker images. We use GitLab for docker images repository and registry. If more computational resources are needed for performing the CI/CD tasks (like building and testing the applications), these computational resources can be disassociated from the GitLab server and delegated to another machine with the help of a *runner*. This runner is an application that can be installed on any infrastructure that we own or manage, preferably on a server that is separate from the one that hosts the GitLab instance for security reasons

3 Kubernetes

Kubernetes[5] is a portable, extensible, open-source platform for managing containerised *"workloads and services (containers orchestration), that facilitates both declarative configuration and automation*" (https://kubernetes.io/docs/concepts/overview/what-is-kubernetes) In a separate section is described in detail what Kubernetes consists in, as well as the steps followed for its configuration and its practical application in this project.
As it is stated in the ReHat web page about Kubernetes, *"it is an open-source container orchestration platform that automates many of the manual processes involved in deploying, managing, and scaling containerised applications*" (https://www.redhat.com/en/topics/containers/what-is-kubernetes). It also supports microservices oriented architectures and containerised components, enabling the deployment of them through declarative files, which defines the necessary resources and allow their monitoring and scalability in an effortless way.
To understand the installation activities of the SMART BEAR platform it is necessary to understand some basic concepts, briefly defined below:

- Kubernetes cluster is the set of resources, nodes that manages the container orchestration environment. That is based on the union of several coordinated independent machines on which to balance the workload.
- Kubernetes resources
  - The *Namespace* is a conceptual and functional grouping of elements (Pods, Services etc.) withing the Kubernetes cluster.
  - *A Pod* is a cluster of containers, "*with shared storage/network resources, and a specification for how to run the containers. In terms of Docker concepts, a Pod is similar to a group of Docker containers with shared namespaces and shared filesystem volumes*" (https://kubernetes.io/docs/concepts/workloads/pods/).
  - A *Deploymet* allow to "*provides declarative updates for Pods ReplicaSets" (https://kunermetes.io)*.
  - A *ReplicaSet's* "*purpose is to maintain a stable set of replicas (identical) Pods running at any given time" (https://kunermetes.io)*.
  - A *Service* in Kubernetes "*is an abstract way to expose an application running on a set of Pods as a network service. Kubernetes gives Pods their own IP addresses and a single DNS name for a set of Pods, and can load-balance across them" (https://kubermetes.io)*.
  - A resource is an endpoint in that manage a specific set of API objects
  - A *Ingress Controller*, is a resource that defines a network entry point for a published service in the cluster, providing the balancing and redirection abstraction layer needed by the frameworks ready for scaling.
  - *Master node (aka control-plane) and Worker node* are the different kind of nodes forming a Kubernetes cluster,. The Master node administrates and controls the infrastructure whereas the Worker node performs the processes.
  - *A Service Account* is a resource used to authenticate to the Kubernetes API server or to external services.

  Role/ClusterRole and *RoleBinding/ClusterRoleBinding* (depens if wanted namespace scope or the entire cluster) are resources that alongside ServiceAccounts provides a RBAC Authorisation mechanism to secure access to operations with other resources

The practical implementation of the Kubernertes framework is described in D6.1.

3.1 Data storage

When deploying a platform in a cloud-like environment, one of the services that are taken for granted is the one providing storage as permanent volumes in a simple way. Therefore, components should be able to request storage (depending on their needs) to Kubernetes. The techniques for providing storage in Kubernetes are based in three concepts:

- The *Persistent Volume Claims* (PVC) is a formal request for storage done by a component just defining requirements of specific size and access mode (*ReadWriteOnce*, *ReadOnlyMany* or *ReadWriteMany*) and expecting from the Kubernetes cluster to be provided.
- A *Persistent Volume* (PV) is a piece of storage provisioned in the cluster. It can be provisioned manually by the administrator or dynamically using Storage Classes. Its lifecycle is independent of any particular Pod that uses it
- The *Storage Classes* is a way to define different provisioned storage, so the PVC can be satisfied in a dynamic, efficient way, and also can create different approaches depending on the levels of service quality and needs.

In the case of the SMART BEAR platform, storage through a Network File System (NFS) has been made available in an independent server for storage. It will be the one configured to be used by the automatised assignation and allocation of storage Remark

In order to use it from the Kubernetes cluster, a provisioner will be configured. This includes a *storage class*, which is a process that manages the assignation of space provided by an installation of *Kubernetes NFS Subdir External Provisioner*[4], named as *sb-managed-nfs-storage*, and configured as the default *StorageClass* to manage the requests from the PVCs.

This storage provider class is an abstraction that can be changed at any time for another with other characteristics due, for example, because of another type of environment where storage services are provided by a third party or even to balance between different types of storage that have different capacity and price and allocate based on your need.

3.2 Cloud oriented and load balance service – Ingress controller(nginx)

The design of the architecture and type of integration with containers orchestration tools such Kubernetes, aims to prepare and get as close as possible the platform for a production environment, where features such as maintenance and scalability are critical. The current trend is to develop a platform oriented to services and deploy it in the SMART BEAR cloud environment

The SMART BEAR platform relies on cloud services, as part of the infrastructure. Specific cloud services like *load balancers* are automatically provided and Kubernetes is ready to manages them. In the case of SMART BEAR the cluster is set based on bare metal servers, therefore the requests are arriving directly from the public IP/domain to the IP assigned to the node. They are usually performed without going through other intermediate requests flow management.

The default way of providing a gateway (or "entry point") to the Kubernetes cluster for the external requests and reach the services, is by setting them using a specific type *LoadBalacer*, which expects from the cloud service provider to be automatically assigned with an IP address from a configured pool, and then let it set for the proper traffic forwarding.

Our infrastructure lacks this load balancer cloud service so we must use an alternative strategy and delegate this first load balancing and port forwarding feature to a process in our own infrastructure, before reaching the Kubernetes cluster. We also need to mention at this point the concept of *ingress*, which is a Kubernetes API object that manages external access to the services within the cluster and may provide load balancing, SSL termination and name-based virtual hosting.

Furthermore, most standard out of the box load balancers can only be deployed to a public cloud provider and are not supported for on-premises installations, but a built-in ingress controller may be provided.
In our bare metal on-premises installation of Kubernetes our strategy details are still pending to be defined, but mainly we will expose the needed services (like ingress-controller or directly the security component in charge of routing, like the gatekeeper service developed by STS as *NodePort* Kubernetes service type, and use an external reverse proxy to redirect requests to such service.
Alternatively, we can simulate (internally) a cloud load balancer service provider so we still can configure Kubernetes services as *LoadBalancer* type and keep a common cloud configuration.
One of the most common strategies for publishing the services provided by a Kubernetes cluster is through the use of the *ingress* type of resources and using a service and process that manages them, the *ingress-controller*.
Its use is a very flexible and convenient way to redirect subdomains, IPs and manage the incoming requests, and its installation has been done in the SMART BEAR Kubernetes cluster. The configuration was made for testing purposes, knowing that its use will probably be withdrawn in favor of a direct forwarding from the reverse proxy installed in the machine, the HAProxy. The main advantage of HAProxy is that all the needed ports are managed by a single security component in charge of its saucerization and routed to the final service and component.

3.3 HAProxy and NodePort service type

In order for the requests to reach the ingress-controller service, given the limitations previously exposed with the *LoadBalancer* type of service, another type is used in the configuration, the *NodePort*[6] Kubernetes service type.
When configuring a service as *NodePort* type, Kubernetes exposes it on each node's IP at a static random assigned port (by default between 30000 and 32767) and a *ClusterIP* Service, to which the *NodePort* service routes, is automatically created. This allows the service to be accessed from outside the cluster and, in the case of using it to redirect incoming requests to the *ingress-controller* service, get such requests to reach the final services. This approach is called by NGINX Ingress Controller "using a self-provisioned edge" [7].
The missing piece is the one that acts as a proxy, balancing and distributing the requests from outside the cluster among the cluster nodes, master or workers, depending on the kind of request. We have used *HAProxy*[8] for this purpose.

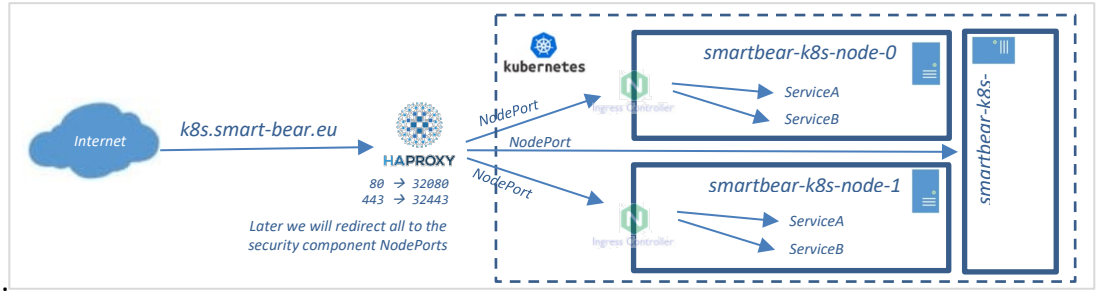

Figure 2: HA Proxy as a front reverse proxy

3.4 DNS Resolution in the cluster (CoreDNS)

In the Kubernetes cluster it is important to understand how components can interact with each other, and how *Services* and *Pods* can be reached[9].
By default, the DNS-Based service discovery installed in Kubernetes is *kube-dns* It enables DNS service within the cluster by resolving services using DNS records of type A/AAAA, following the format `<service>.<ns>.svc.<zone>`:

- <service>, service name declared in the *Service* resource,
- <ns>, the namespace in which the service is associated,
- <zone>, is the cluster domain,

like, for example, *clinical-data-repository-database.smartbear.svc.cluster.local*
With this format, the DNS should resolve the specific IP of the Service and for services within the same namespace, the last part is not needed, and only with the service name should the target be reached.

3.5 Monitoring the cluster (Prometheus)

In order to monitoring the health status of the Kubernetes cluster, an open-source tool has been installed which provides out-of-the-box monitoring capabilities and a metrics endpoint that allow other tools (like Lens GUI) to visualise such data in a convenient way and generate alerts for the system administrator. .

3.6 Automatic renewal of certificates in the cluster (cert-manager)

As in the previous version of the platform, we use Let's Encrypt certificates to secure with SSL and because these certificates have a short life, the automation and management of these must be delegated to an automatic process that does not require the continuous intervention of a system administrator.
Also, mentioned in section 2.5 HAProxy and NodePort service type, in the first phases of the cluster installation, has been used the functionality of the reverse proxy (HAProxy) for the configuration of HTTPS based on certificates, but is still not decided if eventually will be delegated to an internal component, and he will oversee this.
In case that eventually the management of the certificates is performed in the scope of the Kubernetes cluster, a tool called cert-manager10 is used. This tool simplifies the process of obtaining, renewing, and using certificates, and is compatible and can issue certificates from several of the most used sources, including Let's Encrypt, that is the one we use, and will ensure that the certificates are valid and up to date, and will attempt to renew certificates at a configured time before expiry.

3.7 Kubernetes descriptor files

For every single component, several Kubernetes descriptor files should be prepared, in a homogenised way that allows its orderly integration and future maintenance.
Since we had the configuration of the previous version with *docker-compose*, for the migration the procedure is basically to transfer it to Kubernetes descriptors one by one, but now bearing in mind that CI/CD will be used, so the creation of resources should be done in a more coordinated way for integration.

At the time of writing this document, we are following the next phases for each component/owner, providing support, but with the need of active cooperation from the component owners:

1. Identification, grouping and naming of containers into Kubernetes deployments files. This will be similar to the process we did with the docker-compose files approach.
2. Extraction of any configuration, parameters, and passwords into Kubernetes resources of type ConfigMap and Secrets. This is mandatory, for safety and maintenance, even if we have the images in a private repository.
3. Definition of the services and ports exposed. We will maintain the current approach with the gatekeeper in charge of any out-ingoing connections, so the only thing that changes is that a naming convention will be set up for the services to facilitate configuration and maintenance.
4. Definition of disk requirements to define *PersistenceVolumeClaims*. And to identify backup requirements.
5. Definition of CPU/Memory requirements if affinity adjustment is needed
6. Dependencies between components(services) and way to check healthy/readiness (for example, if it depends on a DB to be ready to boot properly)

4 Security

Once the SMART BEAR platform becomes a real production tool used by pilot sites, with real data involved, the highest requirements of security must be assured and guaranteed.
Regarding infrastructure, the possible access routes are limited to the maximum and the essential ones will be secured in the manner detailed in the following sections.
One of the strengths in favour of the infrastructure and platform architecture design is the fact that a specific security component centralises all traffic to the system. Only the Kubernetes control API remains in parallel, but to access it will be necessary to do so through a Virtual Private Network (VPN)
As described in previous sections, in the instances where the Kubernetes installation has been executed precaution has been taken in order to prevent basic uncontrolled access configuring firewall, VPN and certificates for connections.

5 Assistance and Testing

In the transition from the preliminary phase of the PoP until the beginning of the pilots, it was necessary to define a structured approach and have tools to manage the following needs as volumes and complexity grow:

- Tracking and processing *feedback* coming from clinical and technical SMART BEAR teams, leading to platform and process improvements
- Tracking technical *support requests* either raised directly from clinical and technical SMB teams, or forwarded from local pilot help desks.
- Incident management, bug identification and tracking
- Testing both single components and the integrated platform, with a structured approach and a tracking system
- Automatic and non regression testing tools and procedures

In order to respond to the needs a, **SMB Central assistance and Testing processes** have been designed

In order to efficiently manage the feedback and issues reported by the clinicians, the SMART BEAR Feedback Corner was developed. The SMART BEAR Feedback Corner consists of:

- an assistance portal, built on Hipporello framework, and dedicated to clinicians, operators or technicians involved in the SMART BEAR; so far fourteen issues have been reported during the PoP;
- a Trello board is connected to the issues reported by Hipporello. This board allows the technical team to better address user requests. The tool is based on tracking cards and classification lists. All the issues reported by Hipporello during the PoP were satisfactorily

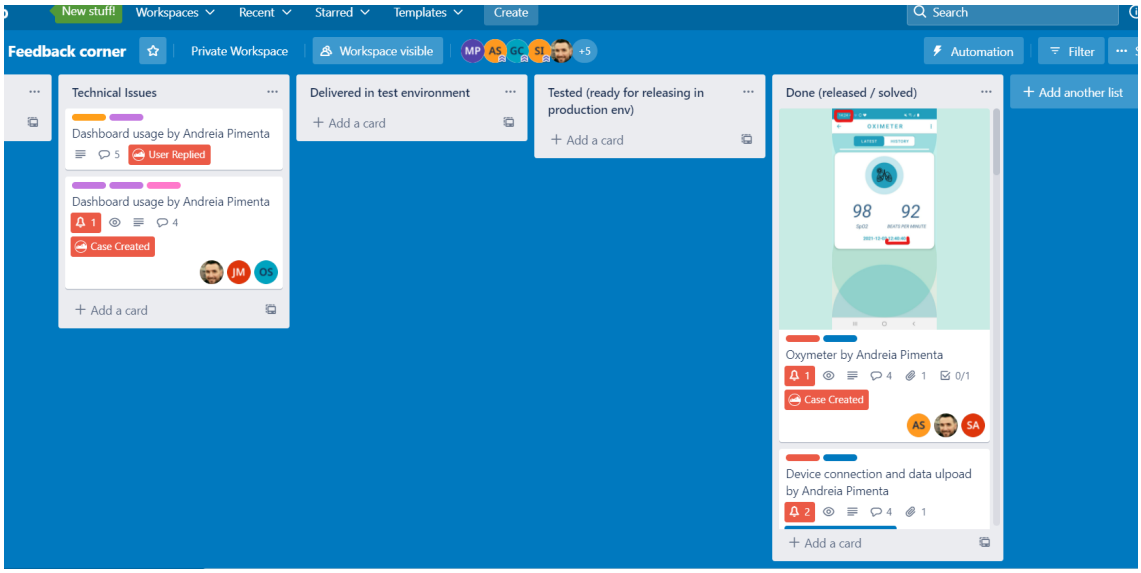

Figure 3: Screenshot taken from the Trello board showing the current testing and maintenance activities

6 Concluding remarks

The activities within the WP6 have been performed according to the plans, with no delays. Issues, problems, and advances are discussed every Friday in a specific WP6 conference call. Still, communications using email, skype and Basecamp are ensuring mutual understanding among technical partners. The SMART BEAR corner is serving not only as an effective way for communicating and exchanging information between clinicians and technical teams, but also for monitoring issues and improving maintenance and collaborative work. The integration based on Kubernetes is moving forward, and the framework will be fully operative for CI/CD integration and ready for the pilots.