Co-funded by
the European Union

Smart Survey Implementation

Grant Agreement Number: 101119594 (2023-NL-SSI)

Work package 3
Developing Smart Data Microservices

Deliverable 3.1: Review stage report

Version 1.0, 2023-10-19

Prepared by:

Joeri Minnen (hbits, Belgium)
Tom Oerlemans (CBS, The Netherlands)

Work package Leader:

Joeri Minnen (hbits, Belgium)
e-mail address : Joeri.Minnen@hbits.io
mobile phone : +32 (0)497 189503

Index

1. General introduction

This document provides a first overview of the work done in work package 3 of the SSI project.

The main goal of this work package is to develop microservices and to arrive to the overall goal 'to develop, implement and demonstrate the concept of Trusted Smart Surveys, realizing a proof of concept for the complete, end-to-end data collection process and demonstration of a solution'. Work package 3 is situated at the Development level, where the microservices are being developed as platform-independent components.

Three different microservices have been indicated to be developed:

- Receipt scanning microservice
- Geolocation microservice
- Energy microservice

The main objectives of WP3 are:

- Develop the selected microservices
- Develop the APIs between the microservices and the core platforms
- Setup and perform development tests to support the development of the microservices and APIs, in an interactive and iterative manner
- Document the microservices and APIs
- Support platforms and NSIs to include the microservices in the core platforms
- Perform a pentest
- Perform a stress test (load performance of eg. geolocation data)
- Containerise the microservices
- Describe the architecture of the core platforms
- Describe the architecture of the (developed) microservices
- Describe and execute the deployment strategy for both the core and microservices
- Write PDCA-cycles
- Keep and maintain a public GitHub repository
- Coordinate with the participating countries
- Provide support to WP1, WP2, WP4 and WP5

Looking to the timeline, the first microservice to be developed is the receipt scanning microservice, followed by the geolocation and the energy microservice.

The work on the receipt scanning microservice started in May 2023 and a first demonstration of the work done on extracting information from a ticket, and on recognizing different parts of the ticket has been demonstrated during the SSI informational meeting taking place on the 20th of October 2023. The work on the geolocation microservice is started in October 2023. The overall approach is to first develop various components of the microservice. In a second stage the components are tied together in a process flow. At that time the microservice can be tested and evaluated. During the entire duration of the project improvements to the components can be taken into account.

While most of the objectives have been started (and the main objective is the development of the services), this review stage report has a first focus on the Microservice software architecture. This document is essential for developing the overall microservice architecture, but also has a relation to the thematic development of the different microservices. A second focus of this report relates to the Receipt scanning microservice, and more in particular to the business, functional, and non-functional requirements of that service.

Both the chapters have been discussed during online meetings and during physical workshops organised by CBS (Heerlen), Destatis (Bonn) and hbits (Brussels). Accordingly, the chapters are reviewed multiple times by the work package leaders. After an in-depth review also the countries and the experts related to WP3 have been provided the opportunity to reflect upon the documents.

2. Microservice software architecture

This chapter describes a generic architecture for data processing microservices.

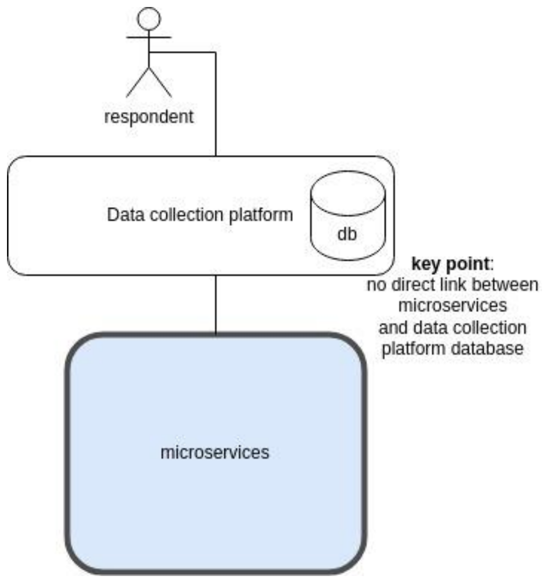
The structure of this chapter is based on views and perspectives. Views illustrate the structural aspects of an architecture (e.g. where is data stored?), while perspectives consider the quality properties (e.g. scalability) of the architecture across a number of views.

The microservice is independent and not coupled to a specific data collection platform.

Views

Context view

Next diagram shows how microservices (as a black box) fit into a general data collection platform architecture.

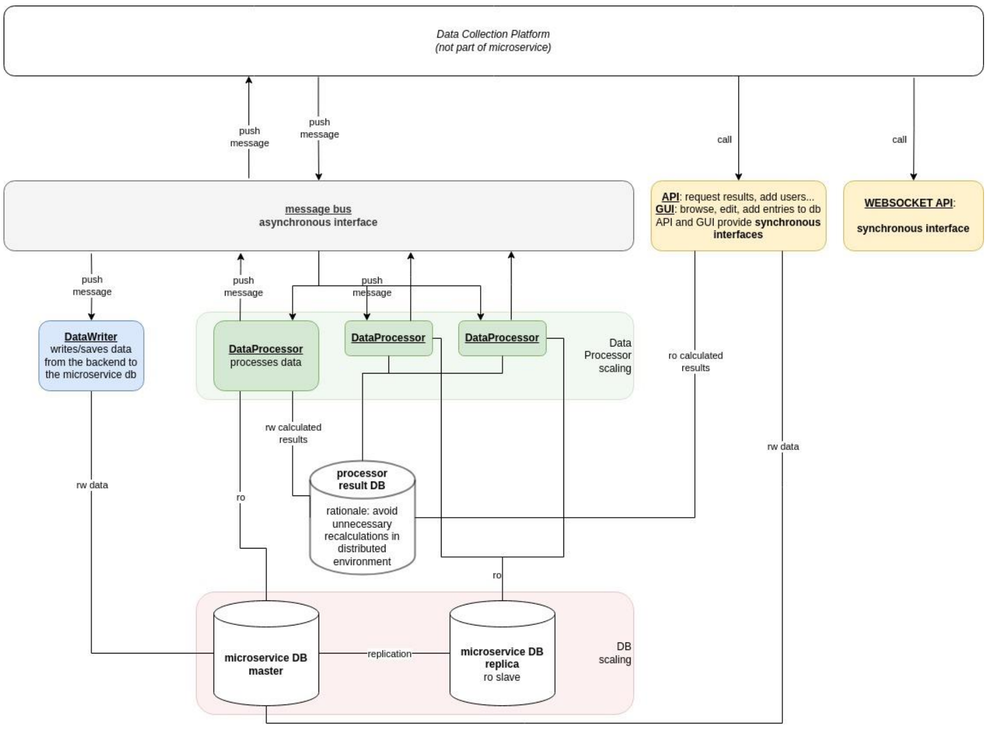

respondent
Data collection platform
db
key point:
no direct link between microservices and data collection platform database
microservices

It is important to mention that:

- there is no direct link between the respondent and the microservices: the data collection platform has full control over microservice usage (who/when).
- there is no direct link between the microservices and the main database. This means that the data collection platform has full control over which data is delivered to the microservices. The exact mechanism which guarantees privacy will be explained in the "Regulation perspective" section.

Functional view

This diagram describes the different runtime functional elements of the microservice.



Main elements and their responsibilities are:

Message bus

The message bus allows for asynchronous communication between the data collection platform and the microservice.

Rationale:

- avoid blocking calls e.g. the platform must be able to quickly forward data (scanned receipt information, geotracking point) to the microservice without being blocked for too long. The message bus is able to fulfill this requirement by putting the data in a queue without any processing. In addition, by putting the message bus on the same server, networking issues between platform and bus are being avoided.
- notification service e.g. the DataProcessor can send a message that (some) data is processed. The platform can then take appropriate action.

Chosen technology: RabbitMQ (https://www.rabbitmq.com/)

API and GUI

Is the synchronous interface of the microservice.

The API is used by the platform to fetch microservice data, request processed data results etc.

The GUI is used by a researcher or operator to:

- browse and inspect the results of the DataProcessor in the processor result DB
- browse, inspect and edit the data of the microservice DB
- possibly other functions e.g. add a scanned receipt and test the outcome

Because microservices have different functionalities, the (optional) GUIs are microservice-dependent. The GUIs are typically built with web technology and preferably share the same web framework than the API part.

Preferably, the GUI is integrated in the platform UI/backoffice in order to get an integrated user experience. This also avoids possible data inconsistencies between microservice database and platform database (e.g. a researcher edits the microservice database but this change is not propagated to the platform database).

It is possible to extend the API with a synchronous call to the DataProcessor's internal algorithm i.e. without doing a request to the DataProcessor container. This is a simpler but more limiting design:

- the number of parallel requests might be limited by the webserver,
- it is a synchronous interface which means the call might block for a while,
- in case of networking issues, there is no queuing of requests or messages (in contrast to the message bus).

The synchronous call might be more practical than the asynchronous one for debugging the algorithm e.g. because no message bus is needed.

Websocket API

The websocket provides a synchronous interface as well. It can be used though to call the DataProcessor's internal algorithm synchronously. See discussing above.

Added value: if deployed, independent product which can be directly used via internet.

Nice to have test platform.

Library vs service. Proposal: take into account in architecture/design (but no development yet).

DataWriter

Receives push messages with data from the platform via the message bus. The DataWriter writes the data in the microservice DB. Data push messages are queued in the message bus until the DataWrites is able to accept them.

There is only one DataWriter process in order to guarantee that the received data is written to the database in the same order as the data was pushed by the platform. This avoids (subtle) race conditions in which a DataProcessor start processing data with missing in-between data (e.g. a tracking point is missing in the db between the first and the last tracking point).

Because the DataWriter only writes data to the database and doesn't process data (no cpu time), it is expected that it will be fast enough to always empty the queue. If this is not the case, platform design (and *not* microservice design) must be reconsidered e.g. by limiting the number of message sent to the message bus.

DataProcessor
Does the real work of processing the data.

Starts processing when it receives a push message of what to process. Note that a single push message is delivered to one and only one DataProcessor. Scalability is achieved by load balancing the requests over the DataProcessors , which is a feature of the message bus. See the perspective on performance and scalability.

Given the required processing time needed by the processors, the platform should limit the number of messages sent to the data processors (via the message bus). In this regard, system design is important. E.g. in case of geo tracking, the processors shouldn't be triggered for each tracking point to recalculate the respondent's itinerary, rather, the tracking points should be bundled before the recalculation is done.

Because data processors act independently, the concurrency aspect of the data processor must be taken into account in its design. At an infrastructure level, limiting resources (e.g. in a k8s cluster) might be necessary.

A DataProcessor pushes a message on the bus when processing is done.

Processor Result DB
This DB stores the results of the DataProcessors. Since results can be re-calculated, persistent storage is not a strict requirement e.g. one might opt to make it a RAM only db.

A non-sql database is probably most convenient to store the results.

It is accessed by the API/GUI element to fetch the results.

It can be consulted by DataProcessors to avoid the re-calculating of data, it therefore also act as a cache.

Chosen technology: redis (https://redis.io/). It is also possible the replicate a redis db over several nodes if needed so (see perspective on performance and scalability).

Microservice DB
Used by the DataWriter to store pushed data.

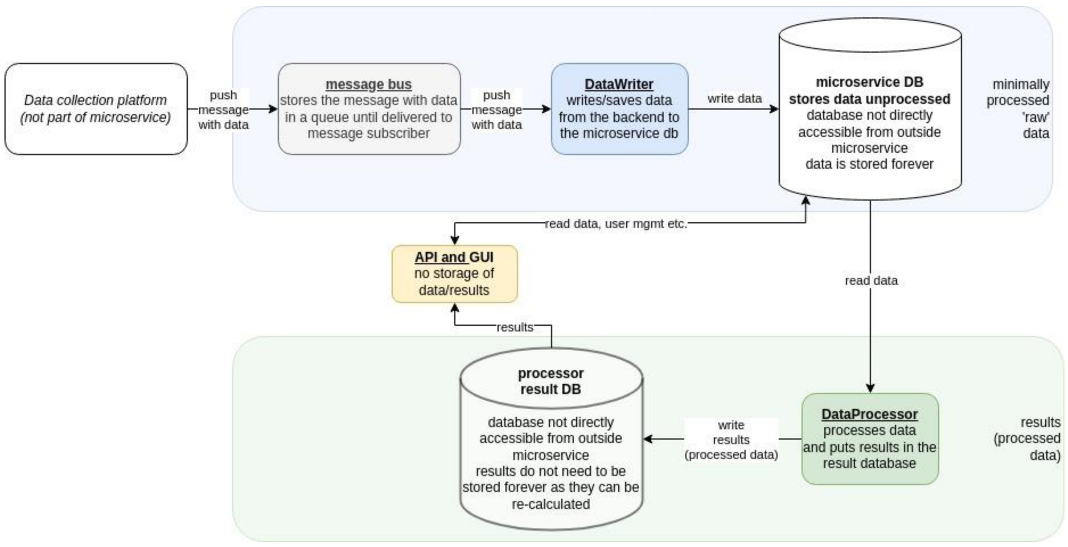
Data is consulted by the DataProcessors.

GUI is able to change data if needed so.

DB scaling is achieved by replication, see perspective on performance and scalability perspective.

Information view

This view describes the way that the microservice stores, manipulates, manages and distributes information. The diagram highlights the key points.

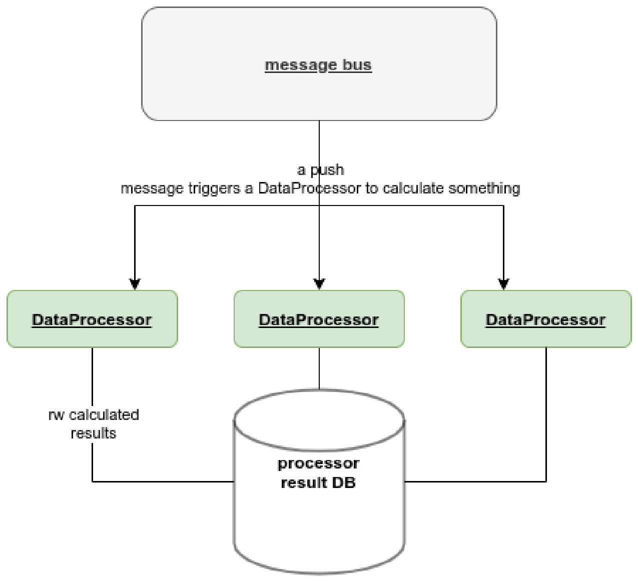

Data collection platform (not part of microservice)
push message with data
message bus
stores the message with data in a queue until delivered to message subscriber
push message with data
DataWriter
writes/saves data from the backend to the microservice db
write data
microservice DB
stores data unprocessed
database not directly accessible from outside microservice
data is stored forever
minimally processed 'raw' data
read data, user mgmt etc.
API and GUI
no storage of data/results
read data
results
processor result DB
database not directly accessible from outside microservice
results do not need to be stored forever as they can be re-calculated
write results (processed data)
DataProcessor
processes data and puts results in the result database
results (processed data)

Concurrency view

This view identifies the parts of the microservice that can execute concurrently and how this is coordinated and controlled.

All functional runtime elements are allowed to run in parallel since their responsibilities are clear and non-conflicting (e.g. the DataWriter writes data while the DataProcessor processes data).

The most important concurrency aspect in the microservice architecture are the different DataProcessors which can process data in parallel:



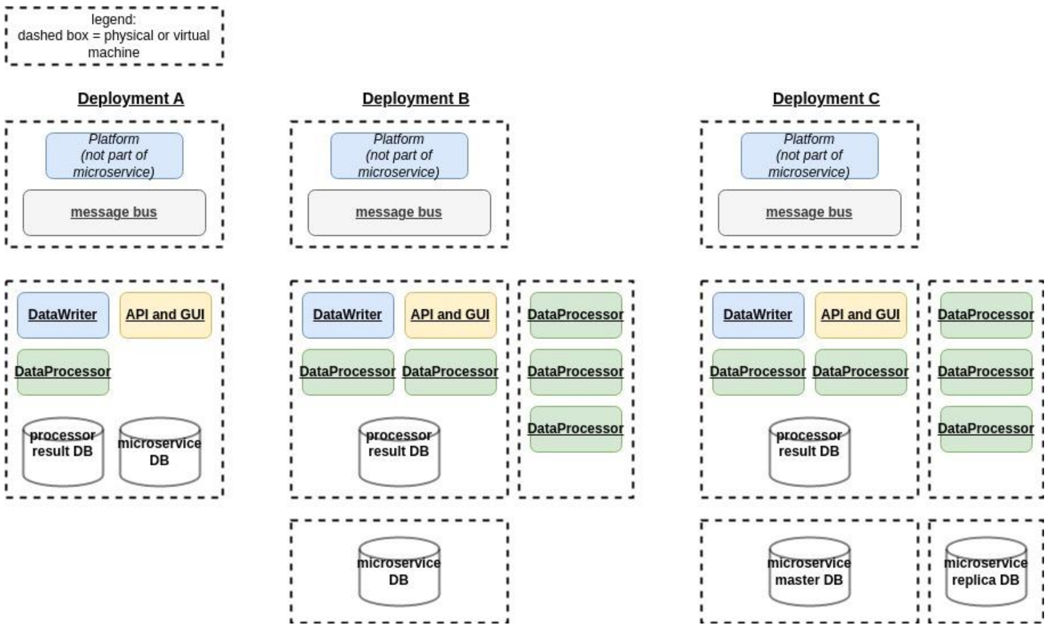
Care must be taken to:

- avoid race conditions: if 2 DataProcessor calculate the same thing, then one DataProcessor might overwrite the results of the other, also if the other's results were more recent.
- avoid unnecessary recalculations: DataProcessors should check the database to make sure the results for its calculation are not already there. In that sense, the processor result DB also acts are a kind of cache. If a single result is a combination of multiple small results, then cache optimizations might be possible by re-using the finer-grained results. E.g. suppose you need to calculate a timelog of a day. If a day is in progress then possibly only recalculating the last hours is enough instead of recalculating the whole day.

Deployment view

A microservice is deployed as a collection of Docker containers: each functional runtime element is built as a Docker container. This makes all elements (almost) independent from the host OS.

Depending on the performance and scalability requirements (see perspective), different deployment strategies are possible. Here are some examples:



Note that the message bus should always be ready to accept messages from the platform (to avoid the platform to be blocked). To exclude networking issues, platform and message bus are on the same machine.

Operational view

Installation and upgrade

The microservice is a collection of Docker containers that will be managed, scaled and deployed with a container-runtime platform (e.g. Kubernetes https://kubernetes.io/ for production environments, docker-compose for development etc.).

Backup and recovery

Two databases are (potentially) part of a microservice:

- processor result database: because results can be recalculated, no backup is needed here except to speed up the recovery process. If Redis is used as technology, then persistency is build-in. Backup/restore is the responsibility of the platform owner (and not of the microservice).

- microservice database: backup/restore is the responsibility of the platform owner.

Note that the choice can be made to store data sent to or received from the microservice *also* in the platform core database. When the microservice is disabled or not needed anymore, then the platform core can function without the microservice (e.g. the respondent's geo itinerary can be retrieved without the microservice).

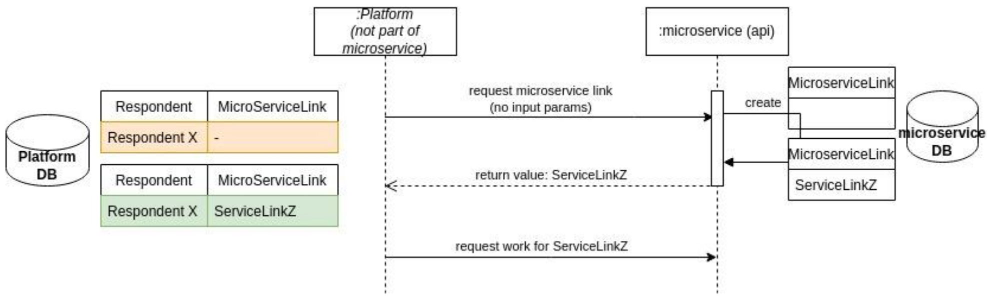
Perspectives

Regulation perspective

Privacy

Sensitive information must be restricted to the database of the main application. The microservice is not allowed to pull user/respondent private information into its own databases.

The following mechanism is foreseen:



The Respondent X entry is never visible in the microservice. Platform and microservice are linked to each other via a "microservice link". The microservice only has knowledge of the abstract "microservice link", which essentially is only an id (uuid, guid...).

If the microservice database would be shared for researchers (e.g. for postprocessing), then the sensitive information of the respondent cannot be leaked.

Performance and scalability perspective

Depending on the application (e.g. type of research) and the type of microservice (e.g. processing intensive vs IO intensive), performance and scaling of the microservice can be tuned as follows:

- DataProcessors can be scaled:
  - by creating multiple instances
  - by distributing instances over multiple machines
  - see 'concurrency view' of how they distribute work

- microservice DB can be scaled:
    - by db replication
    - by distributing the replication databases over multiple machines
- microservice results DB could be scaled similar to the main microservice DB. Since this is probably not a heavily loaded database (no complicated queries, only results), scaling might not be needed

3. Receipt scanning microservice requirements

The goal of the SSI project is to involve and engage households and citizens, and to define and operationalize a new/modified end-to-end data collection process. The project will take a view on all the steps of the data collection process, from the invitation to participate until the completion of the study and the processing of the data. The data collection process includes the use of smart devices and smart data, while privacy is safeguarded.

Centrally stands the use of smart devices and other connected devices to obtain the data. NSIs and linked organizations have worked on platforms to allow households to register their purchases online. These platforms are @HBS of the CBS, MOTUS of hbits and the developments of SSB and Insee.

The SSI project holds 5 different WPs. Whereas WP2 of the SSI has a focus on the interaction of households and citizens with the different platforms and the underlying applications (HCI, Human Computer Interaction) WP3 is within SSI the gateway to include Smart data. The inclusion of Smart data is seen as a need to further support the participation of the respondent in studies like TUS and HBS. WP5 overlooks both WPs from the viewpoint of privacy and security.

An important criterion is the realization of an end-to-end data collection process, that results in qualitative and comparable data. The definition of quality and comparability stems from the mission of the ESS and trust upon the Principles of the European Statistics Code of Practice, which latest update also takes into account the emerging of new data sources and use of new technologies.

In WP3 the Smart inclusion is realized by the development of microservices. This document has a focus on HBS and the inclusion of a microservice for receipt scanning. Through Optical Character Recognition (OCR) HBS relevant information is gained from the ticket and is available in a digitalized format. The microservice is seen as middle part software that is supportive to the household in reducing their burden to complete a consumption diary.

Business requirements

HBS collects in a large detail what households spend on goods and services. In this way, the survey gives a picture of the living conditions in the EU. HBS is performed by each Member State to calculate weighted macroeconomic indicators used for national accounts and consumer price indices. Eurostat publishes output since 1988 and this with intervals of 5 years. The last waves are from 2010 and 2015. In 2026 HBS will enter the IESS agreement.

In a HBS study, (a member of) a household records tickets in a diary. Besides information on the store itself (name, address, logo, registration number, …) a ticket at minimum holds information on the different purchases (or product rows) that are bought and the total price of the ticket. Depending on the shop a ticket can also contain various different contexts to the purchase, and can also hold information on reductions, return items or even empty good claims. The design of the diary defines the amount of detail that needs to be transferred to the diary.

This altogether creates a demanding effort from the participants to the study. Given the drop-in participation rates and supported by the request of the Wiesbaden Memorandum in 2011 Eurostat and the NSIs started to develop and implement new data collection modes to call a hold to this downward trend, and to even improve upon the quality of the collected data.

Initiatives of various countries, and previous EU-funded projects have translated the paper-and-pencil method to an online data collection process, giving households the opportunity to digitalize their ticket by adding purchase by purchase in a step-by-step manner in order to submit the entire ticket.

Notwithstanding the added value of these online applications the burden on the participants remains high, and still too much error prone. The goal of WP3 is to reduce these gaps by developing and implementing microservices that acquire, process and (can) combine data collected from smart devices and other applications, in the case of HBS through the development of a receipt scanning microservice.

A successful realization of the development and implementation will not entirely reduce the active participation of households in the registration of their tickets and purchases, but will provide support and guidance in their task to arrive to qualitative and comparable data for the ESS. It means that besides the development of the microservice also the implementation of the service to the platforms is important, as well as the UI/UX that presents the output of the microservice to the user, and the easiness in which the user can verify, adapt, or even delete the output.

The following objectives are essential in reaching this goal:

- Objective 1: To define an architecture of a microservice (that is also to be reused in the other developments of WP3, being the geolocation and energy use microservices)
- Objective 2: To develop a receipt scanning microservice using OCR
- Objective 3: To implement classification solutions (machine learning, string matching, or search algorithm based) to classify purchases to a COICOP-list
- Objective 4: To develop an API to connect to/from other environments
- Objective 5: To deploy the microservice as a containerized application in the cloud
- Objective 6: To implement/integrate specific microservice parts in the app (e.g. algorithm). This integration should be feasible, should have an added value for the platform and/or should improve the user experience.

The stakeholders are the NSIs and their product owners, and the households (citizens).

Functional requirements

HBS study

In this section HBS studies are being described as they provide the context in which the receipt scanner microservice operates.

In HBS studies questionnaires and a consumption diary are completed by the households. At the moment household members arrive to the diary phase they, at the least, already have completed a questionnaire. If this member is the reference person, or the head of the household also a household questionnaire and a matrix to compose the household is part of the pre-diary tasks. All tasks are defined in a respondent journey or study flow that shows a sequence of tasks. Since the HBS diary setup requires an equal distribution of participation over the entire fieldwork period, and household members are requested to keep their diaries for the same period this study flow can be quite complex.

Central to a HBS study is the registration of tickets and purchases of goods and services in a diary. Households keep one diary over a period of (minimum) 15 days. Left aside paper-and-pencil diaries, a household member partakes to a HBS study via an application, be it via a mobile application, be it via a web application running in a browser.

HBS diary

The diary collects at the minimum:

- a description of the products and services that are bought
- a description of the fixed (repeated) costs that are paid
- the price of each product or service, and
- the date of the purchases and periodicity of fixed costs

The registration of the products and services is linked to a COICOP-classification. COICOP stands for Classification Of Individual Consumption by Purpose, and is demanded to be delivered on 5 digits as part of HBS 2026. NSIs often use more digits to aggregate to a higher level.

The matching COICOP code is selected/mapped from a list:

- a COICOP-code

In addition, on the level of the ticket extra information is/can be gathered:

- the country of purchase
- the shop (brand/type)
- ticket reduction
- professional purchase
- payment method

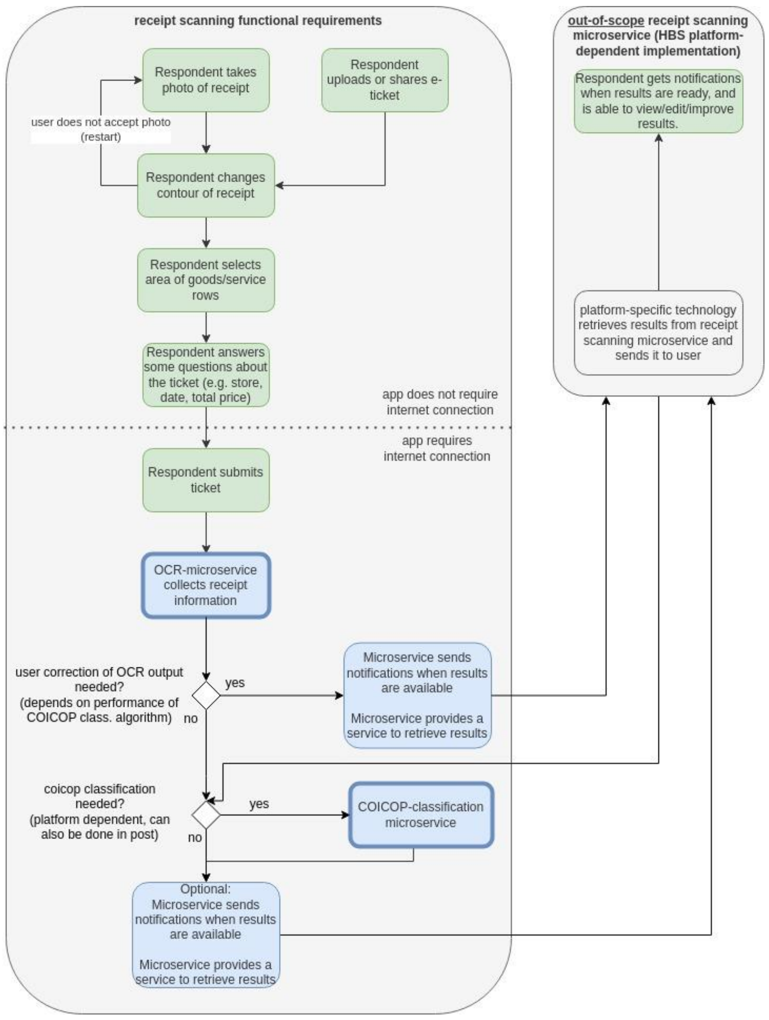
As an extra, on the level of the product or service extra information is/can be gathered (depending on national needs):

- number of items
- price per item
- quantity and metric/unit per item
- discount
- return

Functional requirements

The diagram gives an overview of the main functional requirements:

- functionality related to *user handling* is indicated by the green boxes. The respondent must be able to submit a photo and indicate to the receipt scanner software the receipt location in the photo and provide some receipt details for verification.
- functionality related to the *microservice* is indicated by the blue boxes. The essential function is to find and provide information that the HBS diary collects in a receipt (i.e. all bullet items in section ‘HBS Diary’).


receipt scanning functional requirements
Respondent takes photo of receipt
Respondent uploads or shares e-ticket
user does not accept photo (restart)
Respondent changes contour of receipt
Respondent selects area of goods/service rows
Respondent answers some questions about the ticket (e.g. store, date, total price)
app does not require internet connection
app requires internet connection
Respondent submits ticket
OCR-microservice collects receipt information
user correction of OCR output needed? (depends on performance of COICOP class. algorithm)
yes
no
Microservice sends notifications when results are available
Microservice provides a service to retrieve results
coicop classification needed? (platform dependent, can also be done in post)
yes
no
COICOP-classification microservice
Optional: Microservice sends notifications when results are available
Microservice provides a service to retrieve results
out-of-scope receipt scanning microservice (HBS platform-dependent implementation)
Respondent gets notifications when results are ready, and is able to view/edit/improve results.
platform-specific technology retrieves results from receipt scanning microservice and sends it to user

User handling

Respondent handling (green boxes)	
REQ R1a	Respondent takes photo a receipt
	Select 'take a picture' to open the picture functionality
	Real-time camera opens: • detection of contrast ticket vs background • detection of light (good exposure) • detection of contour of ticket starts (4 dots or polygon around receipt when stabilized)
	App (or user) takes picture when stabilized (contour good enough)

	If the respondent is not satisfied with the photo, he/she needs to able to restart with the photo taking process.
	Note that additional quality checks might (and probably will) be performed later by the app software itself and/or the server-part microservice.
REQ R1b	Respondent uploads an e-ticket with the application (alternative to taking a photo of a receipt)
	Select ‘Submit e-ticket’ to select/download a file from the local filesystem
	It is unclear yet which types of e-tickets will be supported. This strongly depends on the layout and structure of the ticket itself. Possible e-ticket formats to be supported: image, pdf+text(+variants) or pdf+image
REQ R1c	Respondent shares an e-ticket from another application (e.g. store app) to the HBS application (alternative to taking a photo of a receipt)
	Not part of microservice. Platform-specific (app) implementation.
REQ R1d	In case of a web app: scanning can be done by the browser of the smartphone and sent over to the browser running on a computer or laptop
	Respondent is on the web app Respondent wants to take a photo *with the smartphone* so that it is automatically loaded in the *web* app This requirement is a nice to have.
REQ R2	Respondent changes contour of receipt
	Respondent can change the contour (4 dots connected with lines) to define the ticket by moving dots or the line segment (handles) between two dots (parallel movement of two dots).
	Could be skipped if the automatic contour detection works very well. Ideally, this step is not necessary (same as R3).
	Having the complete receipt is important because it contains more info than only the product/service rows. Extra info on the receipt includes: store logo, store details, payment info etc.

REQ R3	Respondent selects other details of the receipt
	Selection of product/service rows. Helps the OCR process.
	Could be skipped if automatic product/service row detection works very well.
	Although extra work for the respondent, this step ensures the software knows the most relevant part of the ticket i.e. the product/service rows. Also, the positional data could be used later for ML training. Ideally, this step is not necessary (same as R2).
	This step does not involve a crop of the image, so at submission, the whole image will be sent to the microservice.
REQ R4	Respondent answers some questions about the ticket (questionnaire)
	The following questions will be asked: • Country • Shop • Language • Date • Total price
	Depending on the specific-platform UI, it must be possible to skip this step. Note however that the output of this step is very interesting for internal quality checks in the OCR process. Furthermore, knowing the store might/will be important for the COICOP classification.
REQ R5	Respondent submits ticket image
	A button allows the respondent to submit the ticket image, the change the contour hint and other selected areas (such as expense items).
	Different UI implementations are possible: • the image is uploaded in the background and the respondent gets a notification when it is done, or • a dialog should run to show the continuation of the upload. • Or, user settings whether he/she wants to send real-time or in background; or, via mobile or only wifi

	Communication of success, or failure which has to be accepted by the user by pressing OK. In case of failure, the action that needs to be undertaken is UI/platform-dependent. E.g. one could wait for a wifi connection before trying to upload the image again. The decision what needs to be done is platform-specific.

Microservice

Microservice (blue boxes)	
REQ M1	Microservice collects receipt information
	The service is best effort and should try to collect the following receipt information: • date of the purchases • a description and price of the products and services that are bought • the country of purchase • the shop (brand/type) • ticket reductions • payment method Then, at the level of a product or service: • number of items • price per item • amount and metric/unit per item e.g. 1,5 L • discount • return or not
	Information retrieved from the user in step (e.g. R2, R3 and R4) could be used as a verification step. E.g. the total price as answered by the respondent should match the total price as derived from the image. If not, the user’s input has priority (esp. in the UI as we don’t want to overrule user’s input).
	Depending on the performance and quality of M1, the number of respondent actions in the UI (more specifically, R2, R3 and R4) might change (e.g. if the service is almost always able to retrieve the product/service rows then step R3 is probably not needed anymore).

	Because the output (receipt information) might contain several text mistakes, it might be desirable to let the user correct those mistakes before the receipt information is handed over to the COICOP classification algorithm. Whether or not user correction is desirable strongly depends on the input requirements of the COICOP classification algorithm.
REQ M2	Microservice performs COICOP classification on detected products and services
	The model (algorithm + training) classifies the product/service description to a COICOP.
	Integration of the model with the OCR microservice.
	Support for different COICOP classifications (the code should not hard-code one specific COICOP classification since NSIs are free to extend the 5-digit demanded classification).
REQ M3	Microservice sends notifications when results are available
	So that a pop-up in the app can inform the respondent that the scanned ticket is added to the overview on the day of the purchases.
	In case of multiple submitted receipts, the microservice will generate a notification for each receipt. It is up to the app (platform-specific) how to handle multiple notifications.
REQ M4	Microservice provides a service to retrieve results
	The output of the microservice can be requested by platform to e.g. include the user into quality control.

Non-functional requirements

Non-functional requirements	
REQ N1	The microservice should be independent from any specific HBS platform.
	The microservice has no dependency to other environments, and has an independent operation.

REQ N2	It must be possible to connect and communicate with the microservice from any HBS platform.
	The microservice receives input, and provides output making use of APIs.
REQ N3	The microservice must have a design in which algorithms (computer vision, AI, ML) can be easily improved/updated.
REQ N4	The service must be deployable at any institute/NSI (shareability).
	The microservices are provided as software packages in containers, which can be easily shared and deployed. Docker is a software that can host containers. Kubernetes is often used as software to orchestrate various containers.
REQ N5	The service must be scalable with the number of receipts it needs to handle.
	Kubernetes is a software used to orchestrate containers. By this Kubernetes allows to horizontally scale the containerised microservice depending to the number of receipts received.
REQ N6	Security by design
	Using the container technology barriers are created between various components used in the study setup, which deliver better privacy, security and maintainability, scalability and high availability. Communication between the platforms runs through APIs and https communication.
REQ N7	Privacy by design
	Using the container technology barriers are created between various components used in the study setup, which deliver better privacy, security and maintainability, scalability and high availability. Communication between the platforms runs through APIs and via UUIDs to avoid transferring personal information.
REQ N8	Support for localization
	Algorithms being applied by the microservice should be configurable or trainable (in case of ML) to support localization, which includes different languages, different currencies, date formats, dots vs commas etc. This is required to make the microservice shareable.

REQ N9	Offline vs online support (app)
	Parts of the microservice are/can be selected to be developed in a Library to run offline in an application. The library must take into account platform-dependency (Angular, ionic, Flutter …) to function.