Co-funded by the European Union

Smart Survey Implementation

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

Work package 3

Developing Smart Data Microservices

Deliverable 3.4: Smart advanced stage report

Version 1.1, 2025-06-26

Prepared by:

Joeri Minnen (hbits, Belgium), Pieter Beyens (hbits, Belgium), Ken Peersman (hbits, Belgium), Enak Cortebeeck (hbits, Belgium), Noël Mingels (C BS, Netherlands), Jonas Klingwort (CBS, Netherlands), Chris Lam (CBS, Netherlands), Tim de Jong (CBS, Netherlands), Yvonne Gootzen (CBS, Netherlands), Marco Puts (CBS, Netherlands), Jerome Olsen (Destatis, Germany), Joël Van Hoorde (Destatis, Germany), Adrian Montag (Destatis, Germany), Miriam Engel (Destatis, Germany), Fabrizio De Fausti (ISTAT, Italy), Claudia De Vitiis (ISTAT, Italy), Marco Terribili (ISTAT, Italy), Francesca Inglese (ISTAT, Italy)

Work package Leader:

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

Index

1. General introduction

This document serves as the Smart Advanced Report and provides a final overview of the work completed in Work Package 3 (WP3) of the SSI project.

The primary objective of WP3 is to develop microservices as part of the broader goal: to design, implement, and demonstrate the concept of Trusted Smart Surveys, delivering a proof of concept for a complete, end-to-end data collection process. WP3 operates at the Development level, where microservices are designed as platform-independent components.

Initially, three microservices were identified for development. However, as the project progressed, it was decided that each microservice would consist of two underlying components:

- A non-domain-specific part, which can be utilized across various statistical domains.
- A domain-specific part, tailored to a specific statistical domain, often incorporating country- and language-specific requirements.

Below is an overview of the developed microservices along with their respective leading partners:

- Receipt Scanning Microservice (overall lead hbits)
  - OCR Microservice – non-domain specific (lead hbits)
  - COICOP classification Microservice – HBS + country/language restrictions (lead Destatis/CBS)
- GeoService Microservice (overall lead hbits)
  - Geolocation microservice – non-domain specific (lead hbits/CBS)
  - HETUS classification Microservice – TUS + country/language restrictions (lead ISTAT)
- Energy Microservice (overall lead CBS)
  - Feasibility setup of study and roll-out

WP3 is responsible for developing non-domain-specific microservices as shareable environments for SSI/ESS and integrating them into the end-to-end data collection process. It is closely linked to WP2.2, which defines the AI/ML models that WP3 further develops.

Additionally, WP3 works on domain- and country-specific solutions, such as HETUS and COICOP classification, ensuring they are adapted and integrated into microservices. The requirements for this integration are discussed within WP3, while the actual implementation occurs at the country level.

WP3 also supports the integration of microservices with data collection platforms. Once integrated, WP2 can test the microservices in interaction with users, completing the end-to-end process.

Each microservice can contain multiple models or algorithms, which together form a process flow within the microservice. When one or more microservices are integrated into a data collection platform, they create a structured process for collecting and processing statistical data.

1.1. Objectives of WP3

The main objectives of WP3 are:

- Develop and containerise the selected microservices.
- Develop the APIs between the microservices and the core platforms.

- Document the microservices and APIs.
- Support platforms and NSIs to include the microservices in the core platforms.
- Perform a “pentest” (security, penetration) and stress test (load performance).
- 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.
- Keep and maintain a public GitHub repository to make the microservices available as open source.

1.2. Documentation strategy

This report presents the technical documentation for the developed microservices. It begins with an overview of the general microservices architecture, followed by detailed documentation of the microservices developed within WP3.

A consistent approach is applied to the microservices related to receipt scanning and geotracking. First, the business, functional, and non-functional requirements are outlined, serving as the foundation for the subsequent work. Each microservice consists of two parts: a non-domain-specific component and a domain-specific component. The non-domain-specific part allows integration into various data collection processes beyond the HBS and TUS processes envisioned in the SSI project. The domain-specific part is tailored to align with HBS/COICOP and TUS/HETUS guidelines. Despite this distinction, both parts follow the same structure: design, implementation, integration, and testing. The Energy Microservice has a more moderate approach showing a feasibility output.

The report concludes with details on the data collection platforms that integrate these microservices, namely the @CBS platform (CBS) and the MOTUS data collection platform (hbits). Additionally, INSEE and SSB assessed the feasibility of integrating the microservices into their respective data collection environments.

1.3. GitHub repository and demonstrations

WP3’s output is only partially captured in the written documentation. The most significant results are available in the GitHub repository (https://github.com/essnet-ssi; under the EUPL-v1.2 license), which not only hosts the code developed by WP3 but also provides detailed and specific guidance for National Statistical Institutes (NSIs) on integrating these environments into their own end-to-end data collection processes.

Another level of providing results is the demonstration of the developed solutions. During the SSI-project multiple demonstrations were given. These demonstrations are recorded and are made available via the CROS portal, which is described in the connected Deliverable 3.3.

2. Timeline

During the project a timeline was kept structuring the development and integration work, and to keep aligned with the usability task of WP2 (also called Test within this document).

2.1. Microservice Architecture and General Approach

The first key task in WP3 was to establish a general approach for designing a microservice software architecture. This architecture defines a structure of runtime elements that connect the microservices to the platform, adaptable based on specific microservice and platform needs.

The SSI consortium recommends that the ESS adopt this structure for newly developed microservices within NSIs. Additionally, it suggests containerising microservices, as documented in the project.

2.2. Receipt Scanning Microservice

The first microservice developed was the Receipt Scanning Microservice.

Development of the Receipt Scanning Microservice, including the OCR and COICOP microservices, began in May 2023 with the definition of functional and non-functional requirements. The progress was demonstrated to the consortium through two key presentations.

The first demonstration in October 2023 showcased three AI/ML models within the OCR microservice:

- Preparing ticket images by detecting contours and orientation.
- Detecting and recognizing text, positioning it in boxes, and performing OCR.
- Connecting extracted text to standardized labels for ticket elements.

The second demonstration in April 2024 showed enhancements in all three models and their integration into a sequential process. Upon receiving a ticket, the microservice automatically processes it through the models, stores the derived data in the microservice database, and makes it accessible via an API.

The OCR Microservice was designed to be language-, country-, and shop-independent, requiring training of the individual models, which is handled in WP2.2. A PDCA model, developed in WP4, was introduced for training and annotation of ticket data.

Development concluded in March 2024, with documentation compiled in this final report and code available in the GitHub repository. The COICOP classification microservice was developed under WP2 with support from WP3, and its outputs are included in this report.

From April 2024 onwards, hbits and CBS have the knowledge and code required to integrate the OCR Microservice into their respective platforms, MOTUS and @HBS. The integration, primarily for the Household Budget Survey (HBS), follows a structured process:

- A user takes or uploads a receipt via the MOTUS/@HBS app.
- The receipt is sent to the OCR Microservice, where it is processed.
- The extracted data is displayed to the user for review and editing.
- Once validated, the data is committed.

To ensure privacy and security, additional processing may occur within the platform's core environment before final data exchange. Once integration and technical testing were complete, WP2 conducted pilot studies with users. A large-scale field test, coordinated by the University of Mannheim, was presented in November 2024 to demonstrate progress.

A crucial output of the OCR Microservice is product descriptions, alongside prices and other purchase details. These descriptions require classification into COICOP codes, handled by WP2 through the COICOP microservice, which can – after the SSI project - be integrated into the end-to-end process once fully functional.

2.3. GeoService Microservice

The second microservice developed was the GeoService Microservice.

Development of the GeoService Microservice began in November 2023, following the same approach as the Receipt Scanning Microservice. The first phase involved defining functional and non-functional requirements, followed by microservice development.

The development was divided into:

- A non-domain-specific component, the Geolocation Microservice, which can be used in any system requiring geolocation data.
- A domain-specific component, the HETUS Classification Microservice, designed for HETUS data collection.

The Geolocation Microservice derives information on stop points, also called clusters, and transport modes using spatial and temporal parameters. Stop points are contextualized using third-party POI databases, such as OpenStreetMap or Google Places. Originally planned for completion by June 2024, development was extended to December 2024 to accommodate integration into data collection platforms like MOTUS.

By November 2024, WP2 utilized the Geolocation Microservice for pilot testing, ensuring it could effectively classify stop clusters and integrate external POI data. The transport mode classification model, however, was not yet mature for full implementation.

For Time Use Surveys (TUS), WP2 applies the HETUS Classification Microservice to predict activities related to the stop cluster. This process incorporates social and work-related background characteristics. To maintain privacy, the data collection platform acts as an intermediary between the Geolocation Microservice and the HETUS Classification Microservice.

A third demonstration in July 2024 showcased:

- Progress in the clustering and transport mode models.
- Predictions for HETUS activities based on stop clusters, POI data, and user background characteristics.
- Integration steps for incorporating the GeoService Microservice into MOTUS, supported by a UI/UX demonstration.

2.4. Energy Microservice

The Energy Microservice is the third microservice. And is considered under PoC development.

CBS carried out a feasibility study and roll-out. A substantial part of the work is described in WP2.

2.5. Real-Time Data Processing Considerations

A key challenge identified during the project is the need for real-time data processing. Since microservices interact with users in real time, ticket and geolocation data must be processed efficiently. During algorithm development, performance limitations of certain programming languages became evident. For example:

- R exhibited slower processing speeds compared to C++.
- Python was preferred over R for integration due to better library support for connecting with bus systems.

2.6. Project Collaboration and Review Process

The work described in this report has been discussed in online meetings and physical workshops organized by CBS in Heerlen, Destatis in Bonn, and hbits in Brussels. The chapters have been reviewed by work package leaders, and additional feedback was gathered from participating countries and WP3 experts.

Given its technical nature, this report is intended for NSI staff responsible for integrating microservices into end-to-end data collection processes. The development process involved many research and development actions. In case of the development of the algorithm for the prediction of the COICOP classification (Chapter 4, section 4.3) and the transport mode prediction (Chapter 5, section 5.3) an extra Annex was produced with more information.

In addition to this report, further resources, including demonstrations and source code, are available via Deliverable 3.3.

3. 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.

A microservice is seen as an independent environment and, is in that respect, not coupled to a specific data collection platform.

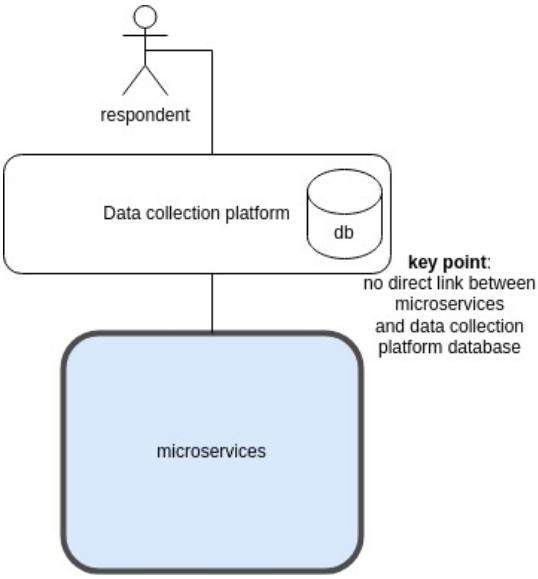
3.1. Views

Below the context, functional, information, concurrency, deployment and operational view are presented.

3.1.1. Context view

Figure 1 below shows how microservices fit into a general data collection platform architecture.

Figure 1: Context view - microservices as part of a system



The respondent is the end-user of the system and uses the functions (e.g. TUS, HBS) provided by the data collection platform. The platform contains the main database and benefits from the microservices to perform specific algorithms such as stop/track segmentation or receipt analysis.

The microservices are represented as a black box. Infrastructure between platform and microservices allows for communication using specific protocols and/or APIs.

It is important to mention that:

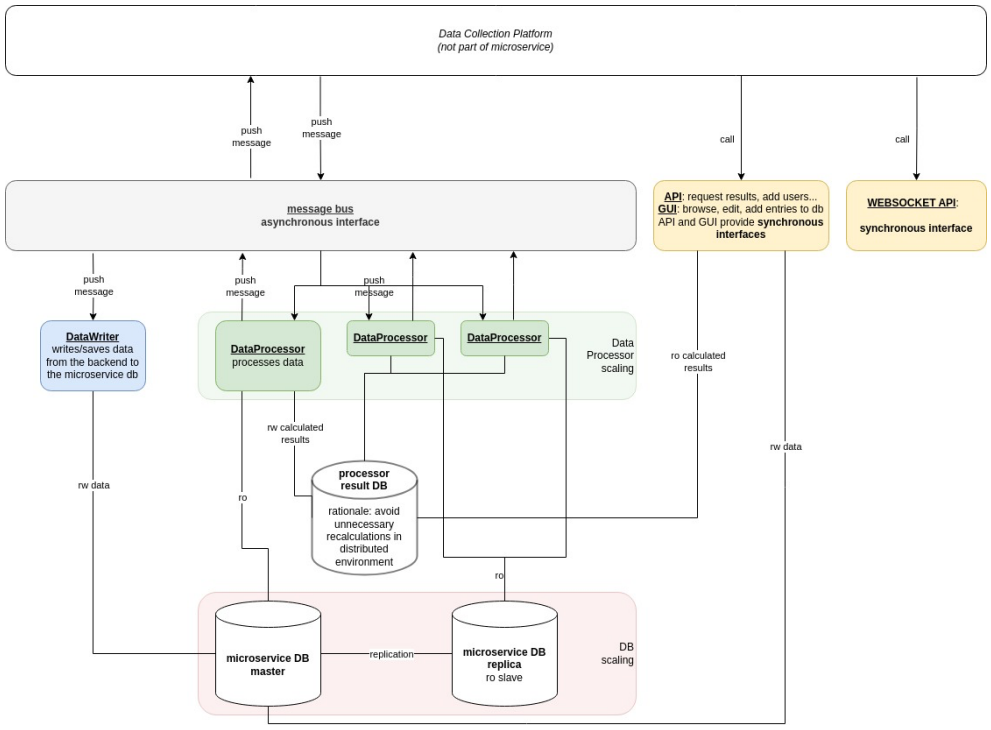
- there is no direct link between the respondent and the microservices: the data collection platform has full control over microservice usage (who/when).
- there is no direct link between the microservices and the main database. This means that the data collection platform has full control over which data is delivered to the

microservices. The exact mechanism which guarantees privacy will be explained in the "Regulation perspective" section.

3.1.2. Functional view

Figure 2 describes the different runtime functional elements of the microservice. Typically, each runtime element is deployed as a container (see Deployment View).

Figure 2: Functional view - overview of the runtime elements



The diagram is complex because it must be generic enough to support different types of microservices and different types of platforms. It is though not obligatory to implement all runtime elements if there is no need for it.

For example, the architecture could be simplified to DataProcessors alone when:

- all input data is directly delivered to the DataProcessor (without DataWriter and other DBs)
- the output data (~result) is also directly pushed on the message bus (without the need for an API or websocket)

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, geolocation point) to the microservice without being blocked for too long. The message bus is able to fulfil 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

The API and GUI are the synchronous interfaces 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 DB 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, it is an independent product which can be directly used via internet, and which can be used as a test platform.

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 messages 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.

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.

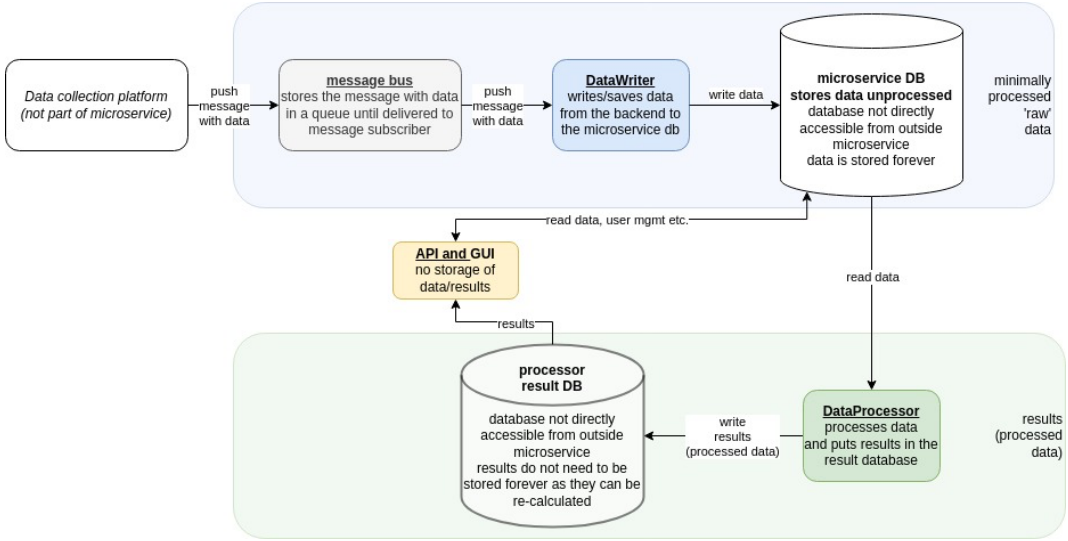
GUI is able to change data if needed so.

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

3.1.3. Information view

Figure 3 describes the way that the microservice stores, manipulates, manages, and distributes information. The diagram highlights the key points.

Figure 3: Information view - how the microservice handles information



Also here, if DBs and DataWriter are not required, the diagram can be simplified to bidirectional communication between MessageBus and DataProcessor.

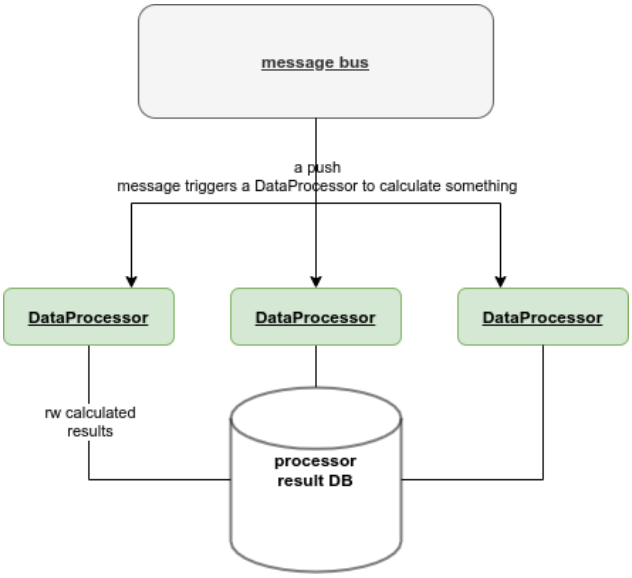
3.1.4. Concurrency view

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.

Figure 4: Concurrency view – how the microservice is horizontally scalable



Care must be taken to:

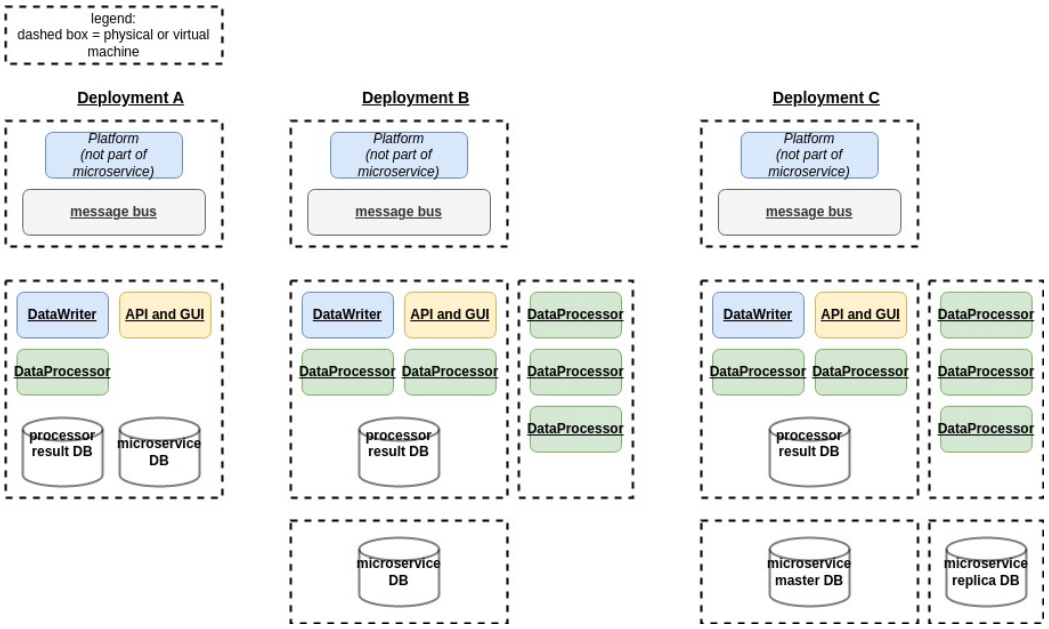
- avoid race conditions: if 2 DataProcessor calculate the same thing, then one DataProcessor might overwrite the results of the other, also if the other's results were more recent.
- avoid unnecessary recalculations: DataProcessors should check the database to make sure the results for its calculation are not already there. In that sense, the processor result DB also acts as 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.

3.1.5. 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. The figure below provides some examples.

Figure 5: Deployment view – different deployment strategies depending on the system's needs



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.

3.1.6. Operational view

This view relates to the installation and upgrade, and to the backup and recovery.

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 built 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).

3.2. Perspectives

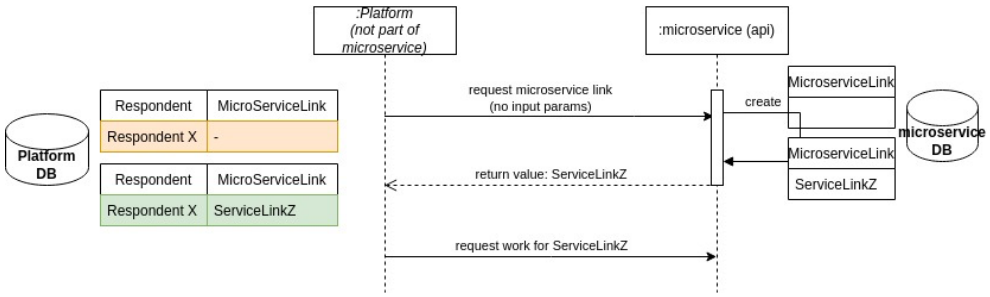
Perspectives are split in regulation and performance and scalability perspectives.

3.2.1. Regulation perspective

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:

Figure 6: Regulation perspective – decoupling the main database from the microservice



The platform (:Platform) request a microservice link to the microservice (:microservice), and creates a respondent-link mapping in its main database.

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.

3.2.2. 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

4. Receipt Scanning Microservice

The Receipt Scanning Microservice is the first environment being developed in this project. The development process began with outlining both the functional and non-functional requirements (4.1), followed by a description of its two main parts: the non-domain specific OCR microservice (4.2) and the (domain) HBS-specific COICOP classification microservice (4.3). For each part, the design, implementation, integration, and testing are discussed.

As a more general-purpose microservice, the OCR microservice is documented from a technical perspective. In contrast, the COICOP classification microservice is presented from a practical standpoint, reflecting established practices and experiences from Destatis and CBS. In doing so also strategies and techniques used by these institutions are presented.

The Receipt Scanning Microservice has been successfully integrated into the MOTUS platform, and integration with the @HBS platform is currently underway. Additionally, INSEE and SSB have evaluated the feasibility of integrating the Receipt Scanning Microservice into their respective platforms. Testing has been performed through the MOTUS platform in collaboration with WP2.

4.1. Functional and non-functional requirements of the Receipt Scanning Microservice

This first section of chapter 4 addresses the functional and non-functional requirements of the Receipt Scanning Microservice. 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.

Central to the SSI project 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 by CBS, MOTUS by hbits as well as the developments at SSB and INSEE.

An important criterion within the SSI project 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 in its latest update also takes into account the emergence of new data sources and use of new technologies.

Within SSI, WP3 is 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. In WP3 the Smart inclusion is realized by the development of microservices.

This section has a focus on Household Budget Survey (HBS) and the definition of requirements for the Receipt Scanning Microservice. The microservice is seen as a middle part software that is supportive to the household in reducing their burden to complete a consumption diary.

The microservice comprises two underlying microservices. The first (OCR microservice) is designed to perform Optical Character Recognition (OCR) and Document Understanding. The second is tasked with COICOP classification (COICOP classification Microservice), with the objective of classifying a product/service to a COICOP category.

Note: when referred to ‘ticket’ the document also means receipt or invoice, note, e-tickets, etc. All of these artefacts can be processed as an image, as long as the procedures were nationally trained for all variations.

4.1.1. 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 and spending habits 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 in intervals of 5 years. The last waves are from 2010 and 2015. In 2026 HBS will enter the IESS agreement and HBS will become a mandatory data delivery.

In the majority of member states, 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; consisting of product name and product price) that are bought and the total price of the ticket. Depending on the shop and product type, a ticket can also contain various different contexts to the purchase as well as information on reductions, amounts, units, 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 declining trend 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 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 digitally respond to all questionnaires as well as digitalize their ticket by adding purchase by purchase in a step-by-step manner in order to submit the entire ticket into a digital diary.

Notwithstanding the added value of these online applications, the burden on the participants remains high, and the process is 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. This will transform the way digital diaries have been used so far and is aimed to result in a true added value of digitalization. Also for the (end) users.

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 ease in which the user can verify, adapt, or even delete the output.

The following objectives are essential in reaching this goal:

- Objective 1: To adopt the generic architecture of a microservice

- Objective 2: To develop a Receipt Scanning Microservice that includes the use of an OCR model and a document understanding model
- Objective 3: To implement classification solutions (string matching, machine learning, 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 (who represent the households (citizens)).

HBS study

In this section HBS studies are being described as they provide the context in which the Receipt Scanning Microservice operates.

In HBS studies questionnaires and a consumption diary are completed by the households. At the moment household members arrive in 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. This is without saying that different NSIs (can) make use of different data collection strategies, and that e.g. the questionnaire and the diary can be in a different sequence or taken with a different tool.

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) 2 weeks/15 days. Left aside paper-and-pencil diaries, a household member partakes in 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 (some countries also include the fixed (repeating) costs into the diary, others collect this information via a questionnaire)
- 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.

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

- a COICOP-code[1]

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

[1] Note: NSI's can decide to classify a product/service to a COICOP also after the data collection stage.

- 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):

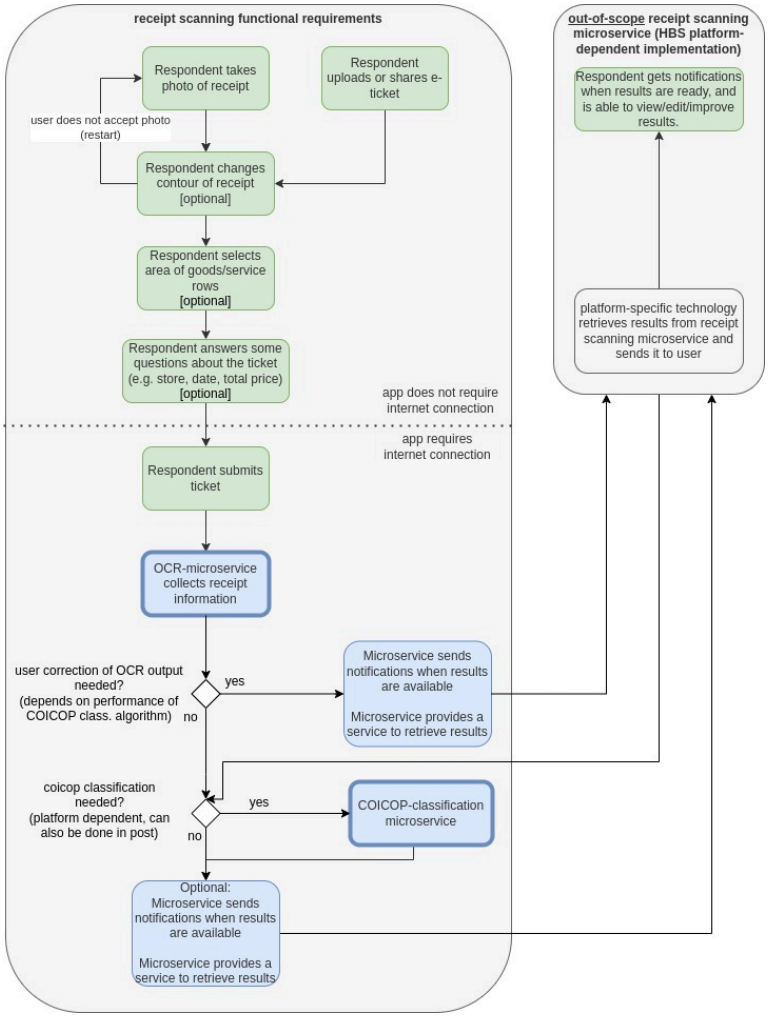
- number of items
- price per item
- quantity and metric/unit per item
- discount
- return

4.1.2. Functional requirements

Figure 7 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 an image of a receipt or invoice by taking a picture or by uploading a document (e-ticket) and (if needed to be able) to indicate the receipt location on the image 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').

Figure 7: Receipt scanning functional requirements – flow diagram



User handling

Respondent handling (green boxes)	
REQ R1a	Respondent takes photo of receipt within the HBS application
	Open the in-device functionality called from the app to take a photo
	Real-time camera opens In order to improve photo quality, the app can optionally hint the respondent about: • the contrast of the ticket vs the background • exposure • the detected contour of ticket (4 dots or polygon around receipt when stabilized) Contour detection is also done by the first part of the OCR microservice.

	Respondent takes picture when stabilized and the entire ticket is captured within one image.
	If the respondent is not satisfied with the photo, the process of taking a photo can be restarted.
	Additional quality checks might be performed later by the app software itself and/or the server-part microservice.
REQ R1b	Respondent uploads a document (e-ticket) via the HBS application (alternative to taking a photo of a receipt)
	Upload a document (e-ticket) from the local filesystem
	The application needs to accept images and PDFs in order to send this information over to the microservice.
REQ R1c	Respondent shares a document (e-ticket) from another application (e.g. store app) to the HBS application (alternative to taking a photo of a receipt)
	Not part of the microservice. Platform-specific (app) implementation.
REQ R1d	In case of a web app: scanning is 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 picture *with the smartphone* so that it automatically is loaded in the *web* app This requirement is a nice to have.
REQ R2	Respondent changes contour of receipt [optional]
	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 an automatic contour detection is embedded and works very well. Ideally, this step is not necessary (same as R3).
	Can be omitted when the microservice (a model within the microservice) is able to find the ticket within the image itself.

	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 [optional]
	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) [optional]
	The following questions could 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, together with gathered information (contour hint, expense items area, question answers)
	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 acknowledged by the user by pressing OK. If a failure occurs, the required action varies depending on the UI or platform. For instance, one might choose to wait for a Wi-Fi connection before attempting to upload the image again. The specific course of action is determined by the platform in use.

The Receipt Scanning Microservice has advanced to a point where it significantly reduces respondent tasks. For example, users no longer need to manually outline the receipt, as the current AI/ML model reliably detects its contours and orientation. Additionally, based on initial developments and testing, a well-trained document understanding model should achieve a high success rate in identifying key details such as the store, date, total price, and individual product entries on the receipt.

Despite this, it is important to mention that an AI model never provides guarantees. If certain input (e.g. shop name) is *required or essential*, then a respondent's input or correction is necessary making it a decision of the platform owner to include certain User handling, yes or no.

Microservice

Microservice (blue boxes)	
REQ M1	Microservice collects receipt information
	The service is best effort and should try to retrieve the following receipt information (if available on the receipt): • 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 received 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). However, the aim is to accurately retrieve this information from the ticket through OCR and document understanding.
	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 COICOP Classification Microservice together 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
	A pop-up in the app informs 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 the connected platform to e.g. include the user into quality control.

4.1.3. 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.
	The microservice must be designed in such a way that several container instances can be running in parallel. The container platform (e.g. Kubernetes) should support horizontal scaling so that the system is able to handle the expected number of receipts.
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.

4.2. OCR microservice (Receipt Scanning Microservice – part 1)

The Receipt Scanning Microservice is supported by two microservices: the OCR Microservice (part 1) and the COICOP Classification Microservice (part 2).

The second section of Chapter 4 deals with the OCR Microservice holding three AI/ML models. This part is non-domain specific. In this section the software design, software implementation, and platform integration of the OCR Microservice are discussed. Software design and implementation explain the inner workings of the microservice, while platform integration explains how the microservice should be integrated technically in a platform. The responsibility for platform adaptations and UI elements lies with the platform developers and is not part of the SSI scope. The integration of the OCR microservice by hbits (MOTUS) is realised, the integration on the side of CBS (@HBS) has not been realised within the boundaries of the SSI project. The CBS has given notice of their state of play later in the document. INSEE and SSB have been given information during an introduction meeting with IT personnel. This information is accompanied with the documentation to let them evaluate the feasibility to integrate the microservice into their production environment. A short report is made on this action point. Other countries are invited to request information on how to integrate this shareable component into their platforms.

Code is to be found in the Github repository https://github.com/essnet-ssi/receiptscanner-ssi.

Demonstrations that were given are available on OpenSocial via https://cros.ec.europa.eu/book-page/information-session-march-2024-wp3 and via https://cros.ec.europa.eu/book-page/information-session-november-2024-wp2.

As a last point this chapter will discuss test information.

4.2.1. Design

The core of the microservice is the OCR pipeline which takes a receipt in the form of:

- a set of images (typically only one image), or

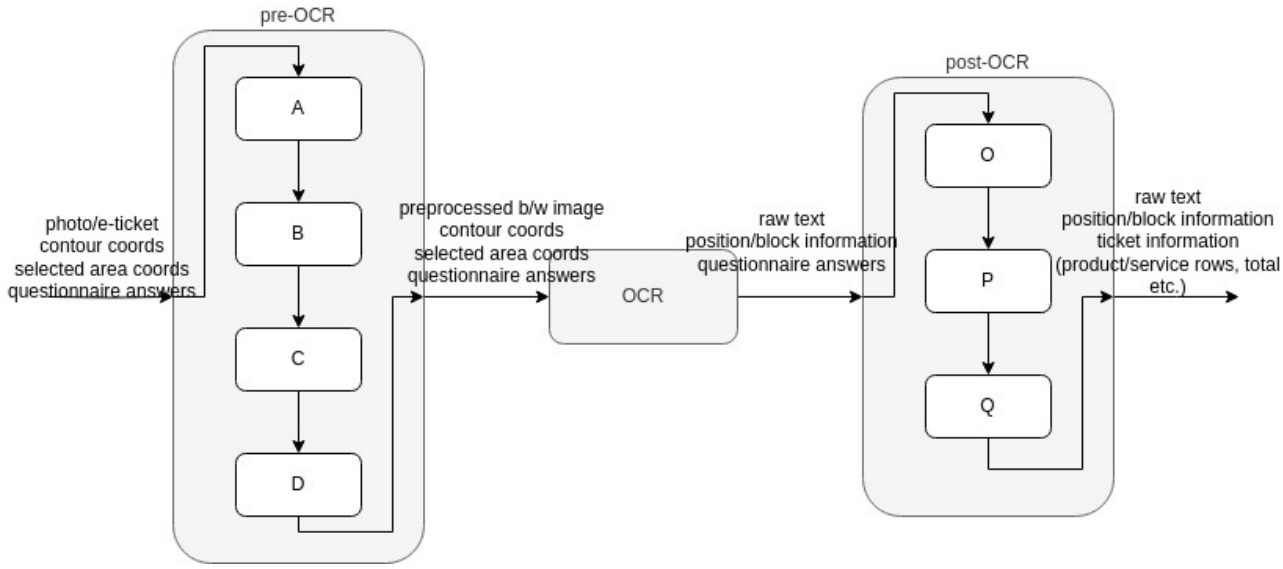
- a (pdf) e-ticket (which accordingly is processed as an image)

Once the receipt has arrived it goes automatically and sequentially through 3 steps:

- Pre-OCR: receipt detection and rotational correction
- OCR: text detection and text recognition
- Post-OCR: machine learning-based and rule-based logic
  - document (receipt) understanding, and
  - final post-processing which has as an output a json file.

The OCR Microservice returns all available information, so that that information can be used by the platform (and its UI/mobile and web application) and/or accordingly to the COICOP Classification Microservice. For example, if a product row cannot be parsed (e.g. price cannot be found), it returns the whole (unparsed) row.

Figure 8: Receipt scanner processing pipeline



The letters (A-D) indicate a pre-OCR processing function/block in the whole chain. E.g. a receipt detection function, orientation correction etc. The letters O-Q are similar but then for post-OCR e.g. document understanding, combining detected rows, validating the total price etc.

Pre-OCR

The main responsibilities of pre-OCR steps are receipt detection, orientation correction and image cropping. Other techniques like noise removal were listed in a MOSCOW analysis and were taken in consideration but did not improve the results and are therefore not taken on board.

The input is a single receipt in the form of:

- a set of images (this means that the microservice can deal with multiple images), or
- an e-ticket (pdf). The pdf is converted into a set of images which then follows the same flow as normal images.
- Optionally: contour coordinates

The output of the pre-OCR step is a correctly oriented, cropped receipt image.

For receipt detection[2], two methods have been deployed:

- semantic segmentation, and
- object detection

Semantic segmentation

1. The pre-OCR pipeline takes as input an image of a receipt.
2. It then segments the image. Which means that it labels each pixel of the receipt as belonging to the receipt or not.
3. It then draws the smallest possible bounding rectangle around the pixels classified as belonging to the receipt.
4. It uses heuristics to try and correctly orient the receipt, such that it is upright.
5. It crops the receipt using the bounding rectangle.
6. It then adjusts the orientation again slightly, using paddle_OCR to fine tune the orientation.

Model training

This section outlines the training process for the SegFormer model, designed to segment and identify the area of store receipt on photos.

- URLs
A guide for training this model can also be found in the code repository:

https://github.com/essnet-ssi/receiptscanner-ssi/blob/main/src/ocr_microservice/model_training/segformer/train.md

The training script itself is located here:

https://github.com/essnet-ssi/receiptscanner-ssi/tree/main/src/ocr_microservice/model_training/segformer.py

- Data Preparation
Labelled Dataset: The training data consists of labelled images exported from Label Studio.
The exported dataset is provided as a CSV file containing:

Image Paths: Paths referencing receipt images.

Polygon Points: JSON-formatted polygon coordinates indicating regions of interest on the receipts.

[2] These processes have utilised open-source-code and models. This is also true for PaddleOCR. PaddleOCR stands as an open-source optical character recognition (OCR) solution crafted by PaddlePaddle, the deep learning platform nurtured by Baidu. Its primary objective is precise text extraction from images, boasting proficiency across diverse languages and font types. Employing cutting-edge deep learning architectures, PaddleOCR excels in both text detection and recognition tasks. Its adaptability shines through the provision of several pre-trained models, each tailored for distinct scenarios like scene text, ID card, and table structure recognition. Offering standalone models and end-to-end OCR pipelines, it accommodates various use cases and deployment settings. Appreciated for its user-friendly interface, robust performance, and comprehensive documentation, PaddleOCR has garnered favor within research and developer circles. Its utility spans document digitization, image text extraction, and intelligent document processing. Furthermore, its open-source nature fosters community engagement, permitting customization to suit specific language or application needs.

All referenced images must be stored locally within a designated directory.

- Training Procedure
The segmentation model training is managed through the primary script train.py. The following parameters guide the training process:

- Batch Size: 8
- Learning Rate: 1e-4
- Number of Epochs: 500
- Loss Function: BCEWithLogitsLoss (binary cross-entropy with logits), using positive class weighting

- Training Steps:

1. Environment Setup: Install necessary Python packages using: pip install pandas torch torchvision matplotlib pillow tqdm
2. Data Organization:
   - Place receipt images in the images/ directory.
   - Include the CSV file containing label annotations (labels.csv) in the project's root.
3. Configure and Train: Adjust paths within train.py: image_dir = './images/' label_csv = './labels.csv' result_dir = './outputs/' Begin training by executing: python train.py
4. Model Outputs: Model checkpoints are saved during training as follows: images/checkpoints/model_epoch_X.pth
5. Visualization: A visualization feature in the dataset class allows inspection of original and segmented images to verify data accuracy.
6. Using the trained model: To use the model, move it to src/ocr_microservice/ocr_pipeline/resources/segformer, and update src/ocr_microservice/ocr_pipeline/config/default.py to match the name of the trained model.

Object detection

An alternative method is to use object detection for receipt detection. In this case a rectangular box is drawn around the receipt in the image. The detection is very accurate but less fine-grained than semantic segmentation. Furthermore, it requires quite some processing to correct the receipt orientation. The orientation of the image is corrected by sequentially applying PaddleOCR to derive text orientation.

The used AI model for object detection is YOLOS. For more information, please visit this link: https://huggingface.co/docs/transformers/model_doc/yolos

Model training

The training of the object detection model holds 4 steps:

- Collect images and divide them into 3 subsets: training, test and validation.
  - Training: for training the YOLOS model,
  - Test: for testing the YOLOS model while it is being trained,
  - Validation: for validating the final YOLOS output (which is not used by YOLOS itself).

- Rotate all images: This step is needed to make the crop of the receipt as tight as possible.
- Annotate and create Coco output: Use a tool to add bounding boxes for object detection to the images (e.g. Label Studio), export in Coco format (note: validation images should not be annotated because they are not being used in YOLOS training).
- Training and retraining

More detailed description can be found in the Github repository under the folder: https://github.com/essnet-ssi/receiptscanner-ssi/tree/main/src/ocr_microservice/model_training/lilt.

OCR

Several OCR models were compared:

- EasyOCR
- Tesseract
- PaddleOCR
- TrOCR

A fine-tuned PaddleOCR model provided the best accuracy for this task. The model was trained on a small, labelled dataset of ~300 receipts.

Model training

Training can be split into two (or three, depending if the CLS model also is counted) groups: detection model and the recognition model.

- The detection model detects words in an image and binds them to a bounding box with annotation.
- The recognition model recognizes shapes within that bounding box and tries to classify them (by 'assigning' meaning to them e.g. characters).

The following specific steps have to be taken in order training own PaddleOCR models:

-1- Convert the image data and annotations in a required format

For the de Detector, at least two .txt files are needed as training and validation sets (make three, to also have an evaluation set). The format of these text-files is as follows (without spaces):

image_name.ext \t [{"points":[[x1,y1],[x2,y2],[x3,y3],[x4,y4]], "transcription":text_annotation, {"points":[...]}] \n

With each image a new line. The same amount of .txt files should be made when training the Recognizer (training, test, eval) with the following format (without spaces):

image_name.ext \t Annotation \n

-2- Download models and sources form the PaddleOCR Website

These can be downloaded via the Github project page of PaddleOCR. You can use an existing, already trained, model for finetuning (look up their models such as en_PP-OCRv3_rec) or you can download other pre-trained models like the MobileNet_V3_Large. The available models for detectors and recognizers vary.

-3- Change the .yml config files

When cloning the PaddleOCR project, configs for training are already included. These can be found within the configs folder, and are seperated per type of training configs (folder det for detector, folder rec for regocnizer etc.). Within these folders, .yml files can be found for specific training routines. You can also create one an own, but going with an existing config like det_mv3_db.yml is easier. Change the following fields: Global.pretrained_model (path of the weights that have been downloaded in step 2), Train.dataset.data_dir and Train.dataset.label_file_list (path of training images and training .txt-file respectively) and finally under Eval.dataset.data_dir and Eval.dataset.label_file_list (path of test images and test .txt-file). Other fields (like output directory, learning rate, loss-functions etc.) can also be set here.

-4- Start training

For training, use the following commands:

```
#Training
!python tools/train.py -c config/det/det_mv3_db.yml \
-o Global.pretrained_model=./pretrained/MobileNetV3_large_x0_5_pretrained
```

This the training-script recognizes whether it is a detector-training or recognizer-training. Do not forget to enable/disable gpu-training in the config. GPU/CPU training is set in the config.

-5- Inference

The last step is to convert the model and export it to a usable format. This can be done through the following commands:

```
!python tools/export_model.py -c config/det/det_mv3_db.yml -o
Global.pretrained_model="./output/trained_model/best_accuracy"
Global.save_inference_dir="./output/trained_inference/"
```

-6- Use it

Load the weights when running PaddleOCR. This can be done by entering the rec_model_dir and det_model_dir when

Post-OCR

The output of the OCR step includes text and text locations (bounding boxes). That information is used:

- to understand the receipt i.e. trying to give a meaning to the recognized text (by OCR),
- to correct OCR mistakes (date, time corrections etc.),
- to produce a final json output which contains all receipt details (and some metadata).

Receipt understanding

In order to understand the receipts, the pipeline applies a fine-tuned model of LiLT. See also here for more information: https://arxiv.org/abs/2202.13669.

LiLT combines text and layout (text position) information to label a text box. A variety of labels were defined within the SSI project to arrive to standardisation, ranging from store address to tax price. See table below.

Table 1: Label descriptions for receipt text

Id	Label	Description
0	O	ignore
1	I-date_text	date string e.g. Date
2	I-date_value	date e.g. 10/9/23, Thursday 3 aug 2019
3	I-time_text	time string e.g. Time
4	I-time_value	time value e.g. 10:29
5	I-datetime	combination of date and time (because combined by OCR) e.g. 3-12-2021 15:04
6	I-heading	Main heading of the ticket, typically between store details and products e.g. Receipt, Account, rekening, klantenbon etc., to distinguish from item.header
7	I-unused8	
8	I-unused7	
9	I-unused6	
10	I-tax.header	tax table: header e.g. Tax, Incl., Excl.
11	I-tax.description	tax table: typically percentage or total e.g. 10%, total
12	I-tax.price	tax table: tax price e.g. 9 EUR (which is 10% of 90 EUR)
13	I-tax.price_excl	tax table: total price excluding tax i.e. total cost without tax e.g. 90 EUR
14	I-tax.price_incl	tax table: total price including tax i.e. what the customer had to pay e.g. 99 EUR
15	I-unused5	
16	I-unused4	
17	I-unused3	
18	I-unused2	
19	I-unused1	

20	I-store.name	name of the store e.g. lidl, Coopcentrum Bert Stuut
21	I-store.address	address e.g. 9693 AE Bad Nieuweschans, Hoofdstraat 41
22	I-store.phone	phone e.g. Tel:0597-621678
23	I-store.email	email e.g. Email:stk@jumbo.com
24	I-store.website	website e.g. WWW.KWANTUM.NL
25	I-store.tax_id	tax identification number e.g. B0840.591.904
26	I-store.unused3	
27	I-store.unused2	
28	I-store.unused1	
29	I-store.etc	belongs to store but combination of several items e.g. Rotterdam 010 414 46 98 (which is part of address and tel. nr)
30	I-item.header	product table: header e.g. EUR, TOT, Price, Description
31	I-item.quantity	product table: quantity, how many items of this product e.g. 2 OR wieght e.g. 0.213kg
32	I-item.description	product table: description e.g. tomatos
33	I-item.unit_price	product table: unit price e.g. 1.00, 1.02 EUR/kg
34	I-item.price	product table: total price of this item row e.g. 2.00 (which is 2x 1.00 in this example)
35	I-item.id	product table: number, code or id of item e.g. 2751338001839, Article 20470047
36	I-item.discount_description	product table: kind of discount e.g. set 2 for 9.99, DISCOUNT
37	I-item.discount_price	product table: discount price e.g. -1,31, 1.31
38	I-item.etc	product table: sometimes contains non-product items e.g. BONUS CARD, Parking ticket etc.
39	I-item.unused11	
40	I-item.unused10	reserved for e.g. multi-line support
41	I-item.unused9	
42	I-item.unused8	
43	I-item.unused7	

44	I-item.unused6	
45	I-item.unused5	
46	I-item.unused4	
47	I-item.unused3	
48	I-item.unused2	
49	I-item.unused1	
50	I-sub_total.text	subtotal string e.g. SUBTOTAL
51	I-sub_total.price	subtotal price e.g. 95
52	I-sub_total.discount_text	receipt discount string e.g. DISCOUNT, KORTING
53	I-sub_total.discount_price	receipt discount price e.g. -3,99
54	I-sub_total.discount_item_text	discount item table (overview of discounts, similar to product items but then for discounts): text e.g. discount
55	I-sub_total.discount_item_price	dicount item table: price e.g. -45,6 EUR
56	I-sub_total.tax_text	tax text (not from the tax table! see line 10-14) e.g. TAX
57	I-sub_total.tax_price	tax price e.g. 9 EUR
58	I-sub_total.tax_excl_text	total price excluding tax (string) e.g. Ex TAX
59	I-sub_total.tax_excl_price	total price excluding tax (price) e.g. 90
60	I-sub_total.tax_incl_text	total price including tax (string) e.g. TOTAL incl TAX
61	I-sub_total.tax_incl_price	total price including tax (price) e.g. 99
62	I-sub_total.service_text	service text e.g. SERVICE 3%
63	I-sub_total.service_price	service price e.g. 3 EUR
64	I-sub_total.item_rows_text	total number of item rows (string) e.g. ? (remove?)
65	I-sub_total.item_rows_value	total number of item rows (value) e.g. 10 (remove?)
66	I-sub_total.quantity_text	total number of items (value) e.g. ITEM COUNT
67	I-sub_total.quantity_value	total number of items (value) e.g. 20
68	I-sub_total.etc_text	related to subtotal (string) e.g. ROUNDING

69	I-sub_total.etc_price	related to subtotal (price) e.g. 0,00
70	I-total.text	total string e.g. TOTAL, TOTAAL
71	I-total.price	total price (typically incl tax) e.g. 99,7
72	I-total.rounded_text	rounded total (in case cash cannot be paid in certain amounts) e.g. ROUNDED TOTAL
73	I-total.rounded_price	rounded total price e.g. 100
74	I-total.unused4	
75	I-total.unused3	
76	I-total.unused2	
77	I-total.unused1	
78	I-total.etc_text	related to total but no other correct label (remove?)
79	I-total.etc_price	related to total but no other correct label (remove?)
80	I-payment.cash_text	string which indicates payment in cash e.g. CASH
81	I-payment.cash_price	value of cash payment e.g. 100
82	I-payment.change_text	change string e.g. CHANGE
83	I-payment.change_price	amount of change e.g. 1.00
84	I-payment.other_text	other payment type e.g. CARD, MEASTRO
85	I-payment.other_price	other payment price e.g. 99
86	I-payment.details_total_text	payment details (esp credit card details on the receipt) can also contain the total price e.g. total
87	I-payment.details_total_price	payment details (esp credit card details on the receipt) can also contain the total price e.g. 100,00
88	I-payment.etc_text	related to payment but no other correct label (remove?)
89	I-payment.etc_price	related to payment but no other correct label (remove?)

Model training

The training of the receipt understanding model holds 4 steps:

- Collect receipts: Images need to be correctly oriented and cropped. Collect receipts in 3 sets:
  - training: for training the LiLT model,

    - test: for testing the LiLT model while it is being trained,
    - validation: for validation after the LiLT has been trained. The validation set does not need to be annotated (except if needed for automatic regression testing).
- Apply OCR to get boxes with text: Tools: e.g. PaddleOCR.
- Annotate and compose the dataset. Every OCR detected text box must get a label (see table of labels in previous section, including the O label, first row to when a text box needs to become ignored). Tools: e.g. Excel.
- Training and retraining.

More detailed description can be found in the Git repository:

https://github.com/essnet-ssi/receiptscanner-ssi/blob/main/src/ocr_microservice/model_training/lilt/README.md

Corrections

This section explains the post-processing methods applied after the OCR step to extract and correct specific receipt information, enhancing accuracy and usability.

- URLs
Relevant code can be found here:

https://github.com/essnet-ssi/receiptscanner-ssi/tree/cbs-final/src/ocr_microservice/ocr_pipeline/rule_based_extractor

- Extraction of Key Receipt Information
Post-processing involves extracting structured information from raw OCR results using text analysis techniques. The main steps include:

- Receipt Date: Extracted using regular expressions that identify date patterns in the OCR output.
- Total Price: Identified by scanning for specific keywords (configurable in total_price_extractor.py). When keywords are detected, the corresponding numerical price values on the same line are captured.
- Shop Name: Determined by matching specific keywords listed in shop_extractor.py, facilitating accurate identification.
- Products and Prices: Columns of products and corresponding prices are identified by examining consistent alignment and similar text height, enabling accurate extraction of product-price relationships.

Each step of the extraction can be customized or expanded according to the project's requirements.

Final post-processing

Given the output of receipt understanding and its corrections, a json output is being generated which contains all found information as well as metadata (i.e. where the information came from).

The json output has the following fields (for explanation please view the aforementioned label list):

- date
- time
- tax_table which has rows with following fields:

    - description
    - price:
    - price_excl
    - price_incl
- store which has next fields
    - name
    - address
    - phone
    - email
    - website
    - tax_id
    - etc
- item_table which has rows with following fields:
    - quantity
    - description
    - unit_price
    - price
    - id
    - discount_description
    - discount_price
    - etc
- sub_total
    - price
    - discount_price
    - discount_item_text
    - discount_item_price
    - tax_price
    - tax_excl_price
    - tax_incl_price
    - service_price
    - item_rows
    - quantity
    - etc_price
- total
    - price
    - rounded_price
    - etc_price
- payment
    - cash_price
    - change_price
    - other_price
    - details_total_price
    - etc_price

Simplified example of a json output:

{

“receipt”: {

```
        "date": {
                "text": "23-04-2024",
                "corrected": "2024-04-23",
                "bbox": [[422, 2253, 574, 2280],[422, 2253, 574, 2280]],
                "ocr_confidence": 0.9,
                "du_confidence": 1.0
        },
        ...
    }
}
```


Explanation of fields:

- text: text as recognized by the OCR engine
- corrected: post-processed text e.g. normalized date
- bbox: bounding box as detected by the OCR engine
- ocr_confidence: OCR engine confidence score
- du_confidence: document understanding confidence score

4.2.2. Implementation

The OCR pipeline is a python runtime. The code is available in the Github repository (https://github.com/essnet-ssi/receiptscanner-ssi). The code also includes an example Docker file for integration into a platform as is discussed in the next section.

Depending on the platform, the python runtime can be deployed as a (containerized) microservice in various ways. In the following sections, the data collection platform developers discuss their specific integration. In sequential order these are the platform of hbits, CBS, INSEE and SSB.

4.2.3. Integration: hbits MOTUS platform

MOTUS is developed by hbits, as a spin-off of the Vrije Universiteit Brussel. How the integration of the OCR microservice is viewed on the user side can be seen in the demonstration videos (see Deliverable 3.3). These videos provide a view on how the MOTUS application presents, on the user side, the output of the OCR microservice. Further information is available via the usability tests in WP2.

Below the integration of the OCR microservice in MOTUS is discussed, as well as how MOTUS communicates via its API.

Integration in MOTUS

The integration in MOTUS discusses the views as explained in Chapter 3 of the Microservice software architecture.

Context view

Figure 9: Context view of receipt scanner integration in the MOTUS platform

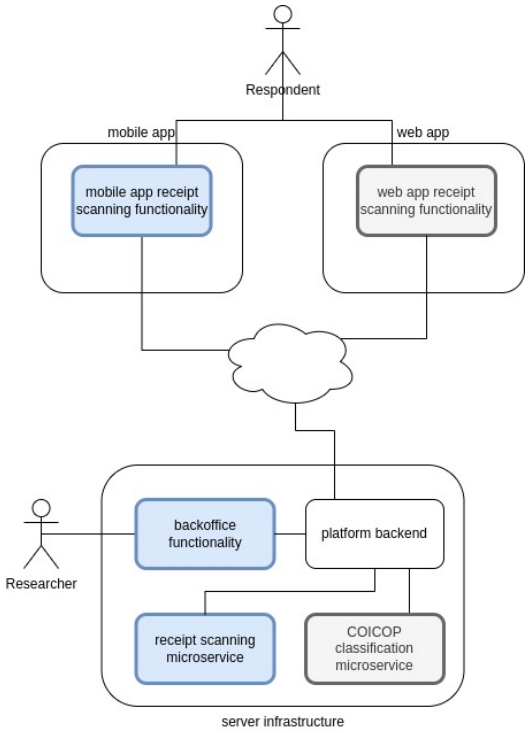


Table 2 describes the functionalities of the different components.

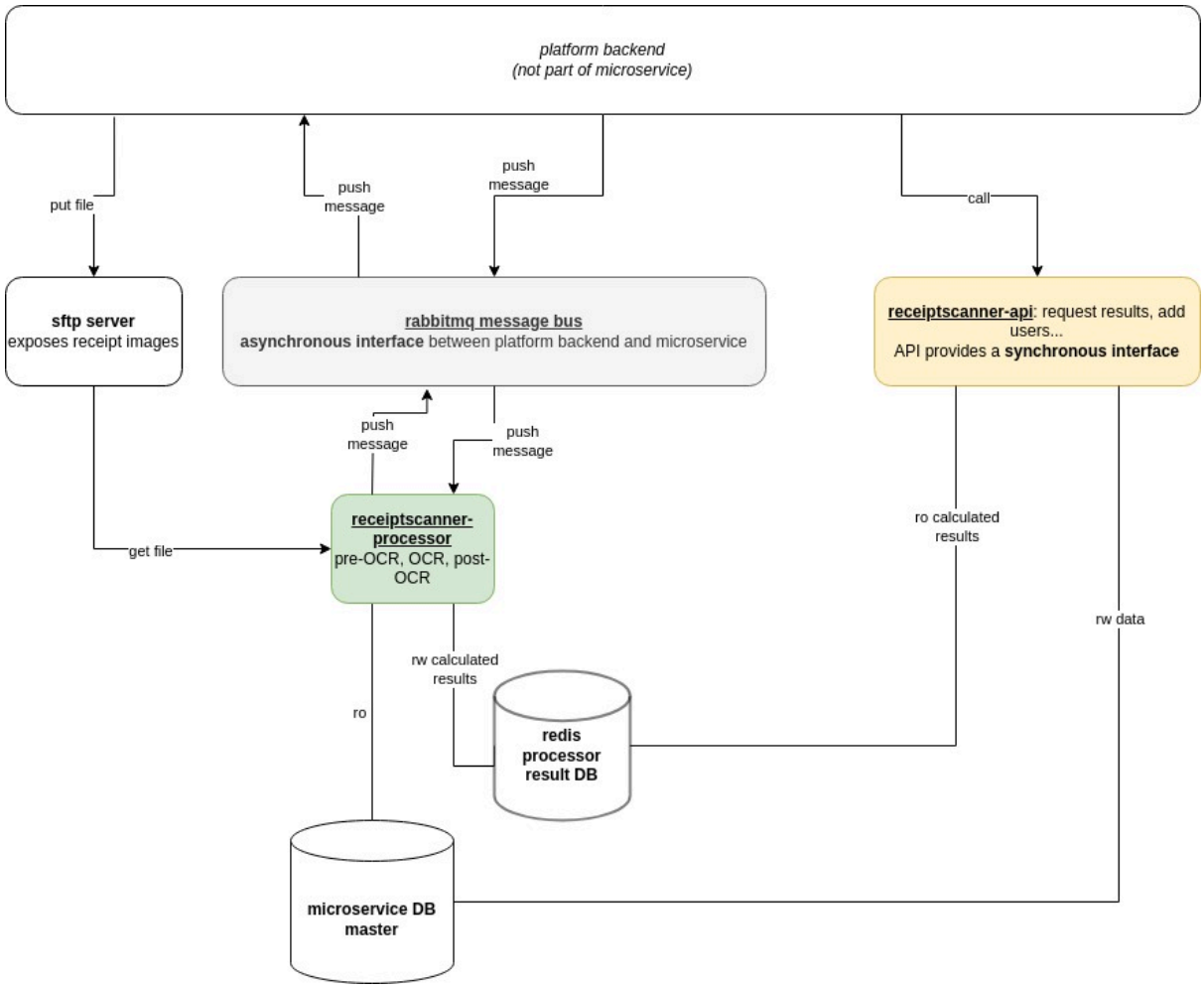
Table 2: Description of functionalities of the MOTUS components

Component	Functionalities
Mobile app receipt scanning functionality	• Camera view (take photo) • Gallery view (select image from gallery) • File view (select pdf from local phone storage) • Photo view + contour editing • Pre-process image: decrease resolution Send photo + coordinate data to backend
Web app receipt scanning functionality	Scanning functionality is not (yet) foreseen. The output of the OCR microservice is nevertheless available via the web app.
Receipt Scanning Microservice	From receipt photo + coordinate data to receipt information (store, total, product/service rows, etc.)
Backoffice functionality	Researcher can review respondent receipt
Respondent functionality	Respondent receives tentative data via the mobile and/or web app, can edit the ticket to commit the ticket

Functional view

Figure 10 below shows the functional view (runtime elements) of the Receipt Scanning Microservice.

Figure 10: Functional view of the receipt scanner microservice in the MOTUS platform



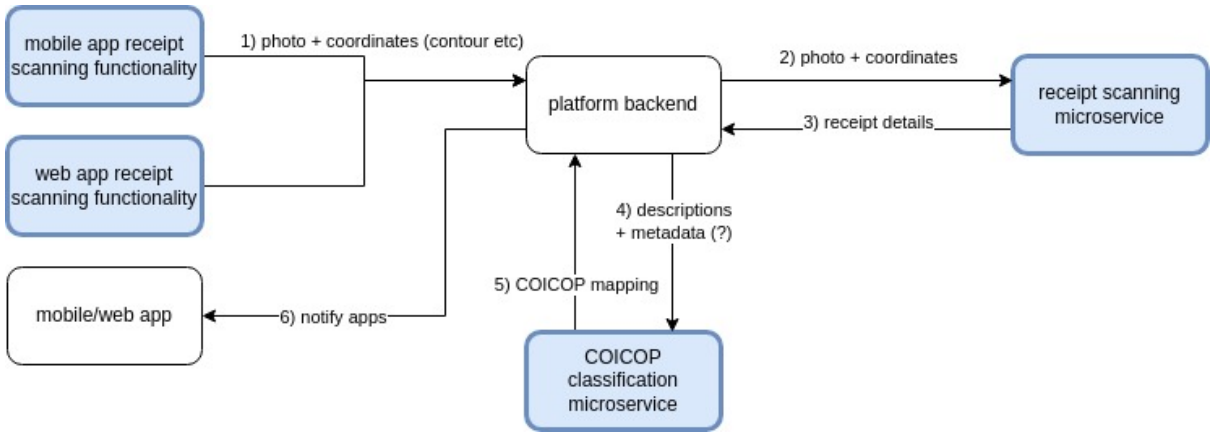
The components have the following functionalities:

- *Sftp server*: makes receipt images available to the microservice
- *Rabbitmq message bus*: used for asynchronous communication between backend and microservice. Messages are requests for processing and notifications when processing is finished
- *Mariadb microservice database*: stores images (which can be coupled to respondents in the backend)
- *Redis result database*: stores processing results
- *Receiptscanner-processor*: downloads a receipt from the sftp server, processing it (pre-OCR, OCR and post-OCR), publishes the results in the redis database and notifies the backend via rabbitmq
- *Receiptscanner-api*: used by the backend to manage scanners (which can be coupled to respondents in the backend) and to retrieve results

Information view

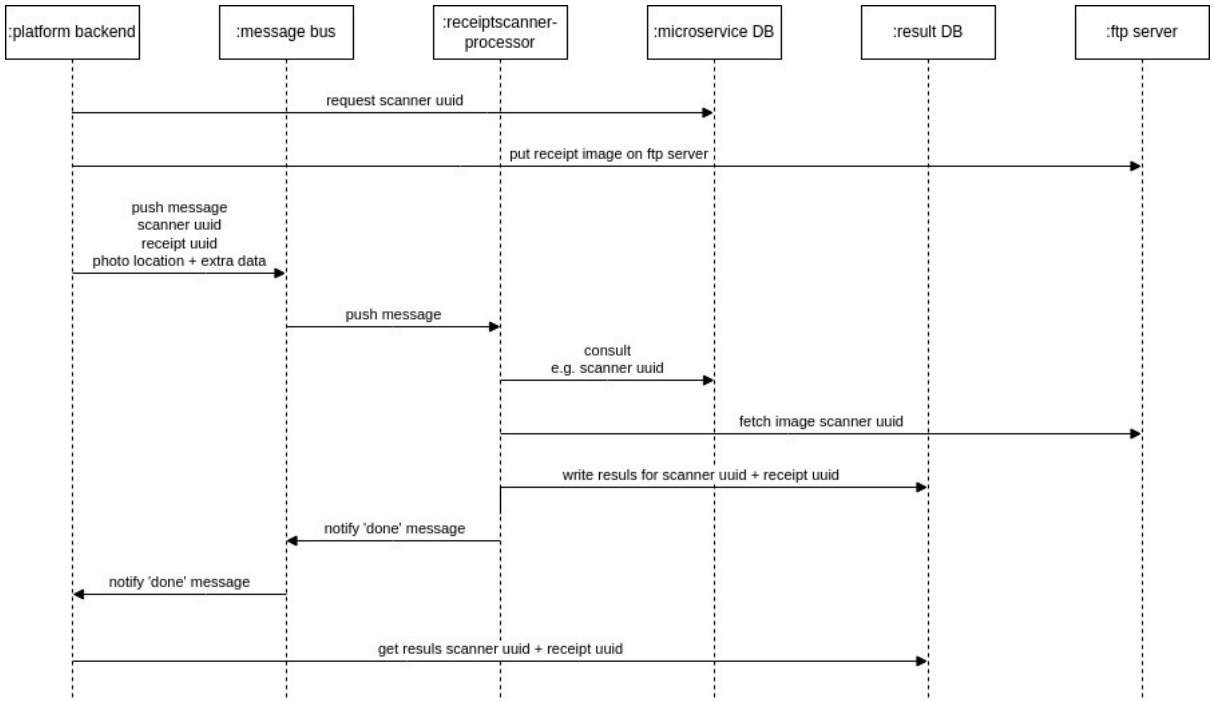
Figure 11 below shows the informational view of the receipt scanner microservice.

Figure 11: Information view of the receipt scanner in the MOTUS platform



A more detailed sequence diagram (Figure 12) highlights the different calls between the various runtime elements.

Figure 12: Sequence diagram of receipt scanner integration in the MOTUS platform



Flow:

- Backend requests the receiptscanner-api to create a scanner entry for a respondent (if Receipt Scanning Microservice is allowed and activated for that respondent via the backoffice configuration)
- When the backend receives a receipt image

    - it saves the image on the sftp server. Files are organized by scanner-UUID directories. Filenames are date and UUID based.
    - it sends an asynchronous message (*receipt.process*) on the rabbitmq bus to process the receipt image.
- Receiptscanner-processor
    - reads the processing request message from rabbitmq
    - fetches the receipt image via sftp
    - processes the image
    - puts the results in the redis db
    - notifies the backend that processing is done via a rabbitmq asynchronous message (*receipt.processed*)
- Backend
    - reads the notification message
    - requests the results from the receiptscanner-api
    - returns the results to the respondent when requested by the mobile/web app.

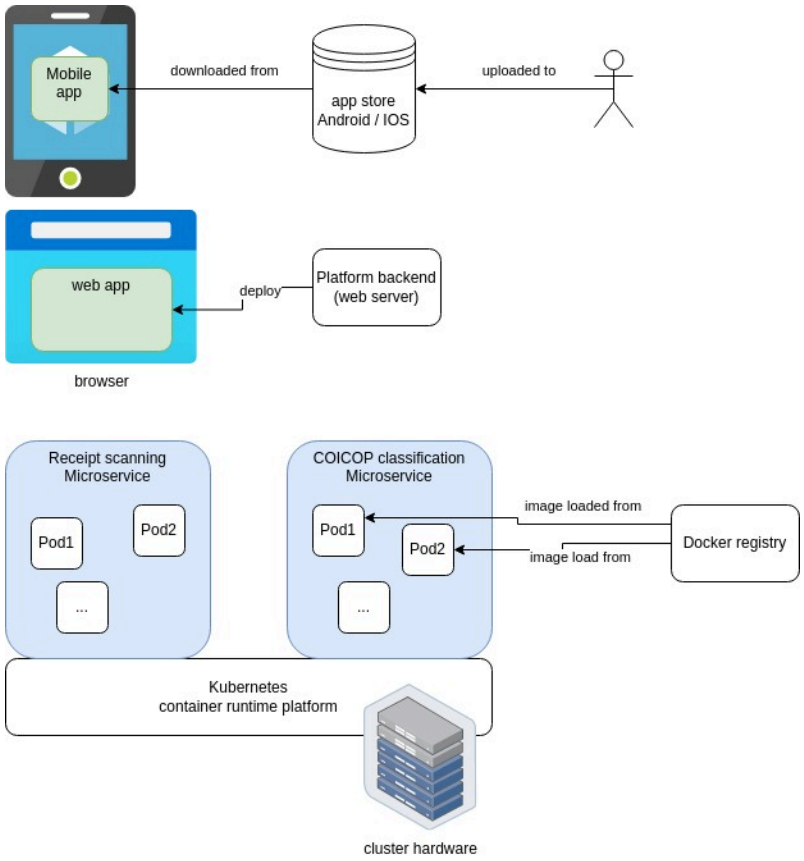
Concurrency view

In mobile/web app is respondent identification via authentication used to isolate backend requests from each other.

The platform backend and microservices are able to handle multiple requests concurrently. The generic microservice architecture (see chapter 3) supports concurrent and parallel handling of requests by queuing requests and scaling data processors.

Deployment view

Figure 13: Deployment view of receipt scanner functionality in the MOTUS platform



The components are

- Mobile app: via app stores
- Web app: loaded via browser from the web server which is part of the platform backend
- Microservices: as pointed out in the generic microservice architecture, a microservice is a collection of Docker containers. The receiptscanner containers are deployed on the MOTUS Kubernetes cluster platform (except for COICOP Microservice -- see later).

API

The asynchronous interface consists of 2 rabbitmq messages:

- The microservice listens for a ProcessReceipt message with format:
  - scanner_uuid: the scanner (~ mapping of respondent)
  - receipt_uuid: identification of the receipt
  - receipt_filename: name of the receipt on the ftp server
  - timestamp of format 'Y-m-d\TH:i:sP'
  - user_selection: optional array of x,y coordinates indicating the corner points which were assigned by the respondent
- When processing is finished the microservice pushes a ReceiptProcessed message on the bus:
  - scanner_uuid: the scanner
  - receipt_uuid: identification of the receipt

The MOTUS backend can then fetch the results via the synchronous REST interface, which has the following functions:

- GET scanners: returns the list of scanners
- POST scanners: create a new scanner
- DELETE scanners: delete a scanner-uuid
- GET scanners/{scanner_uuid}: show a specific scanner-uuid
- GET scanners/{scanner_uuid}/receipts: get receipts of a specific scanner-uuid
- GET scanners/{scanner_uuid}/receipts/{receipt_uuid}: get specific receipt of specific scanner
- DELETE scanners/{scanner_uuid}/receipts/{receipt_uuid}: delete specific receipt

4.2.4. Integration: CBS @HBS platform

The OCR microservice is currently not yet integrated into the CBS HBS application, but preparations are underway for full implementation in January 2026. A successful field test has already been conducted, validating core functionalities for the user like manually adding expenses and taking picture of receipts. The OCR microservice is built as a standalone containerized API, designed to integrate seamlessly into the HBS app.

Upon deployment, the mobile app will upload scanned receipt images to temporary backend storage, after which the OCR service will be triggered. The microservice – as described earlier this chapter – will take over from there and possibly interact back with users.

Authentication for all API communication, including OCR interactions, will be handled through CBS's internal Phoenix IAM system. Mobile and web clients will authenticate via OAuth2, with tokens verified by the API Gateway before requests reach any backend service. The OCR microservice, like other services, will use secure, token-based communication with Phoenix IAM for authorization and access control.

All data processed by the OCR service—both raw and structured—will be stored securely in CBS-managed backend infrastructure, using in-house storage to comply with GDPR requirements. Integration with Phoenix's identity and access management ensures that only authorized services and users can access sensitive content throughout the pipeline.

4.2.5. Integration: INSEE evaluation microservice

INSEE is part of the consortium of the Smart Survey Integration (SSI) project aiming in advancing innovative data collection tools for household surveys, particularly for the Household Budget and Time Use Surveys.

As part of Work Package 3, a generic microservice architecture, a microservice for performing OCR and automatic classification from an image of a receipt/digital receipt has been developed, and a microservice for the collection, analysis and classification of geolocation points.

This following aims to evaluate the feasibility of integrating the microservice into the Household Budget Survey process of INSEE, from a business need perspective as well as taking into account technical considerations. Prior to this evaluation, information was received from WP3 on the generic architecture of the microservice and the example of the OCR+classification of images.

Business integration of the microservice

The microservice meets an essential need in collecting information for the Household Budget Survey (HBS) by enabling, through two steps, the conversion of a receipt photo/digital receipt into an expense coded in the survey's target nomenclature. In this section, first a description is given on how these steps will be implemented during the collection of the next survey wave in 2026, and then how the microservice could be used in 2030.

OCR and Classification process planned for HBS in 2026

The HBS application developed by INSEE from CBS's @HBS application does not perform OCR and classification from the receipt photo/digital receipt. It verifies the presence of written characters in the photo (which prevents accepting photos that are too blurry or dark) and stores the photos in a binary format in the SQL database.

A Python program generates and names JPEG files from this database. These files will be sent to a service provider, who will transcribe the data line by line. Two providers were tested in 2024: one for manual data entry by human operators and another for automated entry. For 74% of the receipts, automated entry provided a total amount and number of lines similar to manual entry. Due to its low marginal cost, automated entry is the preferred solution for data entry in 2026.

The coding of product labels into COICOP 2018 is performed by a machine learning model trained on scanner data from supermarkets. The relevance of integrating data from the 2024 test into the training corpus is currently under study. There might be (at least) two types of labels not found in scanner data:

- Receipts from other stores (clothing stores, specialized retailers, etc.)
- Handwritten labels in paper notebooks, written in a more "natural" language than checkout receipts

The microservice is not planned to be integrated into the production of the 2026 HBS because the timeline does not allow for testing and evaluating the performance of this new method compared to the existing one.

Possibility of integrating the microservice for 2030

For the 2030 HBS edition, the integration of the microservice developed under SSI could be studied. Using an external provider for expense data entry incurs a cost, and mutualizing these tools among countries is a direction in which INSEE is moving.

The microservice integration can be done in two ways:

- Maximalist Integration, with Feedback in the Application within Minutes:

The microservice can be interfaced with the application, being remotely invoked by the application and returning the OCR and classification results to the user within minutes. If the design choice for 2026, which is not to show the OCR and classification results to the respondent, is maintained in 2030, this usage is not relevant for INSEE.

- Integration of the Microservice into the Statistical Processing Chain of the Survey:

The microservice can be used in the processing chain planned for 2026, replacing the already implemented data entry and coding steps. A preliminary step would be testing the microservice on the data collected in 2026 to compare its performance with the reference chain. It would then be necessary to study how to integrate French training data into the two process steps so that the microservice can produce satisfactory results. Given the exceptionally short period between the two survey editions (2026 and 2030), these tasks will need to be carried out within a constrained timeline.

Technical integration of the microservice

If the hypothesis of testing the OCR microservice for the 2030 HBS is validated, INSEE's IT infrastructure would be capable of hosting it.

Current choices and architecture

Before presenting a target solution for integrating the microservice, it is necessary to review the current architecture implemented for the HBS data collection.

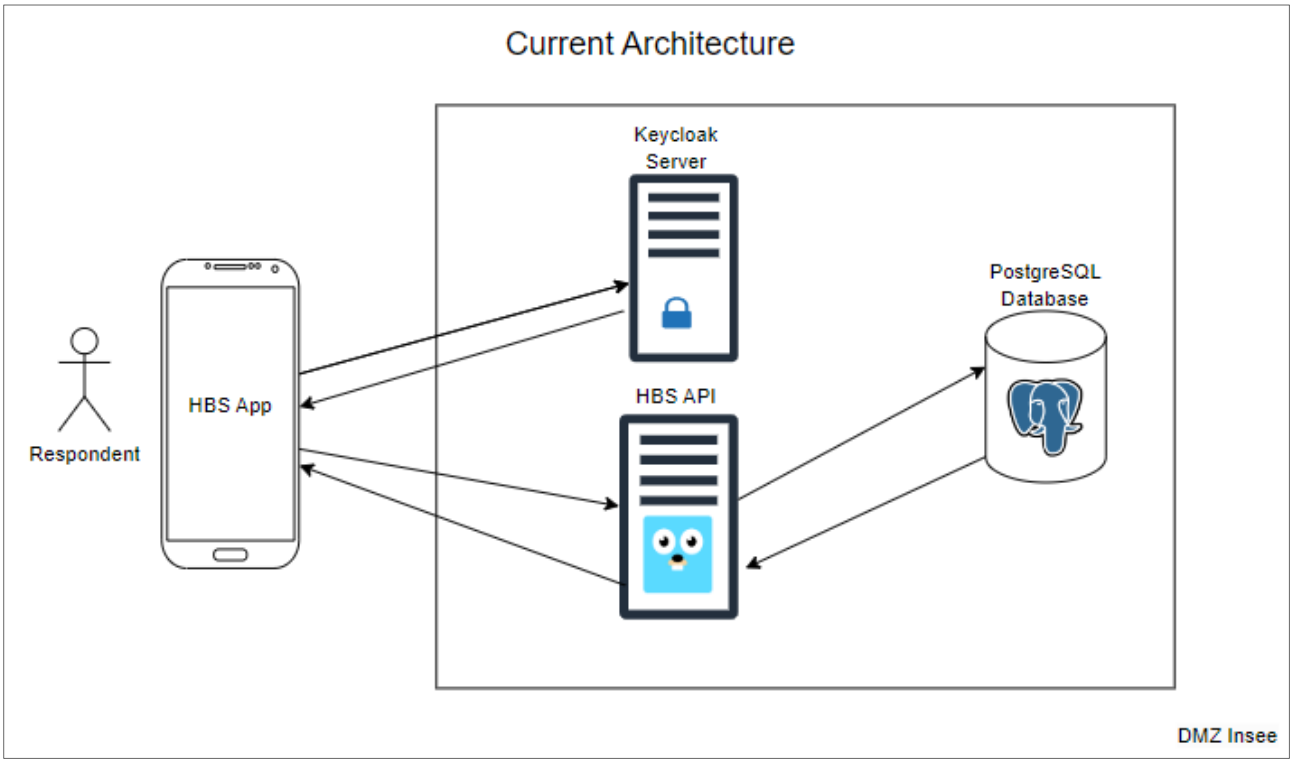
INSEE has forked the project https://gitlab.com/tabi/projects/budget. This project contains the code necessary for building native mobile applications (iOS/Android) for expense recording, the Back Office API (HBS API), and the scripts required to initialize the database.

As mentioned earlier, INSEE's choice for 2026 is to store receipt images in the database with minimal control before saving, ensuring only that the photo contains characters. The corresponding data for the receipt photos is retrieved downstream using Python scripts connected to a clone of the production database.

Beyond ergonomic work within the application, a significant portion of INSEE's development effort has been devoted to interfacing the application and the HBS API with the authentication system based on the Keycloak solution.

The current architecture is shown below.

Figure 14: INSEE current data collection architecture



The exchange dynamics between the application and the API are as follows:

- Upon login, the user retrieves a Keycloak token by logging into a page on the Keycloak authentication server. This token is then used in exchanges between the mobile application and the HBS API, where it is verified with each call.
- The HBS API handles calls from the mobile application and links data persistence in the database.

For information, the application modules are currently installed on virtual machines, but INSEE now has Kubernetes environments in open environments (DMZ).

Integration of the microservice at INSEE

Two possible integrations can be imagined on INSEE's production environments:

- "Classic" Production Infrastructures (Maximalist Integration)

INSEE now has a fairly complete Kubernetes offering, allowing the deployment of containerized microservices. This offering of development, testing, and production environments, whether in the internal zone or DMZ (for production only), enables these services to be accessible outside INSEE's network.

These infrastructures can easily deploy the OCR microservice. The constraints posed by these production environments mainly concern the quality of Docker images. A promotion pipeline is responsible for verifying this quality before making the image available in a trusted registry, thus enabling its effective deployment (searching for vulnerabilities, secrets, misconfigurations, etc.).

In the case of the maximalist integration mentioned above, the data synchronization process between the mobile application and the HBS API would need to be modified as follows:

- Sending a new image from the application to the HBS API will trigger its storage in the database and the sending of a message containing the image to be scanned. This "dual" processing could, in some cases, result in the photo not being sent to the microservice. This problem can be circumvented by slightly complexifying the architecture by adding a dedicated service to scan the database for images that were not sent to the microservice (see https://microservices.io/patterns/data/transactional-outbox.html).
- Once the receipt is processed by the microservice, the HBS API will retrieve this information to insert it into the database. It is not planned at this stage to send the information back to the respondent's mobile device. If this were to be the case, the synchronization process would likely need to be revised to send the scanned receipt data back to the respondent's device.

Below a simplified Architecture for Microservice Integration in Production is shown.

This "streamlined" processing mode would allow smoothing the load and retrieving the data immediately after collection without additional processing.

Since it is not planned to send the information back a priori, another relevant solution would be to process the images in a single batch after collection. INSEE has a "self-service" Kubernetes infrastructure that statisticians can use for this purpose.

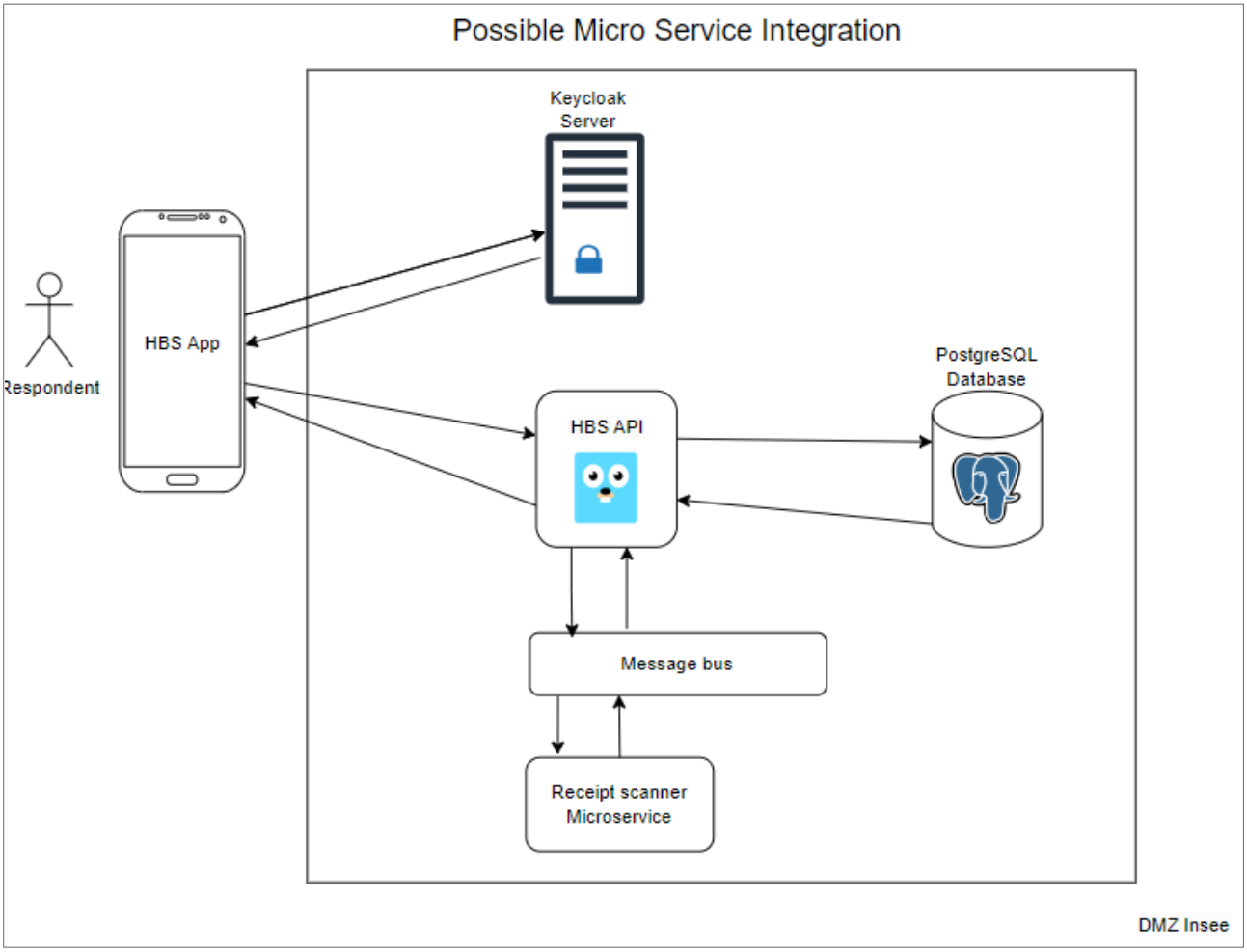
- A Self-Service Offering for Statisticians

INSEE has developed and made available to statisticians a platform called LS³ (for Libre Service "Kube"), which is the new internal data science platform at INSEE.

It allows statisticians to deploy containerized applications on demand for their specific needs. An ArgoWorkflow service is provided, enabling statisticians to schedule tasks. These platforms have the advantage of accessing production databases.

An alternative solution for integrating the microservice would be to use these tools for mass processing of receipts by the OCR microservice.

Figure 15: INSEE possible microservice integration architecture



This "streamlined" processing mode would allow smoothing the load and retrieving the data immediately after collection without additional processing.

Since it is not planned to send the information back a priori, another relevant solution would be to process the images in a single batch after collection. INSEE has a "self-service" Kubernetes infrastructure that statisticians can use for this purpose.

- A Self-Service Offering for Statisticians

INSEE has developed and made available to statisticians a platform called LS³ (for Libre Service "Kube"), which is the new internal data science platform at INSEE.

It allows statisticians to deploy containerized applications on demand for their specific needs. An ArgoWorkflow service is provided, enabling statisticians to schedule tasks. These platforms have the advantage of accessing production databases.

An alternative solution for integrating the microservice would be to use these tools for mass processing of receipts by the OCR microservice.

4.2.6. Integration: SSB evaluation microservice

WP3, represented by hbits and CBS, introduced the Generic Microservice Architecture and the OCR microservice as an example in an informational meeting with both SSB and INSEE. Documentation was provided, as well as recordings of the meeting.

Following is a brief assessment by SSB of its current setup, which includes the use of a third party OCR scanning integrated into the household budget survey, and an evaluation of the possible deployment of the within the SSI developed Receipt Scanning Microservice.

Technical architecture and considerations for OCR integration

The SSB system is already microservice-based, covering both OCR and backend services, making migration from the existing external OCR provider relatively straightforward from an architectural standpoint.

WP3 shows a Microservice architecture in which microservices can be coupled to a platform via Message Queues. Message Queues allow for asynchronous, non-blocking communication between platform and microservice, thereby increasing platform stability and efficiency.

At present, in the SSB system, direct API calls are used for sending receipts and receiving OCR results, but implementing Message Queues (Pub/Sub) could improve efficiency, reduce duplicate purchase registrations, and enhance backend stability. This approach would also facilitate smoother future migrations if needed.

Regarding classification services, a tailored model must be developed to meet specific requirements, as no universal model currently exists. Additionally, the absence of a "man-in-the-loop" classification feature within the SSI solution is a factor to consider. It remains undecided whether classification should take place in real time or post-processing, but Message Queues could be beneficial in either scenario.

With Firebase being phased out at SSB, a structured method is needed to transfer purchase data, including receipt images and OCR results, between the frontend and backend. Implementing Message Queues could provide a scalable and efficient solution for this transition.

For HBS 2026, it is too late to adapt to a different OCR microservice. However, planning for the use of Message Queues in receipt and OCR data exchange is something to follow up upon.

OCR data quality and classification

At SSB, the existing OCR service effectively extracts structured data from receipts and invoices, converting images and PDFs into text with high accuracy, provided the image quality is sufficient. It successfully categorizes key receipt components such as itemized prices, total amounts, and store names while remaining cost-effective. Additionally, the provider is responsive to feedback and willing to fine-tune models based on incorrect recognition examples.

Despite its strengths, the current OCR service has limitations, also in the case of SSB with Norwegian receipt structures. Variations in discount logic across different retailers pose challenges, some of which can be addressed through post-processing, while others require manual editing. An advanced feature of the current used OCR service is its custom classification, allowing receipts to be categorized according to COICOP standards. This functionality has not yet been tested but could be

valuable for the 2026 implementation. The OCR service also performs well in processing invoices and digital receipts.

While the SSI OCR microservice architecture shows some benefits, it still needs to show its feasibility within the Norwegian context. It would need to be seen whether the model would be able to deal with handling important nuances, especially given that it is not currently trained on Norwegian receipts. It needs to be underlined that the SSI solution makes use of a document understanding model to label different parts on the receipt. The SSI microservice can also take into account digital receipts and processes this via the same pipeline.

Besides OCR scanning and document understanding a second part of the microservice is in development and this to classify the purchases to a COICOP code. To this end WP3 developed a pipeline making use of string-matching principles and AI/ML-models.

Vulnerability and privacy

Using an external OCR provider carries the risk of service discontinuation, pricing changes, or major technological modifications.

The OCR Microservice solution is an output of the SSI Eurostat funded project and has a EUPL-v1.2 license. This SSI microservice provided a foundation that can be adapted to the Norwegian context while at the same time cooperation initiatives between users (eg. NSIs) of the solution could enhance scalability. The SSI solution is designed to run as an image in a container that can be deployed at a location of choice.

From a Data Protection Impact Assessment (DPIA) perspective, the greatest privacy concern relates to the transmission of images containing personally identifiable information, particularly invoices. The SSI setup could have an advantage here.

COICOP-classification

The COICOP classification model SSB has used has been effective for Norwegian data in 2022. This model needs to be updated and retrained with new data before it is used in 2026. Currently, we have no preference whether the classification will be performed live or post-collection. The SSI solution can provide output to the user in near real time.

During the presentation the annotation tool tick-it has been mentioned to standardise and structure the training of the models of the microservice.

Conclusion

At this moment SSB does not have important arguments to change the current OCR provider, especially since the production pipeline is already configured to handle the structure and peculiarities of this data.

Since the SSI microservice has the opportunity to train and to focus eg. on discounts it might become a viable option.

Implementing a message queue architecture could enhance system resilience and facilitate potential future migration to the SSI solution or an internally developed OCR solution.

4.2.7. Test

Via MOTUS the integration of the OCR Microservice, and so the Receipt Scanning Microservice part 1 was tested by WP2 in their small-scale tests in Belgium and Germany, and in the large-scale fieldwork carried out by the University of Mannheim.

4.3. COICOP classification microservice (Receipt Scanning Microservice – part 2)

The Receipt Scanning Microservice is supported by two microservices: the OCR Microservice (part 1) and the COICOP Microservice (part 2).

This part describes the COICOP Microservice, which has the task of assigning a 5-digit COICOP code to each product description that has been extracted from a receipt using the OCR Microservice. Hence, the COICOP Microservice is a classical text to classification task, integrated into the Receipt Scanning Microservice. The scope of the SSI project is to develop the technical possibility to integrate one or multiple different COICOP classification techniques into the Receipt Scanning Microservice.

This means that NSI-specific techniques will have to be developed by each NSI, and training is not part of the SSI. Yet, different methodological considerations will be outlined to inform data scientists and methodologists who want to deploy their own NSI specific model into the OCR Microservice. The SSI project will also provide a generic string-matching approach which can be used without advanced technical adaptations, but should be tested thoroughly.

Additionally, it is important to note that the use of the COICOP Microservice is methodologically recommended but technically not strictly necessary. The Receipt Scanning Microservice could also be used without the COICOP Microservice, hence using only the OCR Microservice. This would then result in the need to manually or automatically (but independently from the Receipt Scanning Microservice in a separate data post-processing system) classify each product description to COICOP. In essence, the Receipt Scanning Microservice would then turn into a mere data collection tool, still reducing the participation burden, but not using its full potential.

4.3.1. Design

Matching data

Regardless of the technique employed, assigning a 5-digit COICOP code to a product description requires a dataset linking product descriptions to COICOP codes, referred to here as the matching data. When a new receipt is scanned and product descriptions are identified, the task is to find the best possible COICOP match for each product description based on the matching data. Consequently, an extensive matching data corpus is a prerequisite for the COICOP Microservice.

One promising source of matching data is scanner data from price statistics, collected from retail scanners at points of sale and typically used for compiling national price indices. This data is comprehensive and systematically gathered but is often limited to specific retail sectors, such as supermarkets or drugstores. This data usually gets automatically classified to COICOP and therefore serves as a directly usable matching data source.

If such data is unavailable or inaccessible, an alternative approach is to gather matching data manually, for example, by collecting receipts directly from consumers or retailers. Once the data is collected, a crucial step is coding all extracted product descriptions to their corresponding COICOP

codes before using the data as matching data. While this approach can provide a representative dataset of the most consumed goods, capturing all products in this manner is likely impractical.

Lastly, manually curated tag lists created and maintained by subject-matter experts can serve as a matching data source. While such lists may not cover the full range of products found in dynamic receipt data, they often include the most essential products.

The coverage of the matching data is a critical factor on two levels. First, if the data only includes products within a specific COICOP range—such as food and near-food products from supermarkets or items from drugstores—the COICOP Microservice should be restricted to those ranges. Users should also be informed that receipts from other sectors, such as hardware or home improvement stores, cannot be automatically classified to COICOP. It is up to app developers to integrate an automatic detection whether a given receipt is eligible for the COICOP Microservice or not. Second, once a specific retail sector is included, it is crucial to ensure that coverage within this sector is sufficiently comprehensive. One way to address these challenges is by combining multiple matching data sources, which can significantly enhance both coverage and comprehensiveness.

Another related challenge is the timeliness of the data. Maintaining and regularly updating the matching data is essential, as new products are frequently introduced. Systems with regular updates, such as those typically used in price statistics, provide significant advantages in ensuring the data remains reliable and relevant. Additionally, it should be ensured that frequently unclassified product descriptions are added to the matching data along with their corresponding COICOP codes. This requires either an automated data base monitoring or manual quality control measures.

A key requirement for the matching data is that product descriptions should closely resemble, or ideally match, the product descriptions extracted from receipts. For example, price statistics scanner data from certain stores may closely align with receipt product descriptions, while in other cases, significant differences may exist. The accuracy of the COICOP Microservice will largely depend on the extent of this overlap.

The work described below was developed and tested using price statistics data from Germany and the Netherlands as the matching data source. The variables extracted from this data include the product description, the assigned COICOP code, the product id, and the name of the retail store, all of which were supermarkets.

Matching techniques

To successfully find the best matching COICOP for an OCR-extracted product description from a scanned receipt, three distinct techniques were identified in this project: (1) automatic string matching, (2) machine learning, and (3) manual string searching.

Automatic string matching (1) involves directly comparing receipt extracted product descriptions to the matching data, offering simplicity and efficiency. Machine learning (2), on the other hand, uses models trained on the matching data, i.e., the product description and the corresponding COICOP. As a result, the model learns to recognize more generic patterns and structures with which it can classify previously unseen product descriptions to a certain degree. The downside is the significant initial effort required for training a model as well as the need for substantial computational resources. Finally, manual string searching (3) involves the respondents in the classification process.

In this solution, a receipt extracted product description is entered into a search algorithm and the respondent selects the best-fitting COICOP category. This method is the most burdensome for the participant but can be highly precise since it leverages the respondent's expertise on their own private purchases and can be applied to purchase domains where no matching data is at hand. Also, the respondent could change the search string if the product description itself is not diagnostic enough.

If a given NSI has the matching data and technical resources, we recommend using all three techniques in sequence. Additionally, as a final resort it should be considered to accept unclassified data which is then classified during data post-processing. Therefore, the idealized process can be divided into four main steps within the COICOP Microservice, where each step is initiated in sequence and only if the previous one does not yield a satisfactory result for a given product description. Note that the criteria for such thresholds are country-specific and require calibration.

- Step 1a: If specific matching data for a given store is available, attempt automatic string matching on the matching data of this selected store.
- Step 1b: Attempt automatic string matching on the full matching data.
- Step 2: Prediction made from machine learning model trained on the full matching data to find the highest scoring COICOP match.[3]
- Step 3: Allow the user to manually search within the full matching data, with the option to alter the search string.
- Step 4: COICOP classification by the NSI during data post-processing.

Figure 14 provides an overview of the complete COICOP classification pipeline. Steps 1 and 2 were developed within the SSI project. Steps 3 and 4 depend on the functionalities provided by the app which is used as well as on data processing routines employed by NSIs. The next section will describe the string-matching and machine learning techniques in more detail.

Step 1: String-matching

Step 1 of the COICOP Microservice involves automatic string-matching to assign the most appropriate COICOP code to a product description extracted from a receipt. This step consists of two main pipelines: store-specific (1a) string-matching and (1b) generic string-matching.

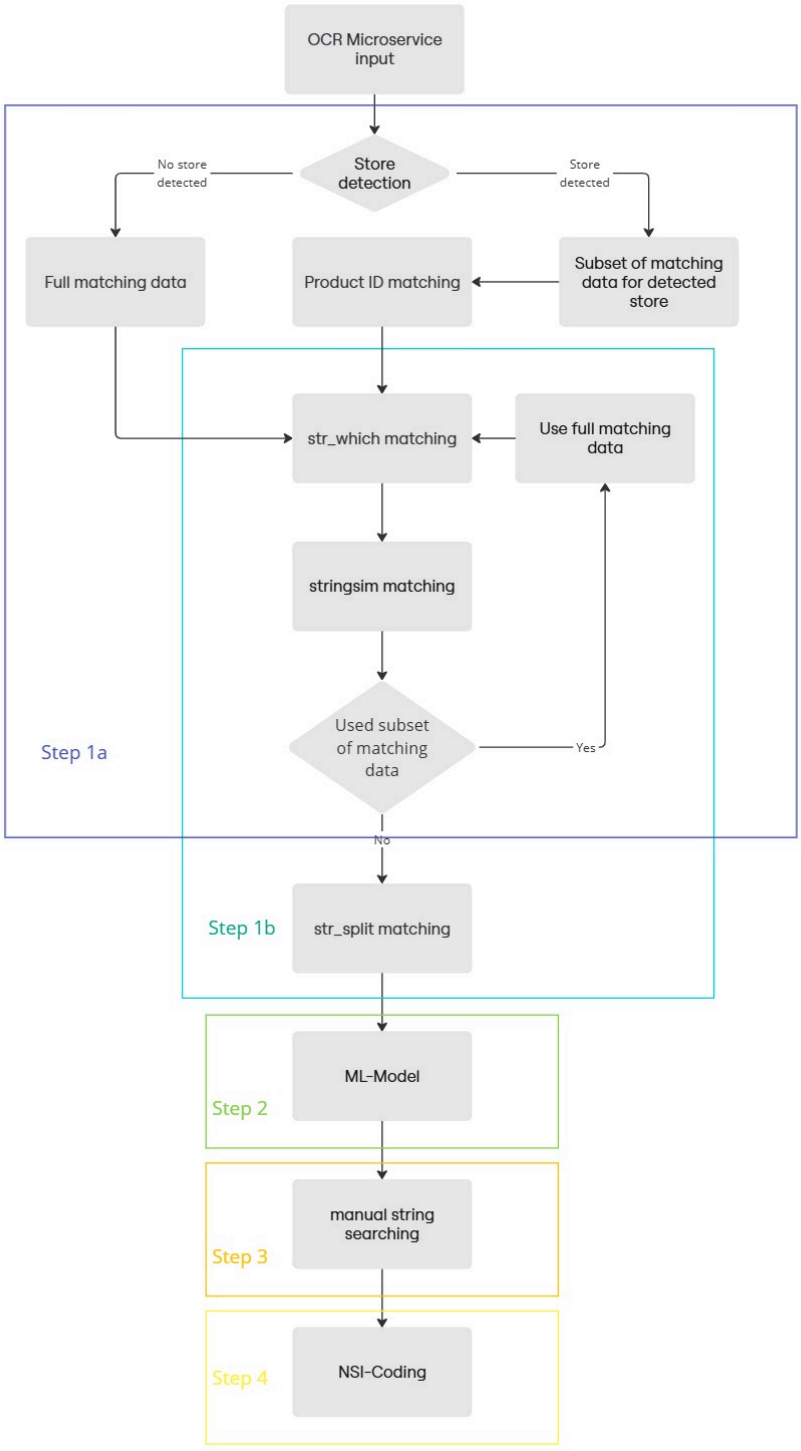
Store-specific matching (1a): If a retail store can be identified on the scanned receipt which is also provided in the matching data, the matching is first performed on a store-filtered subset of the matching data. This store-filtered matching ensures that the product descriptions are matched with the most relevant dataset for the given store. If no satisfactory match is found within the shop-filtered corpus, the process returns to a broader step and performs the matching on the complete matching data (see generic matching).

Generic matching (1b): If no store-specific match is found, no store is detected on the receipt or the detected store is not found in the matching data, the process bypasses the store-specific pipeline and directly matches the product description with the complete matching data.

[3] There would also be the option to split Step 2 following the same logic as for Steps 1a and 1b. Hence, first store-specific models would be used, followed by a generic model. This option was discussed, but not systematically tested or implemented within the project.

Both of these main pipelines run through various sub-steps. These sub-steps are described next. Note that the first sub-step (product ID matching) is only used in the store-specific pipeline for a selected subset of retail stores. The str_which and stringsim sub-steps are used in both pipelines. The last approach (str_split) is only used as a final sub-step in the generic string-matching pipeline. Also see Figure 16.

Figure 16: Process model of the COICOP Microservice



Product ID matching

Scanner data deliveries from retail stores usually contain some type of product ID. In most cases this information is irrelevant, however, selected retail stores print the product ID on their receipts. One such example is the supermarket chain Aldi in Germany. Therefore, if the possibility is given, the most straightforward way of matching product descriptions is by means of such a product ID. This method is highly accurate and relies on the correct extraction of product IDs from the receipt.

str_which matching

In most cases the matching process will rely on actual product description comparisons between the receipt extracted text and the matching data. The str_which represents an exact 1:1 match and therefore checks whether a receipt strings is found in the matching data.

Example process

If the receipt string is "apple spritzer," the str_which function identifies potential matches such as:

- "apple spritzer,"
- "apple spritzer 6x1.5l," or
- "apple spritzer bottles 6x0.5l."

In case multiple potential matching candidates are identified, the process evaluates the string similarity between the receipt string and all potential matches by using a fuzzy-string matching algorithm. For instance, in the case of "apple spritzer," an exact match (e.g., "apple spritzer") is identified as the best match. If no match is found, the next sub-step is initiated.

stringsim matching

The stringsim function calculates a similarity score between the receipt string and all strings in the matching data that are between 30% less or 30% more string length. This limits the number of computations to the most plausible comparisons and increases performance. The similarity score then indicates how closely a product description matches entries in the corpus based on the minimum number of operations (insertions, deletions, or substitutions) needed to align one string with another. The algorithm used here is called optimal string alignment.

Example process

Each product description in the matching data is scored on a scale from 0 to 1, with higher scores indicating a closer match to the receipt string. Strings with a similarity score above a predefined threshold (e.g., 0.7) are considered potential matches. If different potential matches are available, as earlier fuzzy matching is used to fine the best potential match.

str_split matching

The str_split function introduces an additional preprocessing step before performing string matching. The receipt extracted product description is tokenized at the word level and split into substrings based on delimiters such as spaces, commas, periods, or dashes. Before this step is applied, special characters are removed and substrings with a length of 2 characters or fewer are excluded, as they are often non-informative or may introduce noise. The idea of this step is to find partial string matches which are close enough to be deemed equivalent.

Example process

The receipt string "rücker Käse Natur" is split into the following substrings:

- "rücker,"
- "Käse," and
- "Natur"

Starting with the first substring ("rücker"), the matching data is filtered using str_which. Only strings containing "rücker" are retained. The process is repeated iteratively for each subsequent substring ("Käse" and then "Natur"), progressively narrowing the search corpus. After applying str_split, a vector of potential matches is generated. From this vector, fuzzy matching is used to identify the best possible match and assigned if a certain threshold is reached.

Summary

Step 1 uses a combination of direct string-matching, similarity-based matching, and substring tokenization to identify potential COICOP matches. The integration of fuzzy matching ensures that even when exact matches are unavailable, a well-fitting COICOP code is identified based on string similarities. By prioritizing shop-specific matching data when available, the process enhances both efficiency and accuracy. If no match can be identified, the machine learning step 2 is initiated.

Step 2: Machine learning

When the string-matching techniques in Step 1 do not yield a satisfactory COICOP classification, we turn to machine learning as the next step. Machine learning offers the advantage of handling textual data with greater flexibility, allowing for the classification of product descriptions that may not exactly match entries in the existing matching data. Unlike rule-based approaches, machine learning models learn patterns from data, enabling them to generalize to a certain extent beyond the exact words seen during training. This is particularly useful when dealing with variations in wording, spelling errors, abbreviations, or newly introduced products.

A text information to COICOP coding task represents a supervised learning task where the input is a product description and the output is a predicted COICOP code. The success of this approach heavily depends on the quality and quantity of labelled training data (i.e., the matching data), as well as the choice of modelling and pre-processing techniques. In our project, we explored various approaches, but two distinct approaches will be described here as potential candidates. Note that the performance and accuracy of these approaches will depend heavily on country-specific characteristics of the data and should be evaluated individually.

A machine learning model will always generate a prediction, even if the model is uncertain about its classification. Therefore, some caution should be exercised when using these outputs. One way to

mitigate incorrect classifications is by considering the confidence score of a prediction. Only predictions with a confidence score above a predefined threshold should be automatically accepted. If the score falls below this threshold, the product description should remain unclassified (i.e., as a "no match"). It would then get passed on to Step 3, which is not a focus of this project (see Figure 16).

N-gram based preprocessing with logic regression and random forest

This approach begins by transforming product descriptions into a series of string character chains. In our case, we used three- and four-character n-grams. For example, the product description "apple juice" would be broken down into n-grams like "app," "ppl," "ple," "le," "jui," and so on. N-grams can make the model more resilient to minor variations or spelling errors. Once the textual data is transformed into n-gram representations, we explored two classification modelling techniques, which yielded similar results in terms of accuracy: Logistic regression and random forest.

FastText

FastText represents a popular approach to text classification tasks. Unlike the more traditional n-gram approach described above, FastText learns word embeddings, which are dense vector representations capturing semantic meanings. Additionally, FastText incorporates subword information, allowing the model to understand and generalize across different word forms and spelling variations. For instance, the words "yogurt" and "yoghurt" would be recognized as similar due to shared subword components. FastText excels in scenarios where the textual data is rich in linguistic nuances or domain-specific terminology. It can quickly train over large datasets, making it highly efficient for large-scale classification tasks. Moreover, its ability to perform well with limited computational resources makes it an attractive choice for (near) real-time applications. Accordingly, the accuracy of the FastText model was quite similar to the n-gram based models, but the processing time and efficiency was much better.

Summary

Both approaches offer unique benefits. N-gram-based models are straightforward to implement and interpret, but can take up much more computational resources than FastText. The choice between these techniques depends on factors such as data availability, computational constraints, and accuracy with NSI-specific data. Destatis has adopted a FastText model for integration.

4.3.2. Implementation

The code can be found under the following two links:

https://github.com/essnet-ssi/cbs_coicop

https://github.com/essnet-ssi/destatis_coicop

4.3.3. Integration

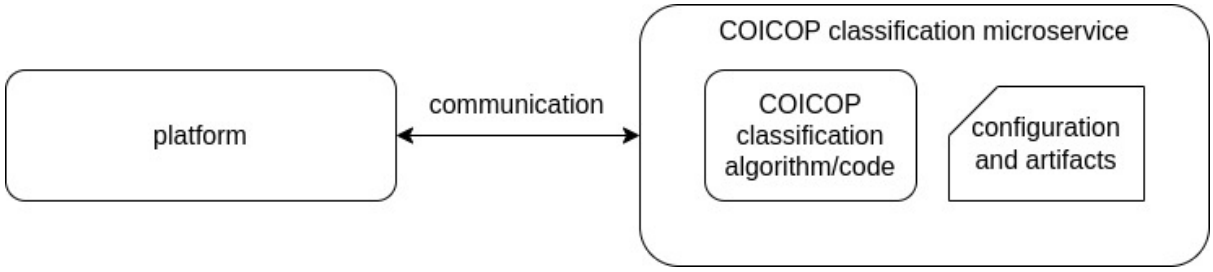
The integration of COICIOP classification in a platform involves:

- the implementation of the algorithm/code in a microservice e.g. docker container,

- providing support in the microservice for configuration and artifacts e.g. algorithm parameters, models...
- enabling communication between microservice and platform,
- the execution of the COICOP classification algorithm in the microservice when a request has been received, and

Figure 17 gives a high-level overview of the different aspects.

Figure 17: Microservice integration



Since this microservice is domain-specific, it is - in the context of this project - not integrated in a platform. It will be integrated in a later phase.

4.3.4. Test

Two different approaches were undertaken to test the two COICOP classification techniques. CBS conducted tests where the scanner data used as matching data was divided into two separate sub-datasets. The first one was used as actual matching data, while the second was used as test data to validate the accuracy of the developed techniques and run comparison benchmark tests. Since no actual receipt data was used, this test approach can be called an *internal validity test*. Conceptually this test assumes that product descriptions on receipts are identical to production description in scanner data records. Given that deviations between these two data sources are likely, the accuracy scores should be interpreted with caution when it comes to the absolute levels.

In contrast, Destatis collected a sample of receipts from actual supermarkets and manually classified product descriptions to COICOP. This data was then used with the different COICOP classification techniques to validate and benchmark their accuracy and performance. These tests mimic the actual production use case and can be termed *external validity tests*. However, they come with the limitation that the sample size was not very large and test results should be interpreted with caution as well.

Internal validity tests – CBS

Test strategy

CBS used scanner data of three supermarkets: Lidl, Albert Heijn (AH), Plus. The data lists all their products offered in each month, allowing us to track products that were added and removed. The volume of data varies per supermarkets: 7 million records for Plus, 3 million records for AH, and 400,000 records for Lidl.

Using this scanner data, we test how well an ML approach classifies product descriptions. These tests are centred around two research questions:

(1) How well do ML models classify product descriptions into COICOP-classifications, and does it change over time?
(2) How well do ML models classify product descriptions from supermarkets not encountered during training?

The first question addresses how quickly ML models becomes outdated. This is relevant given that supermarkets constantly add and remove products from their changing catalogue. This question gives insight into how well models handle new products, how quickly they degrade over time, and how frequently they need to be retrained.

The second question addresses whether ML models can handle supermarkets absent from the training set. It answers the scope of its use in the real-world. If ML models perform well on unseen supermarkets, then we may use it for other supermarkets without collecting its data. If not, then its use is limited to the supermarkets in the training set. Classifying a new supermarket will also require new training data.

To translate the questions into concrete experiments, we split the data set in different ways. Each way of splitting simulates the relevant conditions for the ML model in the real-world.

For question (1) we split the data by some date. The products offered before this split date will be used as matching data, or as training data in ML terms. The products after this split date will be used to test the models, where we form separate validation sets for each month. Indeed, we validate the trained models on multiple sets rather than one, allowing us to track the performance of ML models over time.

In our validations we chose June 2023 as the split date. All products before this date form the matching data, and the products in the months June 2023, July 2023, and August 2023 form separate validations sets. With this setup, we therefore validate the ML models on three consecutive months. Table 1 shows an overview of the sets, along with the number of instances.

To translate question (2) into experiments, we make data splits for each supermarket. In each split, we leave out one supermarket for validation and use the remaining supermarkets as matching data. By leaving out one supermarket, we validate how well ML models deal with supermarkets absent from the matching data. Table 2 shows an overview of the splits along with the number of instances.

We emphasize one key difference in the splits for questions (1) and (2), which is how the supermarkets are divided between the matching data and the validation data. In the time split for question (1), we test ML models on instances only from supermarkets seen earlier in the matching data; the validation data will not feature unseen supermarkets. In the supermarket splits for question (2), however, we test ML models precisely on supermarkets not encountered during training. The splits therefore test the ML models under different assumptions.

Table 3: Validation split by time

Split nr.	Months of products in matching data	Month of products in validation set
1	< June 2023 (n=445,468)	June 2023 (n=66,860)
2		July 2023 (n=88,741)
3		August 2023 (n=74,248)

Table 4: Validation split by supermarket

Split nr.	Stores in matching data	Supermarket in validation set
1	Lidl, Plus (n=476,946)	AH (n=26,198)
2	AH, Plus (n=477,425)	Lidl (n=25,719)
3	AH, Lidl (n=51,917)	Plus (n=451,227)

Tested ML algorithms and feature extractors

This section outlines ML algorithms and the feature extractor tested. These are the two ML components needed to classify product descriptions. Since ML models only accept numerical input, and cannot classify the product descriptions directly, feature extractors first transforms them into a numerical representation. From this, an ML algorithm can then produce a classification model, a process also known as training.

Since there are many options for feature extractors, we have conducted preliminary experiments for finding the most suitable feature extractor for product descriptions. The results showed that the tf-idf[4] vectorizer with character-level n-grams performed best, most likely because these texts are short and riddled with abbreviations. All ML algorithms in the following will be tested with character-level tf-idf.

The ML algorithms tested, including some string-matching methods for comparison, are listed in Table 5. It also lists the string-matching pipeline by Destatis, described earlier in Figure 16. We have included two versions of the pipeline. This is because the full version has a subroutine that matches products by article ID, giving it an unfair advantage over the other algorithms. Another version was therefore added to remove this unwanted advantage.

We note that some ML algorithms were trained on a simple random sample of 100,000 product descriptions rather than the full matching data set. These are marked with an asterisk in Table 5. All other algorithms are trained on the full matching data. A random sample is used because these algorithms were designed to handle only 100,000 instances in the sci-kit learn implementation. Exceeding this number may lead to computational or memory problem.

[4] Tf-idf stands for Term Frequency-Inverse Document Frequency and is a numerical statistic used for text mining and information retrieval to evaluate how important a word is to a document in a collection (or corpus).

Table 5: The tested (ML) algorithms

Algorithms
Logistic Regression*
Logistic Regression SGD (Stochastic Gradient Descent)
Naïve Bayes (Multinomial)*
Random Forest Classifier *
Multilayer Perceptron Classifier*
Constant Predictor
Exact String-matching Algorithm
FastTextClassifier
DeStatis String-Matching Pipeline
DeStatis String-Matching Pipeline (w/o article-ID matching)

Validation split by time

Table 6 shows the accuracy scores of the ML algorithms on the validation sets of three months in 2023: June, July, and August. The last column shows the averages and standard deviations over all three months. Similarly, Table 5 outlines the scores calculated for each supermarket's subset of products. Intuitively, the score shows the ratio of correctly classified product descriptions in the validation set. For example, an accuracy score of 89,9 means that 89,9% of all product descriptions in June 2023 were classified the correct COICOP-label.

In terms of raw accuracy score, Logistic Regression (Stochastic Gradient-descent variant) scored best, averaging 89,4% accuracy. With Random Forest and the Destatis' Pipelines scoring similarly. Most algorithms, however, scored high, indicating that the validation sets are easy, owing to the large overlap of products across sets.

All algorithms performed worse over the months. Although the rates of decline vary, they all range between 0,5% and 1%. The worst monthly performance drop, however, is seen in exact string-matching with 1,5%.

Compared to the pipeline by Destatis, the ML algorithms performed similarly. Meaning that there is no significant difference between string matching techniques and ML in terms of how quickly models degrade.

Table 6: Accuracy scores for the validation split by time

Algorithm	Jun 2023	Jul 2023	Aug 2023	Mean tests
Logistic Regression (SGD)	89,9	89,4	88,8	89,4 ± 0.5
DeStatis Pipeline	89,9	88,8	88,2	89,0 ± 0,7
Random Forest Classifier**	89,9	89,2	88,6	89,2 ± 0.7
DeStatis Pipeline (w/o art. ID matching)	88,3	87,4	86,7	87,5 ± 0,7
Exact String-Matching Algorithm	84,6	82,7	81,5	82,9 ± 1.5
FastTextClassifier	82,2	82,1	80,4	81,6 ± 0.4
Multilayer Perceptron Classifier*	81,4	81,7	80,8	81,3 ± 0.4
Logistic Regression*	77,7	78,0	77,4	77,6 ± 0.5
Naïve Bayes (Multinomial)*	77,4	77,6	76,8	77,3 ± 0.4
Constant Predictor	9,9	10,5	9,9	10,1 ± 0.3

Table 7: Accuracy scores for the validation split by time, calculated for each supermarket

	AH			Lidl			Plus		
	Jun	Jul	Aug	Jun	Jul	Aug	Jun	Jul	Aug
Logistic Regression (SGD)	88	87	87	91	91	89	91	90	90
DeStatis Pipeline	93	92	92	93	92	89	88	87	87
DeStatis Pipeline (w/o art. ID matching)	88	87	87	92	91	88	88	87	87
Random Forest	87	87	86	91	90	89	91	90	89
Exact String-Matching	88	87	86	88	84	79	83	81	80
FastTextClassifier	69	70	71	79	82	80	81	84	82
Multilayer Perceptron	70	69	69	78	77	76	86	86	86
Logistic Regression	63	63	63	73	73	72	84	83	83
Naïve Bayes (Multinomial)	66	66	66	75	75	74	82	81	81
Constant Predictor	7	6	6	7	8	8	12	12	12

Validation split by supermarket

Table 8 shows the accuracy scores on the validation splits by supermarket. Again, the last column is per algorithm the average over all three splits.

All algorithms performed poorly in these tests. Even the best performing algorithm had an average accuracy of only 48,1%, meaning that less than half of the product descriptions in the validation sets were correctly classified. This suggest that the tested algorithms classify poorly on unseen supermarkets.

In this test, however, ML algorithms outperform the string-matching algorithms. Although most ML algorithms tested ranged from 30% to 40% accuracy, DESTATIS' string-matching pipeline scored on average only 13,3% accuracy. Therefore, when classifying unseen supermarkets, the ML approach is better than the string-matching approach.

Table 8: Accuracy scores for the validation split by supermarket

Algorithm	Split 1: Test AH (n=26.198)	Split 2: Test Lidl (n=25.719)	Split 3: Test Plus (n=455.227)	Mean tests
Logistic Regression (SGD)	49,6	45,3	49,5	48,1 ± 2,0
Multilayer Perceptron Classifier*	43,7	45,4	45,0	44,7 ± 0,7
Random Forest Classifier**	46,7	44,9	41,8	44,5 ± 2,0
Naïve Bayes (Multinomial)*	47,3	45,3	43,4	45,3 ± 1,6
Logistic Regression*	42,0	45,1	44,7	43,9 ± 1,4
FastTextClassifier	43,2	33,7	39,0	38,7 ± 3,9
DeStatis pipeline	35,9	10,0	10,9	18,9 ± 12,0
DeStatis pipeline (w/o art. ID matching)	21,6	10,0	8,2	13,3 ± 5,9
Constant Predictor	6,4	6,9	13,1	8,8 ± 3,8
Exact String-Matching	4,7	2,9	0,7	2,8 ± 1.6

Summary

On the classification of product descriptions of ML algorithms, we tested two aspects.

We first tested the base performances of the ML algorithms on previously seen supermarkets and the change over time. The tests show that the best model scored on the first month 89,9% accuracy, with a monthly accuracy drop of 0,5%. Similar performances, however, were tested for Destatis string-matching pipeline, meaning that there is no significant difference between the ML approach and the string-matching approach.

The second aspect tested is how well ML algorithms deal with supermarkets absent in the matching data. The results showed that all algorithms performed poorly, with the best ML model scoring on average only 48,1% accuracy. Although ML algorithms performed better than Destatis string-matching pipelines (13,3%), they are still poor at classifying unseen supermarkets.

In conclusion, ML models classify product descriptions well, but only for supermarkets on which they are trained. They classify poorly on supermarkets absent from the matching data set. All models perform worse over time due to changes of product catalogues.

External validity tests – Destatis

Test strategy

Destatis collected 326 receipts originally intended for training of the OCR Microservice algorithms. These receipts were processed through an OCR extraction and document understanding algorithm, with resulting text elements manually annotated and existing OCR errors corrected. This allowed the data to also serve as test data for the COICOP Microservice. As part of this process, all product

descriptions were assigned to a COICOP category, resulting in 2,274 product descriptions matched with a corresponding COICOP code from five different supermarket chains. Of the 2,274 product descriptions, a subset of 725 was from supermarkets where the store could not be identified or was unknown.

This test data consisted of the original OCR product description extraction, the manually corrected product description, a COICOP code, and a store name. This provided the opportunity to test the COICOP Microservice with both corrected and uncorrected OCR strings.

This differentiation is crucial because OCR extractions often involve errors. These errors can result from wrinkles, poor image quality, or other external factors. However, even with perfect images, OCR is prone to mistakes, such as confusing similar characters like "O" and "0" or "I" (capital i) and "l" (lowercase L), due to uniform fonts and tight spacing. While the idea of developing an algorithm to correct common OCR errors was discussed several times during the project, it ultimately wasn't pursued due to time constraints.

The conducted tests focused on evaluating COICOP classification accuracy and runtime performance. Two distinct test strategies were applied. First, the test data was processed sequentially through classification steps 1, step 2, as well as in combination of the two, mirroring the path-dependent workflow of production. This ensured that a product description advanced to the next step or sub-step only if it had not already been matched to a COICOP category. These results are presented here. Second, benchmark tests were conducted by submitting the entire test dataset to each step and sub-step independently. These benchmark tests aimed to uncover runtime bottlenecks and accuracy outliers, though the results on sub-step level are not included in this report. NSIs are advised to test each step and sub-step with their own data.

Corrected OCR product descriptions

The tests presented here distinguish between three classes of results: correct classifications, incorrect classifications, and no match cases. Correct and incorrect classifications refer to instances where a processing step made a prediction that was either accurate or false. The no match class applies to cases where a COICOP code was not assigned due to high uncertainty. In the production pipeline, any no match case is passed on to the next step for further processing.

Figure 18 illustrates the proportions of correct, incorrect, and unmatched product descriptions across different COICOP classification approaches. To allow a direct comparison, both the string-matching and FastText methods were tested on the full set of 2,274 product descriptions. The string-matching approach correctly classified 71% of the product descriptions, whereas the FastText model achieved 64% accuracy. However, the FastText model was configured with a conservative confidence threshold to minimize false positives (incorrect matches). As a result, while the proportion of incorrect classifications remained similar between the two approaches, FastText had a higher proportion of unclassified product descriptions.

When evaluating the two approaches in sequence—where only the unmatched product descriptions from the string-matching step are passed to FastText—the combined results show an increase in correct classifications to 77%. This demonstrates that the two-step approach enhances the COICOP Microservice's performance. The overall error rate remains 19%, while 5% of product descriptions

remain unclassified. This finding highlights the added value of the machine learning component, as it successfully captures cases that the string-matching approach alone could not classify.

Figure 18: Accuracy by COICOP classification step

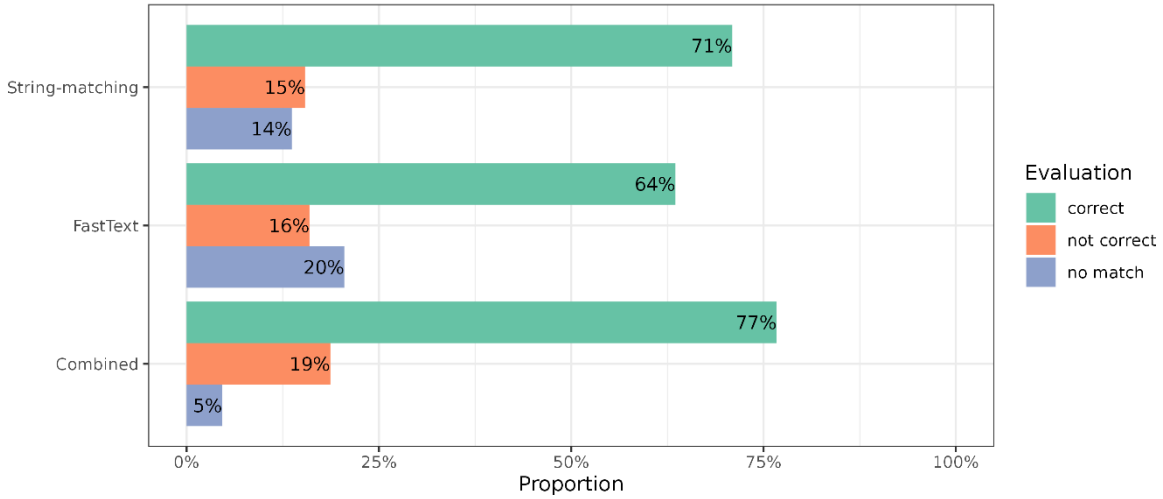
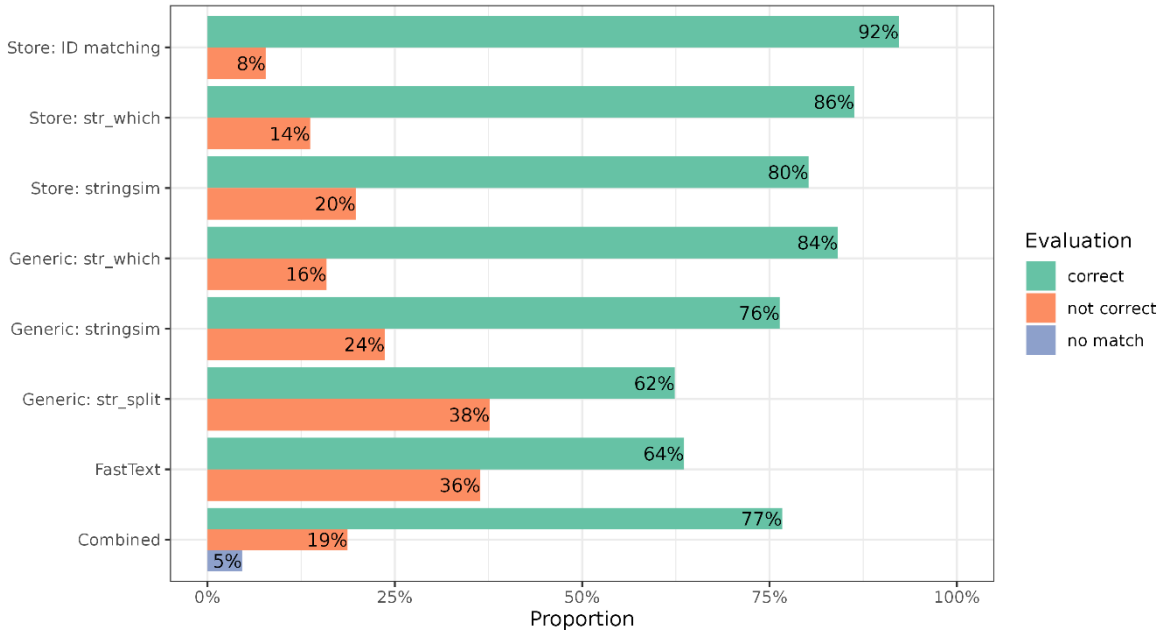


Figure 19 breaks down the sequential string-matching sub-steps, showing how classification accuracy evolves at each stage. Within the string-matching pipeline, the proportion of correct classifications starts at 92% and gradually declines to 62% as product descriptions progress through the sub-steps until a COICOP code is assigned. This trend is expected, as more challenging product descriptions—those not classified in earlier steps—are passed on to subsequent sub-steps. Interestingly, the only increase in accuracy occurs when transitioning from the store-specific to the generic matching pipeline. This suggests that searching the full matching dataset is beneficial if store-specific matching fails. A possible explanation is that certain newly introduced products in a given supermarket may have already appeared in another retail chain, making them identifiable through a broader dataset.

Figure 19: Accuracy by COICOP classification sub-step

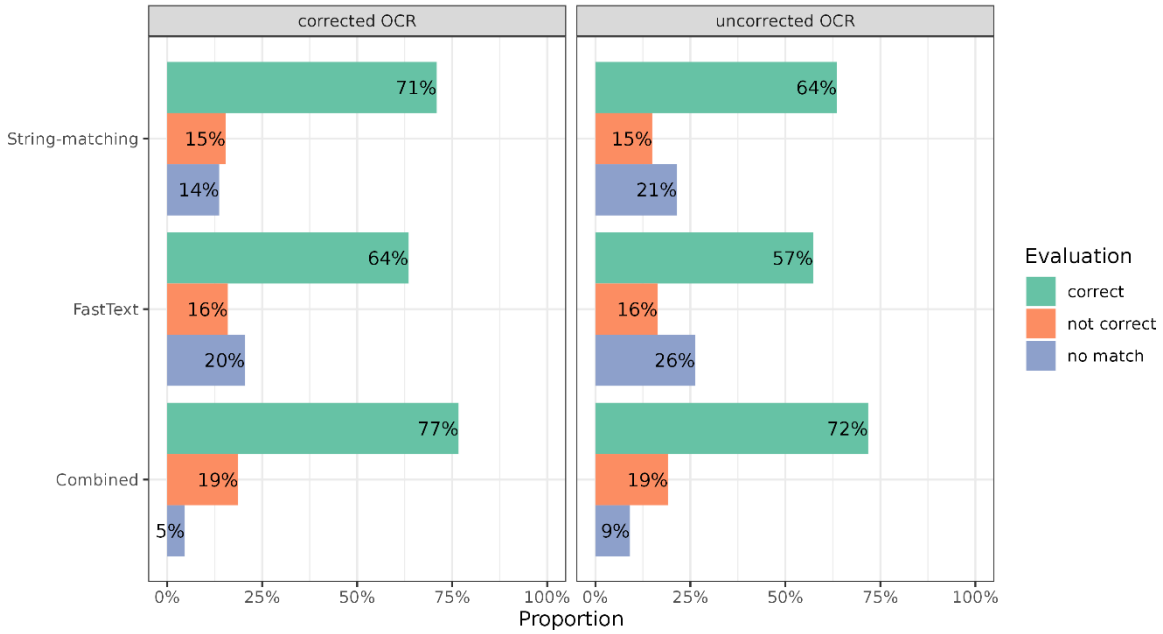


Uncorrected OCR product descriptions

OCR extractions often introduce errors, which can impact classification accuracy. To assess this effect, we conducted the same tests as before, this time using uncorrected raw product descriptions. Figure 20 compares accuracy across COICOP classification steps, differentiating between corrected and uncorrected OCR extractions.

As expected, uncorrected OCR product descriptions resulted in a lower proportion of correct COICOP classifications compared to corrected inputs. Specifically, the accuracy of the full COICOP Microservice pipeline in sequence decreased from 77% to 72% with uncorrected data. Interestingly, this decline in correct classifications did not lead to an increase in incorrect classifications. Instead, it resulted in a higher number of product descriptions remaining unmatched to a COICOP code. It is unlikely that this result generalizes, but in this case it would result in increased manual efforts (in step 3 or step 4) rather than increased error rates.

Figure 20: Accuracy comparison by COICOP classification step and OCR quality

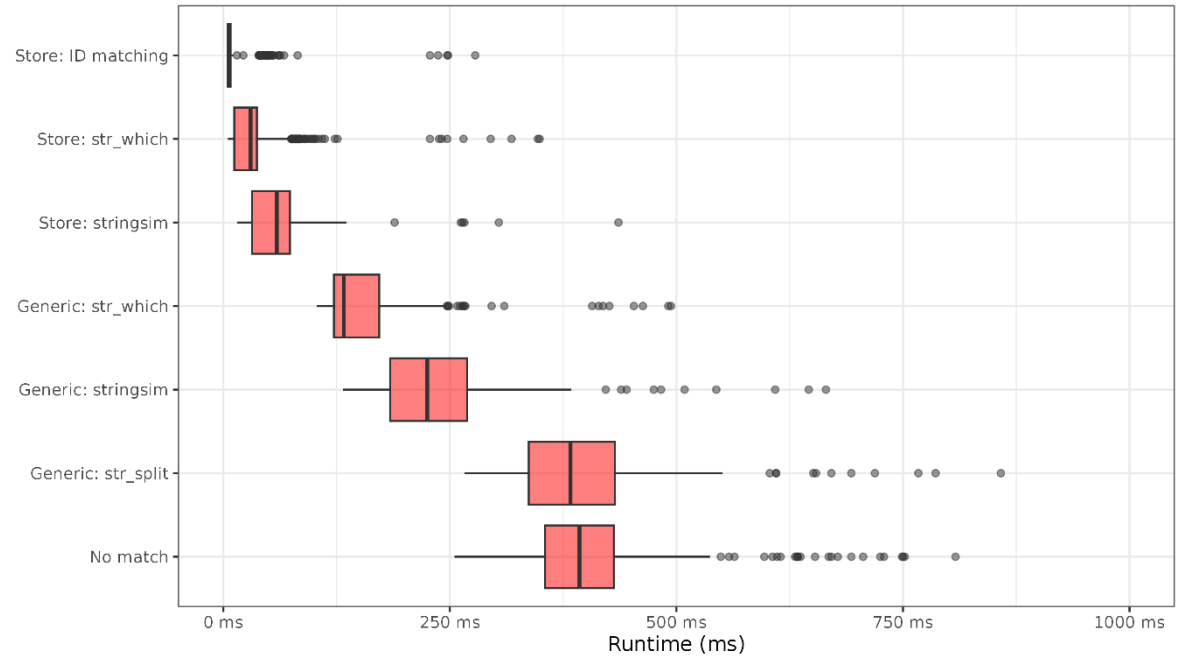


Processing time

Figure 21 illustrates the processing time for each string-matching sub-step in milliseconds (ms). As single data rows progress sequentially through these sub-steps until a COICOP code is assigned, the measured processing time is cumulative across the sub-steps. The boxplots display the median processing time. Store-specific sub-steps are highly efficient, with average processing times of less than 60 ms per product description. In contrast, the more generic sub-steps, which rely on the full matching dataset, significantly increase processing time, with median values ranging between 130 and 380 ms. This is because these steps handle a larger dataset and process more challenging product descriptions that were not classified earlier. Additionally, later sub-steps that allow similarity-based rather than exact matches take longer to compute. If no match is found in the string-matching pipeline, the product description is passed to Step 2 (machine learning) after an average processing time of 390 ms.

The code was implemented in R running on a large server. The current implementation is characterized by a research and development focus. Processing times could likely be improved by refining processing steps, using other R packages, or transitioning to a different programming language better suited for high-performance computing.

Figure 21: Processing time by string matching sub-step



Summary

The results show that chaining the various string-matching sub-steps with the machine learning approach adds predictive power, making sequential use highly recommended. Comparisons between uncorrected and corrected OCR product descriptions revealed differences, but these were less severe than initially anticipated in the project. Nonetheless, investing in algorithms to correct OCR-extracted product descriptions would be worthwhile.

Calibrating thresholds at each sub-step seems critical, as error rates depend heavily on how conservatively each step is applied. These values have a large influence on the number of incorrect COICOP classifications in relation to those predictions where no assignment was made (no matching result). Depending on the resources of a given NSI for quality control and manual post-processing, the decision to set these thresholds more conservatively or not will vary. Since these thresholds will vary by country, universal recommendations cannot be provided. However, it is vital for NSIs to dedicate time and resources to this calibration process. The results above should be regarded as a proof of concept.

5. GeoService Microservice

The GeoService Microservice is the second environment being developed in this project. The same principle was followed as with the Receipt Scanning Microservice. First the functional and non-functional requirements (5.1) are described, followed by a description of its two main parts: the non-domain specific part, and the domain specific part. The non-domain part holds two developments: the development of the algorithm to define stop-track clusters (5.2) and the development of the transport mode prediction model (5.3). The domain part describes how the HETUS classification can be matched with the stop clusters (5.4). For each part, the design, implementation, integration, and testing are discussed in detail.

As a more general-purpose microservice, the GeoService Microservice is documented from a technical perspective. In contrast, the HETUS classification microservice is presented from a practical standpoint based on the work of ISTAT.

The GeoService Microservice has been successfully integrated into the MOTUS platform. Testing has been performed through the MOTUS platform in collaboration with WP2. The microservice was not integrated in other platforms.

5.1. Functional and non-functional requirements of the GeoService Microservice.

Just like in chapter 4, chapter 5 has the focus on better involving and engaging households and citizens by defining and operationalizing a new/modified end-to-end data collection process. This time the focus is on making use of geolocation points to support the Time Use Survey.

To collect these geolocation points the use of internal sensors of smart devices is needed. NSIs and linked organizations have worked on platforms to allow households to register their time spending in an online diary. The past few years a multitude of applications were developed to collect time use data, and those related to the ESS have developed these applications in light of the HETUS guidelines. In the SSI project the CBS, hbits, Insee and SSB represent this focus on official (time use) statistics.

The general idea is to provide to the users/respondents a framework of places (stops) and travels (tracks) and the mode of transportation in order to support them in keeping their timeline up-to-date. Within WP3 the GeoService Microservice is developed as middle part software that processes the sensory data in order to provide tentative input to the timeline of the diary.

The main functionality is Stop-Track prediction, Transport Mode prediction, and the connection of tracks to transport motivations following the HETUS guidelines to gain TUS relevant information.

5.1.1. Business requirements

TUS gathers information on the daily activities' household members perform. Typical to a TUS is that these activities are being collected with their temporal, spatial and social context. TUS is harmonized via the HETUS-guidelines with the first edition being published in 2000, and recently received its third update with the 2018-guidelines. Member States have the option to collect TUS data. Currently the third data collection is running. Methodological variations between countries apply. In 2030 TUS will enter the IESS agreement on an optional level.

Just like HBS, TUS is a household study. In TUS, all eligible household members are invited to keep a 2-day diary, for the same two days to be able to study the intra-household allocation of time. Following the HETUS-guidelines, one diary day contains 24 hours running from 4 am until 4 am the next day. Each activity is reported verbatim, both for the main and (possible) parallel activity. The same counts for information on the location or the mode of transport. The use of an electronic device is answered with a tick-box (yes when checked). The social environment is also captured with through tick-boxes collecting information on whether the activity was done alone or together with someone known. A distinction is made between social partners within (partner, parent, child up to 17 years old, other household member) and outside the household. A new episode starts when either an activity and/or one of the contexts change. Every diary day is ended with a small questionnaire asking about the level of satisfaction during the reported day.

Under the wings of the process of modernization, and also under the auspices of EUROSTAT, TUS underwent a mode shift to an online data collection strategy making use of web and mobile supported applications to collect time use data. Initiatives were taken by various Member States and are inventoried on the EUROSTATS' wiki page: https://webgate.ec.europa.eu/fpfis/wikis/display/ISTLCS/TUS+TOOLS+MENU.

Taking all developments into account, one of the main thresholds for TUS comes from the detailed reporting of activities in a the time-space framework. An important indicator to picture this threshold is the time between the actual action and the reporting of the activity. Studies show that the quality of reporting remains good upon a reporting delay of at maximum 24 hours (see Yesterday reporting). It is however expected that the burden to reconstruct the day turns higher the longer the actual activity has been performed. Depending the detail of the activity (i.e. more activities on the detailed level) an extra impact expected.

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 TUS through the development of a geolocation microservice that through sensor activation captures geolocation points to derive information on the trip, mode of transport and the stops. Related to the stops, extra context can be added through the connection of third-party databases, and a classification algorithm would be able to link a HETUS code activity (or list of activities) to the stop.

A successful realization of the development and implementation will not entirely reduce the active participation of household members in the registration of their daily activities and context, 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.

This project will focus on the smartphone as (1) the device to install the mobile application on, and used by the user as interface to partake to the study, as well as (2) being the motion tracker to collect the movements of the respondent/user, as a proxy of the person itself.

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 development of WP3)
- Objective 2: To develop a geolocation microservice to predict trips and stops
- Objective 3: To implement classification solutions (machine learning, string matching, or search algorithm based) to classify stop to the HETUS-classification
- 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).

TUS study

In this section TUS studies are being described as they provide the context in which the geolocation microservice operates.

In TUS studies questionnaires and a time 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 TUS 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 TUS study is the registration of activities in a diary. All eligible household members keep a diary for the same 2 days, one weekday and one weekend day.

TUS diary

The diary collects at the minimum episode information, where an episode is defined by a beginning and ending time and a change of:

- A main activity as defined in the HETUS Activity Classification List (ACL)
- (If any) a secondary activity as defined in the HETUS Activity Classification List (ACL)
- The place of the activity or a mode of transport when moving
- The use of an electronic device, and
- The social context

The registration of the products and services is linked to the HETUS Activity Classification List (ACL), and is demanded to be delivered on 3 digits. NSIs often use more digits to aggregate to a higher level. The HETUS guidelines further describes the other contexts: place 2 digits, electronic device 1 digit, and social context 1 digit.

In addition, extra context questions can be added to the online diary, an example is asking about the motivation for performing an activity.

Every diary day collects extra information through a small questionnaire, it relates to:

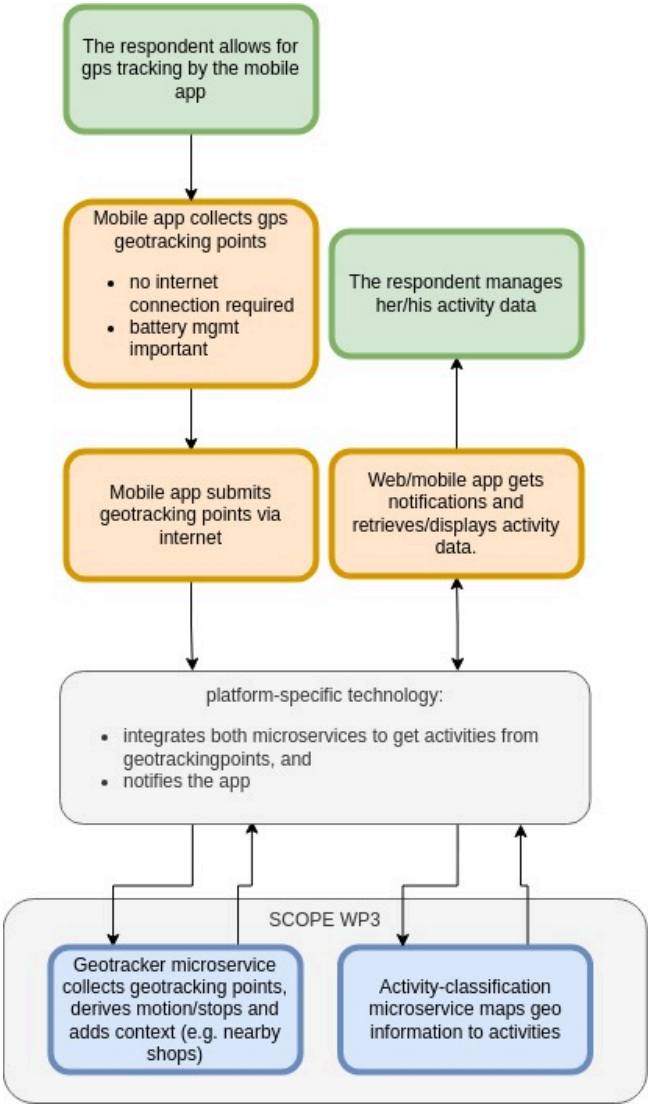
- When the diary was completed
- What the most pleasant activity was
- What the most unpleasant activity was
- What the most stressful activity was
- The overall appreciation of the day
- Whether the day was ordinary or unusual
- Whether a trip within the country or abroad was made, and how far the trip was

5.1.2. Functional requirements

Figure 22 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 switch on the sensors to track the movement of the smartphone. After processing by the microservices, the respondent can view/edit/manage her or his activities.
- functionality related to the *app* is indicated by the yellow boxes. The app is responsible for GPS tracking, display of information and communication with the platform core.
- functionality related to the *microservices* is indicated by the blue boxes. The function is to derive essential information on trips, mode of transport, stops, context of stops and activity classification. The blue boxes are in scope of WP3.

Figure 22: Geotracking functional requirements – flow diagram



User handling

Respondent handling (green boxes)	
REQ R1	The respondent allows for GPS tracking by the mobile app
	Technically, the mobile phone OS needs permission to enable GPS tracking for the mobile app.
	Formally, user consent is required.
REQ R2	The respondent manages her/his activity data
	A possibility (~MOTUS) is to present the activity list to the respondent as tentative data. The respondent is then able to edit this list before it becomes final.

Microservice

Microservice (blue boxes)	
REQ G1	Geotracker microservice collects geotracking points
	Geotracker (regularly) receives (new) tracking points from the platform.
	An internal database stores all tracking points
REQ G2	Geotracker microservice derives motion/stop
	An algorithm processes the tracking points in order to find a timeline of motions (transport) and stops.
	The algorithm could use external sources such as openstreetmap in order to improve the results.
	Support for user-specific locations (home, work etc.) is required as well.
REQ G3	Geotracker microservice adds context to motion/stops
	External sources can be consulted to add context. E.g. for stops, a list of nearby places/shops could be added.
REQ A1	Activity microservice assigns scores to POIs
	A score (POI-score) is assigned to each POI inside an adaptive radius around the stop centre location, based on the weighted median of the distances calculated between each POI and all GPS points of the stop, weighting by the accuracy of GPS points.
	A short list of POIs is identified using the elbow criterion on the POI scores. *Categories of place are assigned to each POI*
REQ A2	Activity microservice associates activities to categories of places
	Through a Bayesian decomposition, for each POI of the short list, the conditional probability of HETUS activities are calculated starting from the distribution observed in TUS data. The variables considered (duration and time of the day, HETUS place category, occupational status, age classes) in the decomposition are linked with the corresponding variables observed in the stop and for the specific respondent.
	A rank of the HETUS activities is assigned to the stop, based on a final score calculated aggregating the probabilities of the activity weighted by the POI-score associated with the activity for each POI in the shortlist.

5.1.3. Non-functional requirements

Non-functional requirements	
REQ N1	A microservice should be independent from any specific platform.
	A microservice has no dependency to other environments, and has an independent operation.
REQ N2	It must be possible to connect and communicate with a microservice from any platform.
	A microservice receives input, and provides output making use of APIs.
REQ N3	A 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).
	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.

5.2. Geolocation microservice and stop-track clusters (GeoService Microservice – part 1a)

The GeoService Microservice is supported by two microservices: the Geolocation Microservice (part 1) and the HETUS classification Microservice (part 2).

The design of the Geolocation Microservice holds two important elements: the definition of the stop-track clusters by trajectory segmentation and the prediction of the travel mode upon the track clusters. These elements will be discussed in 2 sections but finally result in one microservice, the Geolocation Microservice. This section discusses the algorithm to derive the stop and track clusters.

5.2.1. Design

After the Geolocation Microservice gets the geolocation points, the stop-track part starts with the stop detection algorithm which takes into account 4 steps:

- Filter GPS points based on accuracy,
- Determine which GPS points are significant stop points
- Cluster the stop points, and
- Post-processing
  - reduce number of clusters (merging)
  - guarantee stops and tracks alternately

The required input parameters are timestamp, longitude, latitude and accuracy. The list of geolocation points needs to be time-ordered.

In an extra step, also, extra information can be attached to the stop clusters by connecting to a POI or places API:

- Point-of-interest: find places inside or nearby stop clusters (e.g. OpenStreetMap, Google Places)

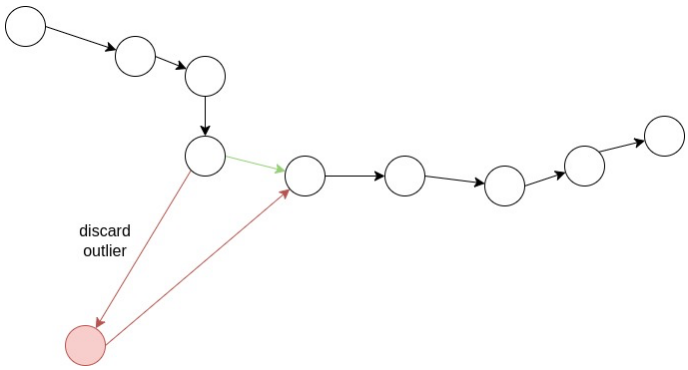
The output of the algorithm is a list of alternating stop and track clusters. Stop clusters will have extra information regarding nearby places (e.g. shops).

Pre-processing

Filter GPS points

Filter GPS point which has good enough accuracy. Currently, the algorithm requires an accuracy of 100m, but this value is configurable.

Figure 23: Eliminating outliers by filtering GPS tracking points according to their accuracy

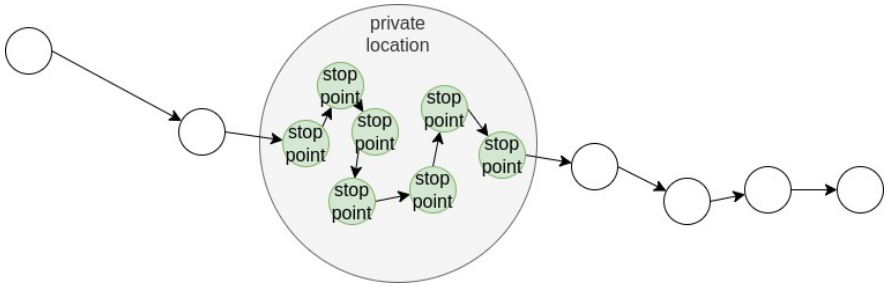


Determine significant stops and clusters

Private locations

When a GPS point falls in a private location (home, work etc.) then it is always regarded as a stop. A private location is defined as a circle (lon, lat, radius).

Figure 24: Private locations are considered as stops



ATS to determine significant stops

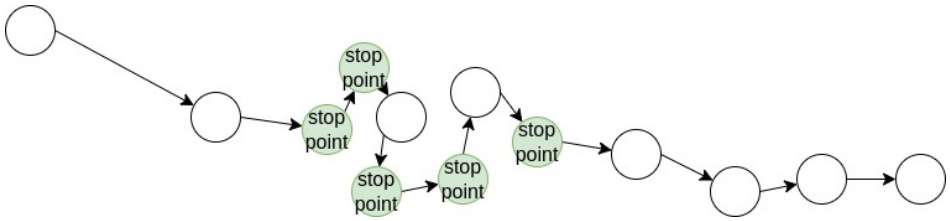
The algorithm for stop detection is implemented from the paper “Individual and collective stop-based adaptive trajectory segmentation” from Agnese Bonavita, Riccardo Guidotti and Mirco Nanni, as published in Geoinformation (2022) 26:451-477. From this paper, only the individual stop-based adaptive trajectory segmentation (ATS) has been implemented.

Essentially, the algorithm decides that a GPS point is a stop point when more than t seconds is spent between the current GPS point and the next GPS point that is more than x meters away.

The algorithm can be tuned by changing the temporal and/or spatial parameters. By default, the implementation used 50m as spatial parameter (as advised in the paper) and 180s for the temporal parameter.

Note that deriving the temporal parameter from the GPS data by means of a Thompson tau statistic is also supported by the algorithm. The description can be found in the paper.

Figure 25: Identification of stop points

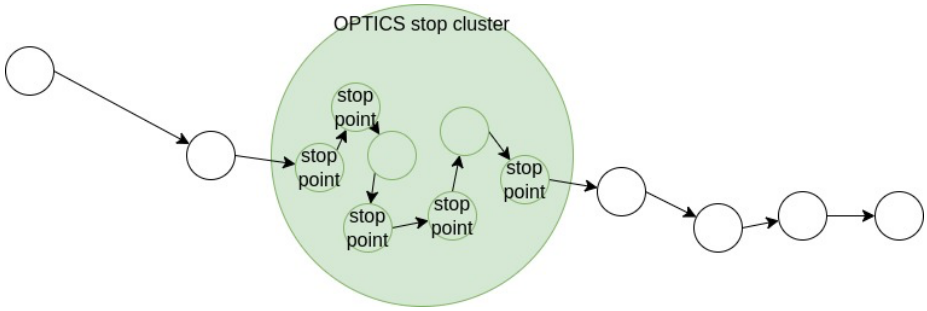


Cluster stop points

Once the stop points have been determined, the next step is to cluster the stop point in order to get stop clusters. In order to achieve this, the project decided to use the well-known OPTICS algorithm (derived from DBSCAN), which applies a density-based technique on spatial data (https://en.wikipedia.org/wiki/OPTICS_algorithm).

Because OPTICS does not take into account the time aspect of the data, the results are further processed to split the spatial clusters based on time as well.

Figure 26: Clustering of stop points



Post-processing

Finally, in order to deliver a clean output, merging of stop clusters is done after which track clusters are added between the stops

Point-Of-Interest (POI)

The geolocation microservice produces a set of stops and tracks. To utilize this output as input for the TUS activity microservice, it's beneficial to add points of interest (POIs) to the stop clusters.

Several services exist which map a coordinate and a set of properties to a list of POIs. Commercial online services include Google Places and Mapbox. A well-known non-commercial, self-hosted solution is Openstreetmap.

hbits decided to opt for a self-hosted Openstreetmap service which exposes the Overpass API (https://wiki.openstreetmap.org/wiki/Overpass_API).

The MOTUS platform sends a request to the service with the following parameters: latitude, longitude, radius and a list of OSM tags. Latitude, longitude and radius correspond to the stop cluster centre location and radius. The OSM tags include aerialway=station, aeroway="aerodrome|heliport|terminal", amenity, leisure, shop, sport and tourism, but the list of requested tags is configurable.

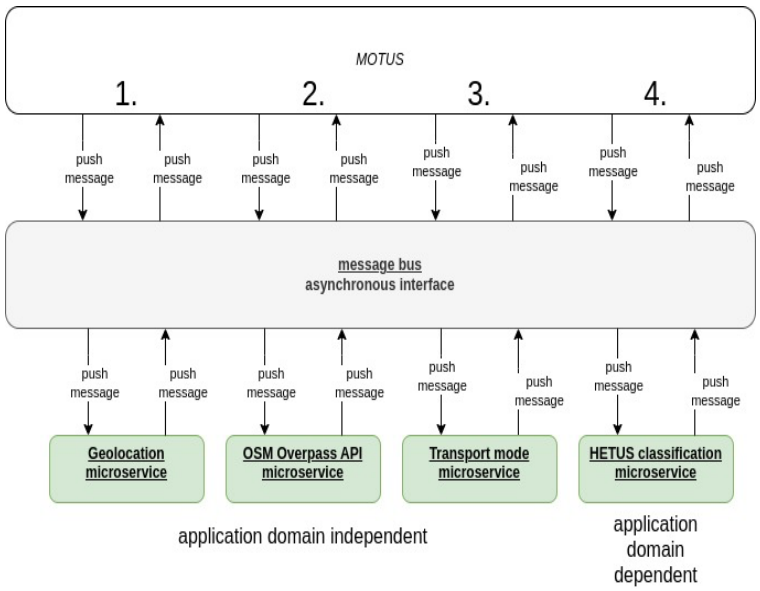
5.2.2. Implementation

An R implementation with bindings to C++ is made and available in Github repository. Depending on the platform, the python runtime can be deployed as a (containerized) microservice in various ways. In the following section, the MOTUS data collection platform is used to show the integration of the development.

5.2.3. MOTUS integration

The integration of geo in MOTUS takes place by sequentially requesting data from the microservices, beginning with the geolocation microservice. This is followed by retrieving Points of Interest (POIs) from the OSM Overpass service and deriving the transport mode from the tracks. Finally, all the gathered information is compiled and - once the activity microservice integrated - sent to the TUS activity microservice. The final output of this process is a collection of stops and tracks. The stops have additional POIs information. If the activity microservice is present, MOTUS will create and propose tentative activities for the respondents based on geo information.

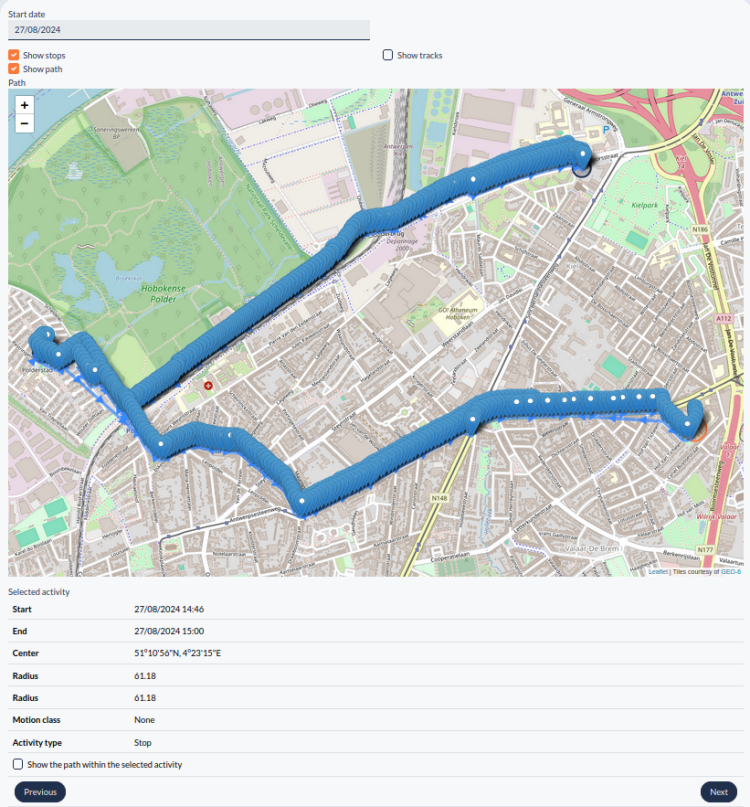
Figure 27: Microservices integration in relation to MOTUS



The MOTUS integration also includes visualization and handling in the back office and mobile app (front office will follow in the future).

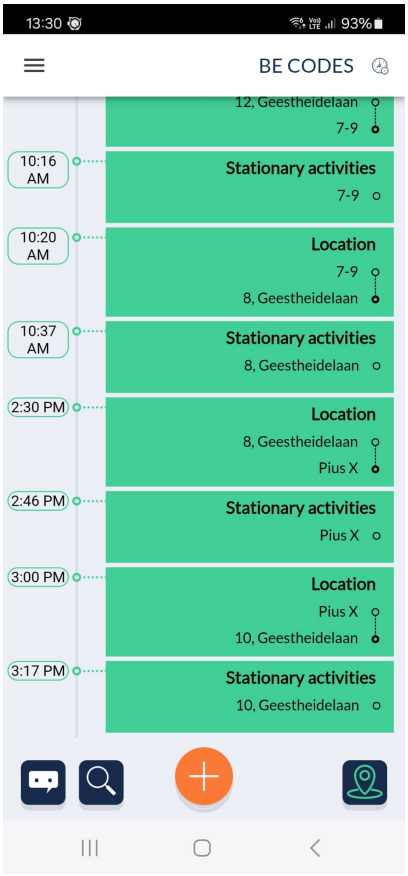
In the back office, a day-based view of tracking points and stops can be retrieved from diary in the respondent's flow. An example is shown in the screenshot. The markers are tracking points while the circles indicate stops.

Figure 28: Visualisation of tracking point in the MOTUS back-office



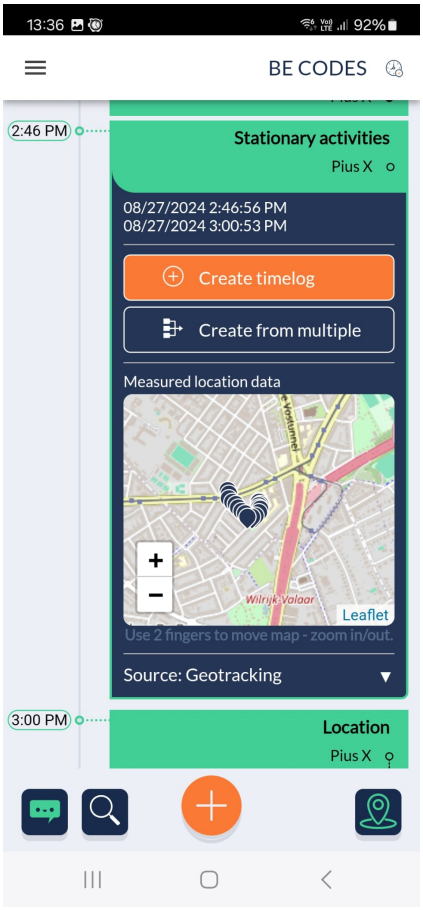
In the respondent's mobile app, geo data is visualized as an itinerary of tentative data, alternating stops and tracks. This itinerary helps the respondent to recall his/her activities.

Figure 29: Tentative overview of stop and tracks in the MOTUS application



A respondent can create a timelog in her/his diary based on the information available in an entry of the itinerary. Examples of pre-filled information could be start/end time and suggested activity. The intent of the pre-filled fields is to help the respondent fill in the timelog. Each field can be changed by the respondent.

Figure 30: Detailed information per tentative cluster the MOTUS application



5.2.4. Test

Via MOTUS the integration of the Geolocation microservice, and so the GeoService Microservice part 1 was tested by WP2 in their small-scale tests in Belgium, Germany and Italy.

5.3. Geolocation microservice and the mode of transport (GeoService Microservice – part 1b)

The GeoService Microservice is supported by two microservices: the Geolocation Microservice (part 1) and the HETUS classification Microservice (part 2).

As being stated earlier, the design of the Geolocation Microservice holds two important elements: the definition of the stop-track clusters and the prediction of the travel mode upon the track clusters. This section has a focus on the detection of the mode of transport, which together with the stop-track model becomes part of one microservice.

An integral component of smart time-use, travel, and mobility surveys is the ability to predict respondents' modes of transportation, thereby minimizing the necessity for manual data labelling and reducing the response burden. This section documents the development of a transport mode prediction algorithm specifically designed for integration with smart surveys and discusses the detection of the mode of transport upon the track clusters.

5.3.1. Design

After classifying the geolocations into stop and track clusters, the track clusters are plugged into the transport mode prediction algorithm. The algorithm is based on a decision tree, using smartphone GPS data and infrastructure information from OpenStreetMap (OSM).

Before going into detail about the algorithm development, the underlying data is explained. There are two sets: the development data and the open geo-data.

Development data

The development of the algorithm is based on data collected by Statistics Netherlands (Schouten et al. 2024). Later on, the developed algorithm will be evaluated on open geo-data (test data) (see section "Open geo-data").

The development dataset is based on a Dutch general population sample collected from 2022 to 2023. Data from 255 participants were used for the development. The dataset contains 4.298 tracks and a total of about 20 million observations. An observation consists of a timestamp and geo-location (longitude and latitude). We refer to Schouten et al. 2024 for general details about the dataset.

Data processing

For the specific development of the transport mode classification algorithm, the following data processing steps were applied:

- tracks exceeding 10 hours were excluded,
- tracks containing fewer than ten GPS observations were removed,
- labels for similar transport modes were grouped: 'car (driver)' and 'car (passenger)' were merged into a single transport mode `car'. The categories 'bike' and 'e-bike' were also merged into one transport mode `bike'.

After preprocessing, the average number of GPS observations per track was 843, although the median was notably lower at 402, indicating a skewed distribution. Similarly, the average track duration was 53 minutes, but the median was 12 minutes, reflecting the skewness. Regarding track length, the mean was 15 km, while the median was considerably shorter at 2.8 km, again indicating a skewed distribution in the data.

Transport modes

The target variable of the classification task is the transport mode used during a track. The distribution of track labels across transport modes in the entire dataset is presented in Table 9. The developed algorithm will only classify a single mode. Multi-modal tracks cannot be classified. The target variable also contains the label 'Other'.

Train and test splits

The dataset was partitioned at the user level, ensuring that each user was assigned exclusively to the training or testing set.

The number of users in the dataset is limited, and some users contribute disproportionately with a large number of tracks. As a result, fully random training-test splits may become imbalanced across labels, potentially affecting the robustness and generalizability of the algorithm. Therefore, it was

decided to split the dataset by partitioning users into separate subsets for training (70%) and testing (30%), ensuring that no user appeared in both sets. Stratification was applied based on each user's dominant mode of transport to maintain a balanced representation of transport modes.

The public transport modes bus, metro, and tram were grouped for the train and test splits (they were used as individual classes for the remainder of the development). This practical solution prevented the case that tram was once only the most prominent mode, and therefore, no split could have been applied because the stratification would require at least two occurrences of a transport mode. This approach preserved the variation and distribution of transport modes across both subsets, ensuring that the test set accurately reflected the training set's characteristics while preventing user overlap between the two sets. Table 10 and Table 11 show the training and test splits.

Table 9: Distribution of transport modes in development data. Rows ordered by count.

Mode	Count	Percentage
Car	1.892	44,02
Walk	1.002	23,31
Bike	946	22,01
Train	161	3,75
Other	111	2,58
Bus	96	2,23
Metro	58	1,35
Tram	32	0,74
Total	4.298	100

Open geo-data

This dataset was reserved exclusively for testing the developed algorithm, with no portion used during the development or training phases, ensuring an unbiased evaluation of the algorithm's generalization capabilities.

The dataset was collected in the summer of 2024 to obtain data with high-quality labels without errors for the transport mode. This data was collected by a small group of CBS staff and staff from the University of Utrecht.

Table 10: Trained set. Rows order by count.

Mode	Count	Percentage
Car	1.333	44,43
Walk	684	22,80
Bike	643	21,43
Other	103	3,43
Train	99	3,30
Bus	75	2,50
Metro	45	1,50
Tram	18	0,60
Total	3.000	100

Table 11: Test set. Rows ordered by count.

Mode	Count	Percentage
Car	559	43,07
Walk	318	24,50
Bike	303	23,34
Train	62	4,78
Bus	21	1,62
Tram	14	1,08
Metro	13	1,00
Other	8	0,62
Total	1.298	100

Furthermore, the data contains tracks within the Netherlands and Germany. Accordingly, this test set will inform how well the algorithm generalizes to a different app version, which compared to the app used to collect the development data has a revised sensor configuration, and data collected in a different country. Data from 5 users with 137 tracks are available. The transport mode distribution is shown in Table 12.

Table 12: Transport modes in open geo-data. Rows ordered by count.

Mode	Count	Percentage
Walk	78	56,93
Tram	27	19,71
Bike	10	7,30
Train	9	6,57
Bus	5	3,65
Metro	5	3,65
Ferry	3	2,19
Total	137	100

Note that the decision tree was not trained on data containing the 'ferry' label. Thus, the algorithm will fail to predict this label. However, it was collected to evaluate the algorithm's decision for this label. Lastly, the most prominent mode in the development data, 'car', is not included in the open geo-data. This dataset is publicly available in the SSI Git repository (https://github.com/essnet-ssi/geoservice-ssi).

Methods

In what follows different steps are described which are related to the construction of GPS features, the construction of OSM feature, and the pre-processing of GPS and OSM features.

Later on, the development of the decision tree, and the transport mode prediction algorithm is handled.

Feature construction GPS

Several GPS features were tested during the development of the decision tree:

- speed (speed, acceleration, jerk, snap),
- GPS (accuracy, frequency),
- direction (bearing, altitude),
- trip (length, duration), and
- time (weekday, weekend indicator).

For most features, several variants were created based on different statistics. A list of all evaluated features is given in the Annex 2 of this report.

Feature construction OSM

The features are based on publicly available OpenStreetMap (OSM) data obtained from the official Geofabrik download portal (https://download.geofabrik.de/). A documentation of all OSM infrastructure contained in the database can be found at: https://wiki.openstreetmap.org/wiki/Map_features.

OSM data complements the GPS features by providing details on infrastructure such as road networks, transit routes, and stations. Integrating this data improves usually the quality of the transport mode classifications. Fourie (2025) found that OSM did not help:

- to improve the classification quality for the transport modes, walk, bike, and car, and
- as OSM provides a variety of data on transportation and travel infrastructure that does not improve classification performance (roundabouts, traffic junctions, stop signs, speed cameras, and streetlamps)

Accordingly, in the development of the algorithm, only OSM features about bus, metro, train, and tram stops and routes were created. The required Python code can be found in the accompanying script osm features.py on Github.

Track buffering

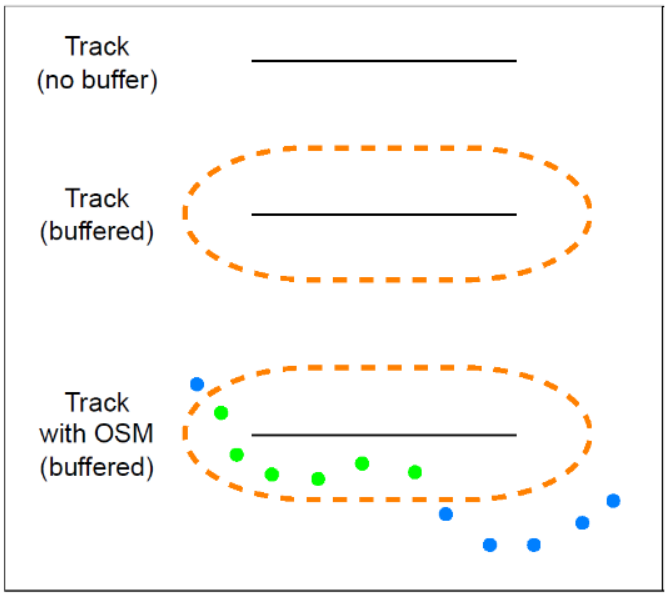
Buffering a GPS track when calculating features using OSM data is beneficial because it helps include relevant spatial context around the track, improving feature extraction and accuracy.

This step is helpful because:

- First, it accounts for GPS inaccuracies and noise.
- Second, it captures nearby infrastructure and context.
- Third, it enables more robust feature engineering.

The buffering process takes the GPS coordinates representing the track and generates a buffer zone around it. This buffer is defined by a specified radius or distance, which determines how far the area extends from the track's centreline. For instance, a buffer with a radius of 25 meters would create a region 25 meters wide on either side of the track. This is the radius so that the diameter will be 50m. An example of this procedure is shown Figure 31.

Figure 31: Simplified example of track buffering: a single track (black solid line), a buffered track (black solid line with surrounding orange dashed line), and a buffered track with mapped OSM infrastructure



Once the buffer is constructed, spatial operations are performed to identify which OSM coordinates or features lie within the buffered area. This is achieved using spatial indexing and intersection techniques, which compare the locations of OSM features to the buffer's boundaries. For the OSM count features, the total buffer was also used to normalize features for a fair comparison between shorter and longer tracks. A buffer of 25 meters is used but has no difference compared to different buffer sizes (10, 20, 50, 75, and 100 meters).

Pre-processing of GPS and OSM features

Some GPS calculations did not result in reasonable numeric values.

- If the calculation of a feature resulted in an infinite value, the infinite value was replaced with twice the maximum value (inf → 2 * max).
- A negative infinity value was set to zero (-inf→ 0).
- Missing values remained unchanged since a decision tree can handle missing data.
- String variables were factorized for the decision tree.

For the OSM features, some count variables contained missing values. This occurs when there is no OSM infrastructure in the buffer of a track. Here, the missing data was replaced with a zero count, reflecting this feature's actual absence.

Decision tree development

A decision tree is a supervised learning algorithm used for classification and regression tasks. It is a tree-like model where each internal node represents a decision based on a feature, each branch represents an outcome of that decision, and each leaf node represents a final prediction. The process of dividing a node into two or more sub-nodes is based on feature conditions.

Grid search was done which is a hyperparameter tuning technique used to find the best combination of parameters that optimize the model's performance. It systematically searches through a predefined set of hyperparameters by testing all possible combinations and selecting the best one based on a scoring metric. The hyperparameter search was conducted using the following space:

$$
\begin{aligned}
\text{Maximum depth:} \quad & d \in \{3, 5, 7\}, \\
\text{Minimum samples split:} \quad & m_{\text{split}} \in \{10, 25, 50\}, \\
\text{Minimum samples leaf:} \quad & m_{\text{leaf}} \in \{5, 10, 15, 20, 25\}, \\
\text{Criterion:} \quad & c \in \{\text{gini}, \text{entropy}\}, \\
\text{Maximum features:} \quad & f \in \{1, 3, 5, 7\}.
\end{aligned}
$$

Although the hyperparameter space should have limited the number of features, the final decision tree contains 39 features. This is, because the hyperparameters are not strictly enforced.

5.3.2. Implementation

The algorithm is currently implemented in Python. The Git repository (https://github.com/essnet-ssi/geo-transportmode-prediction-ssi) contains the following Python scripts that contain the code required to implement the transport mode prediction algorithm.

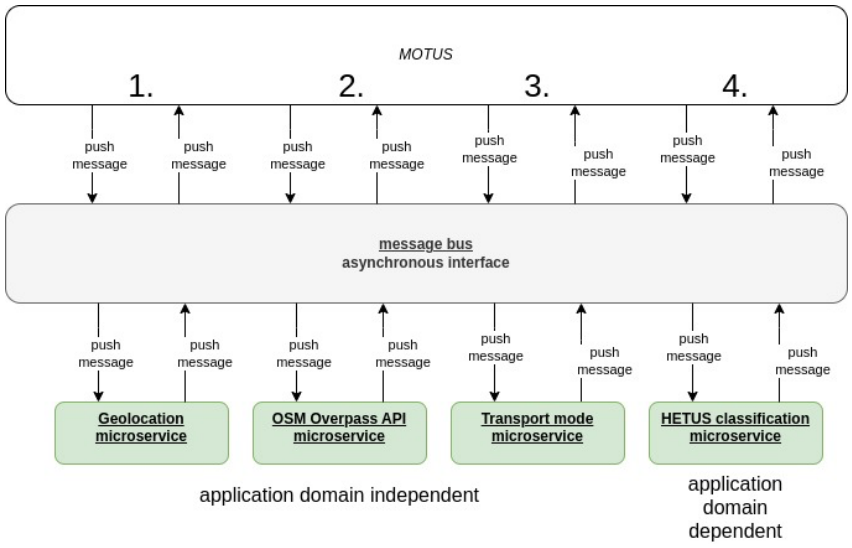
- transport mode main.py

    - The main script for transport mode prediction that will load and run the other scripts.
- options.py
    - This script contains all options regarding file paths, data preprocessing and model training required in the other scripts.
- functions general.py
    - This script contains general functions required for the transport mode prediction process.
- gps features.py
    - Contains functions for gps-based feature creation for events and locations data. These features are added to the events dataframe.
- osm features.py
    - Contains functions for osm-based feature creation for events and locations data. These features are added to the events dataframe.
- train decision tree.py
    - This script runs a grid search over a hyperparameter set to train the best decision tree model for the given data. The current best result is 20250424 decision tree ssi.pickle.
- decision tree ssi.pickle
    - The best decision tree model for the available development data resulting from the combination of options, feature creation and model training.

5.3.3. Integration

In MOTUS, transport mode prediction is integrated as an independent microservice as illustrated in Figure 32:

Figure 32: Microservices integration in relation to MOTUS



Transport prediction on the tracks is performed in step 3. of the geo pipeline.

5.3.4. Test

Tests are done via the development data, and the open geo-data.

The algorithm's performance will be assessed using precision, recall, F1-score, accuracy, and balanced accuracy, metrics commonly used in transport mode classification. Precision evaluates the model's ability to minimize false positives, while recall measures its ability to capture true positives. The F1-score combines both metrics to provide a balanced evaluation, particularly useful for imbalanced datasets. Accuracy represents the overall correctness of predictions but can be misleading in imbalanced data, where balanced accuracy offers an evaluation by averaging recall across all classes. Key definitions include true positive (TP), when the model correctly predicts the actual class (e.g., predicting 'walking' when correct), false positive (FP), where an incorrect class is predicted (e.g., predicting 'car' instead of 'bike'), false negative (FN), when the correct class is missed, and true negative (TN), when incorrect classes are correctly excluded. The formulas for each metric are:

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

$$\text{F1-Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}$$

$$\text{Balanced Accuracy} = \frac{1}{2}\left(\frac{\text{TP}}{\text{TP} + \text{FN}} + \frac{\text{TN}}{\text{TN} + \text{FP}}\right)$$

Evaluation on development set

The results from the confusion matrix in Table 13 and classification report in Table 14 for the training data indicate the following key findings: The confusion matrix reveals that the model performs well in predicting car and walking but struggles with categories like bus and tram, where most instances are misclassified. Bike also shows a strong prediction rate, though some misclassifications occur with cars and walking. The `Other' category is highly misclassified, with many instances incorrectly labelled as car or bike, indicating potential difficulties distinguishing less common transport modes.

Table 13: Classification report for training data

Observed\Predicted	Other	Car	Bus	Bike	Metro	Tram	Train	Walking
Other	0	64	0	25	0	0	5	9
Car	0	1161	2	106	5	0	20	39
Bus	0	46	3	15	6	0	3	2
Bike	0	31	0	554	2	0	4	52
Metro	0	4	0	1	27	0	8	5
Tram	0	7	0	7	3	0	1	0
Train	0	10	0	5	1	0	82	1
Walking	0	29	0	48	0	0	2	595

The classification report reveals that the model performs well for high-frequency classes like car and walking, achieving precision, recall, and F1-scores around 0.85-0.87, indicating strong and balanced performance for these categories. Bike and train also show relatively high F1 scores (0.79 and 0.73, respectively), with the train having a high recall (0.83) despite moderate precision. However, the model struggles with underrepresented classes. Other and tram have 0.00 F1-scores, as the model fails to classify these instances correctly. Buses have low performance, with an F1 score of 0.07, mainly due to extremely low recall, meaning most buses are misclassified as other categories (especially cars). Metro performs better, with precision and recall around 0.60--0.61, but still shows room for improvement.

Table 14: Classification report for training data

Class	Precision	Recall	F1-Score	Support
Bike	0,73	0,86	0,79	643
Bus	0,60	0,04	0,07	75
Car	0,85	0,87	0,86	1.333
Metro	0,61	0,60	0,61	45
Train	0,66	0,83	0,73	99
Tram	0,00	0,00	0,00	18
Walk	0,85	0,87	0,86	684
Other	0,00	0,00	0,00	103
Accuracy			0,81	3.000
Macro avg.	0,54	0,51	0,49	3.000
Weighted avg.	0,77	0,81	0,78	3.000

Although the overall accuracy is relatively high at 81%, the macro average F1-score of 0.49 and balanced accuracy of 0.51 indicate poor performance in less common classes. The weighted average F1-score of 0.78 is boosted by the well-classified dominant classes, masking the severe misclassification of minority classes. This suggests the model may be biased towards common classes, struggling to capture the nuances of less common transport modes.

The results from the confusion matrix in Table 15 and classification report in Table 16 for the test data indicate the following key findings: The model performs well for high-frequency classes like car, bike, train, and walking, with relatively high precision, recall, and F1-scores. For instance, car has an F1-score of 0.85, and walking achieves 0.84, reflecting consistent performance compared to the training set, where these classes also had high scores. Train maintains strong recall (0.87) and a high F1-score (0.82), showing that the model reliably identifies most train instances. Similarly, bike achieves an F1-score of 0.76, demonstrating the model's ability to generalize reasonably well to this class.

Table 15: Confusion matrix of test data

Observed\Predicted	Other	Car	Bus	Bike	Metro	Tram	Train	Walking
Other	0	3	0	1	0	0	0	4
Car	4	459	0	67	6	0	9	18
Bus	0	15	0	4	1	0	0	1
Bike	0	12	0	249	6	0	1	35
Metro	0	2	0	1	4	0	5	1
Tram	0	3	0	3	7	0	0	1
Train	0	5	0	2	0	0	54	1
Walking	0	18	0	24	2	0	0	274

However, the confusion matrix shows various misclassifications, especially for underrepresented classes. For example, `Other' is never classified correctly, with instances being mistaken for car, bike, or walking -- mirroring the training set where `Other' had an F1-score of 0.00. Bus also performs poorly, with all instances misclassified, mostly as car or bike, leading to a 0.00 F1-score, just like in training data. This suggests the model struggles to learn meaningful patterns for rare classes, likely because of class imbalance and overlapping features.

The metro and tram classes continue to be problematic. Metro shows a slight improvement over the training set, with a 0.21 F1-score on the test set, but remains low, with many instances misclassified as train. Tram remains entirely misclassified, with an F1-score of 0.00, indicating the model failed to generalize this class from the training set to the test set. These results indicate that minority classes are poorly represented in the decision boundaries, possibly because the model is biased toward more common classes like car and walking.

Table 16: Classification report for test data

Class	Precision	Recall	F1-Score	Support
Bike	0,71	0,82	0,76	303
Bus	0,00	0,00	0,00	21
Car	0,89	0,82	0,85	559
Metro	0,15	0,31	0,21	13
Train	0,78	0,87	0,82	62
Tram	0,00	0,00	0,00	14
Walk	0,82	0,86	0,84	318
Other	0,00	0,00	0,00	8
Accuracy			0,80	1.298
Macro avg.	0,42	0,46	0,44	1.298
Weighted avg.	0,79	0,80	0,79	1.298

Despite an overall accuracy of 80%, a macro average F1-score of 0.44 and a balanced accuracy of 0.46 reveal that performance varies widely across classes, with the model performing well on frequent categories but failing on rare ones. The weighted average F1-score of 0.79 is heavily influenced by the well-classified majority classes, masking the poor recognition of smaller classes. The test results confirm the patterns observed in training: the model captures dominant class features well but struggles with minority classes, leading to repeated misclassification patterns across both datasets.

Evaluation on open geo-data

The results from the confusion matrix in Table 17 and classification report in Table 18 for the test on the open geo-data show the following key findings:

The confusion matrix reveals considerable misclassification patterns, particularly among specific transport modes. Walking is the most accurately predicted class, with 52 correct classifications, though it is still confused with bike (16) and car (9). Tram shows the highest misclassification rate, frequently being predicted as car (14), bike (7), or metro (5). Bus is rarely identified correctly and is often confused with car, bike, metro, and train. Similarly, the ferry is misclassified as the bike. The label "ferry" did not appear in the observed labels in the development data. However, this label was kept to study what prediction would result for this label. This is especially interesting, because GPS signals usually get noisy when the smartphone is close to or on the water. Train and metro show moderate accuracy, but car and other categories are never correctly predicted. These results highlight challenges in distinguishing between modes with similar speed profiles and infrastructure characteristics.

Table 17: Confusion matrix of open geo-data

Observed\ Predicted	Other	Car	Bus	Ferry	Bike	Metro	Tram	Train	Walking
Other	0	0	0	0	0	0	0	0	0
Car	0	0	0	0	0	0	0	0	0
Bus	0	2	0	0	1	1	0	1	0
Ferry	0	0	0	0	3	0	0	0	0
Bike	0	2	0	0	7	1	0	0	0
Metro	0	3	0	0	0	1	0	0	0
Tram	0	14	0	0	7	5	0	0	1
Train	0	3	0	0	0	0	0	6	1
Walking	0	9	0	0	16	1	0	0	52

The classification report highlights performance variations across transport modes. Walking and the train achieve the highest F1 scores (0.79 and 0.75, respectively), indicating relatively good performance. The bike also shows moderate recall (0.70) but low precision (0.21), leading to a modest F1-score of 0.32. In contrast, bus, ferry, tram, car, and other categories are never correctly identified, resulting in F1-scores of 0.00. Metro has a low F1-score (0.14) due to poor precision and recall. The overall accuracy is 0.48, the balanced accuracy is 0.32, and the macro F1-score of 0.22 reflects substantial class imbalances and misclassification issues, particularly for underrepresented classes.

Table 18: Classification report for open geo-data

Class	Precision	Recall	F1-Score	Support
Other	0.00	0.00	0.00	0
Car	0.00	0.00	0.00	0
Bus	0.00	0.00	0.00	5
Ferry	0.00	0.00	0.00	3
Bike	0.21	0.70	0.32	10

Metro	0.11	0.20	0.14	5
Tram	0.00	0.00	0.00	27
Train	0.86	0.67	0.75	9
Walking	0.96	0.67	0.79	78
Accuracy			0.48	137
Macro avg.	0.24	0.25	0.22	137
Weighted avg.	0.62	0.48	0.53	137

Feature importance

Table 19 shows the feature importance of the top 20 selected features. In total, 39 were selected. The feature importance results reveal that the model relies heavily on a few key features, with `bus route mean distance' as the most influential feature, contributing 22.3% to the decision-making process. This suggests that distance patterns along bus routes play a critical role in distinguishing transport modes. Speed-related features also dominate the model's decisions, with metrics like `speed IQR' (17.7%), `speed percentile 10' (5.1%), and `speed standard deviation' (3.1%) collectively contributing a large share of the importance. This heavy reliance on speed variation could explain the model's struggles with modes with overlapping speed ranges (e.g., bus vs. car or metro vs. train). Interestingly, railway station count (10.4%) is another essential feature, likely helping the model identify train and metro trips. Proportion-based speed features (e.g., proportion 45–80 km/h, 7.9%) also influence predictions, possibly helping differentiate slower modes like walking from faster ones like cycling or driving. Lower-ranked features, like jerk percentile 85 (0.9%) and tram route standard distance (0.75%), contribute minimally. Overall, the model leans heavily on speed and distance metrics, explaining its success with frequent modes like cars and bikes and its failures with underrepresented classes. Strengthening the model with more contextual features (see discussion) or refining route-based features for specific transport modes could help improve classification performance, especially for minority classes.

Table 19: Top 10 decision tree feature importance

Feature	Importance
Bus route mean distance	0.223053
Speed iqr value	0.176813
Railway station normcount	0.103649
Proportion 45 80	0.079107
Speed percentile 10	0.051288
Proportion 5 15	0.039756
Speed stddev	0.030690
Speed average	0.030036
Speed percentile 90	0.027969
Speed percentile 80	0.024554
Bus route std distance	0.021325
Bus route max distance	0.021116
Proportion 15 30	0.019196
Speed percentile 85	0.016795
Acc kurt	0.015727
Accuracy percentile 85	0.012062
Jerk percentile 85	0.009044
Proportion 80 120	0.008715
Tram route std distance	0.007534
Speed median value	0.007435

5.4. HETUS classification microservice (GeoService Microservice – part 2)

The GeoService Microservice is supported by two microservices: the Geolocation Microservice (part 1) and the HETUS classification Microservice (part 2).

The second part of the GeoService aims to provide a prediction about the most likely activities carried out by the user during a stop. For the activity taxonomy, we refer to HETUS, "Harmonised European Time Use Survey".

To predict the activity performed by the user during a stop, several data are exploited: the spatiotemporal characteristics of the stop, the types of points of interest near the stop; the socio-demographic characteristics of the user (age groups and employment status).

This important information is the input of the algorithm and must be provided in the request to the service, specifically the spatiotemporal characteristics and the list of points of interest (POI) in the stop, provided by the GeoService Microservice part 1 as described in section 5.2.

The algorithm underlying this microservice exploits mainly the data collected by the TUS surveys of the NSI and the contextual data through a probabilistic model, as a labelled dataset associating a set of input variables with the activity carried out needed to train a machine learning algorithm was not available.

5.4.1. Design

The steps of the pipeline of the algorithm for the Activity prediction are the following:

- Pre-processing of input data
- Assignment of Categories of place to each POI of the stop
- Assignment of a score (POI-score) to each POI
- Identification of a shortlist of POIs, applying the elbow criterion to the POI scores
- Application of a statistical model to predict stop activity with respect to each POI
- Assignment of a rank to the HETUS activities of the stop based on a final aggregated score.

Input structure description

For each stop identified by the ATS_OPTICS algorithm, the input of the algorithm consists of:

- For each GPS point of the stop:
  - GPS Longitude
  - GPS Latitude
  - GPS Accuracy
  - GPS Timestamp
  - GPS Speed
- For the stop:
  - Centroid Longitude
  - Centroid Latitude
  - Time Start
  - Time End
  - Duration
  - Radius
- For each Point of Interest (POI) inside the radius of the stop:
  - POI Longitude
  - POI Latitude
  - Tag (textual description provided by MapService)

Other data coming from the platform concerns the profile of the user:

  - Age class
  - Condition (employed, student, other)

Pre-processing

Data cleaning operations are performed by filtering out low-quality GPS points, specifically those with low accuracy or excessively high speed. These parameters can be configured through an algorithm parameter configuration file.

Additionally, each stop is assigned a time slot based on the classification provided in Table 20. The assignment of the time slot depends on the stop's start and end times. If the stop interval overlaps with two or more time slots, the one with the greatest temporal overlap is selected.

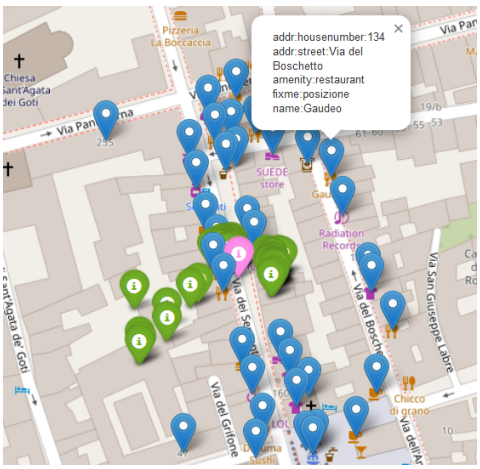
Table 20: List of time slots

Slot ID	Time Interval	Description
0	08:00 – 12:30	Morning
1	12:30 – 14:30	Lunch/Dinner
2	14:30 – 19:30	Afternoon
1	19:30 – 21:30	Lunch/Dinner
3	21:30 – 00:00	Night
4	00:00 – 08:00	Early morning

Assignment of Categories of place to each POI of the stop

The textual description depends on the map service used. In Google Maps, the name and type of the POI can be used. In Open Street Maps, the name of the place and the values of tags such as Amenity, Shop, Office, Leisure, etc., are used.

Figure 33: Example of stop, POIs and sample of GPS points



In Figure 33, some information about the input to the microservice is shown. The blue points are the POIs in the stop, the green points are a sample of the GPS points belonging to the stop, and the pink point is the centroid of the stop. For one of the POIs, the tag provided by the OSM map service is displayed.

Each point of interest is classified according to the Italian Time Use Survey classification of the places (TUS place) where the user's activity takes place. The TUS place classification is a specific classification of locations used internally at the statistical institute that provides TUS data. In particular, in the developed prototype, the algorithm has been developed based on data from the Italian TUS survey (Table 21).

Table 21: List of Italian TUS places

TUS_PLACE	TUS_PLACE Description
3	Other people's home
4	Hotel and similar
5	Workplace
6	School, university
7	Library
8	Study center
9	Restaurant, pizzeria, brewery, ice cream bar
10	Shopping center
11	Market
12	Shops
13	Public offices
14	Hospital, clinic
15	Sports venues
16	Entertainment venues
17	Discotheque
18	Museum, exhibitions
19	Theme park, game room
20	Places of worship, oratory
21	Equipped public green spaces (park, garden, villa)
22	Green area (countryside, mountain, meadow, forest)
23	Sea, beach
24	River, lake
25	Road, other places

The link between the tag and the TUS place is made through keyword searches and regular expressions.

Table 22: Example of association between the tag and the TUS place

lat	lon	scorePoi	tag	TUS_PLACE
41.895596	12.490317	0.043904	amenity:restaurant name:Temakinho Rome Monti	09
41.895668	12.490338	0.041848	addr:housenumber:130 addr:street:Via dei Serpenti amenity:fast_food name:il Gelatone	09
41.895516	12.490369	0.031499	amenity:restaurant cuisine:regional name:Al Vino al Vino restaurant:type:enoteca	09
41.895688	12.490533	0.028297	name shop	12
41.895742	12.490499	0.023218	name name:it shop	12
41.895758	12.490278	0.020034	addr:street:Via dei Serpenti amenity:cafe cuisine:sandwich;coffee_shop internet_access:wlan name:Antico Caffe del Brasile - Torrefazione name:de:Antico Caffe del Brasile - Torrefazione	09
41.895919	12.490231	0.010666	addr:housenumber addr:street delivery name operator phone ref:vatin shop	12

Table 22 shows, for each POI, the textual description, provided by the map service (tag), the TUS PLACE associated through regular expressions, and the POI score assigned based on the time slot and the position of the GPS points and POIs.

Assignment of a score to each POI and shortlist of POIs

For each point of interest (POI) in the stop, a score is calculated based on the median distance between the POI and the GPS points associated with the stop, weighted by their accuracy. This score

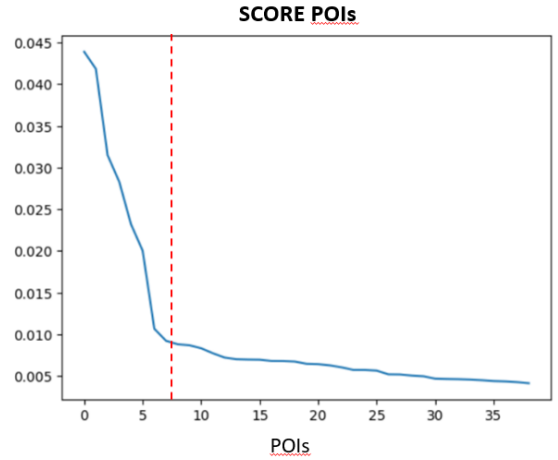
is further weighted by the probability of carrying out any activity in the TUS place associated with the POI and within the specific time slot assigned to the stop. The formula is the following:

$$ScorePOI_i = median_j^{N_{GPS}} \left(\frac{1}{distance(GPS_j, POI_i) accuracy_j} \right) P(TUS\_PLACE_{POI_i}, TimeSlotStop)$$

The probabilities in the formula are estimated by the TUS data.

Next, a shortlist of POIs with the highest POI scores is defined. The selection is made by identifying where the slope of the POI Score curve changes significantly, using the *elbow criterion*.

Figure 34: Example of shortlist of POIs based on the Elbow criterion



Statistical model to predict stop activity with respect to a POI

For each POI in the shortlist, the probability of performing an activity according to the HETUS classification concerning a POI is calculated. This probability is decomposed using a Bayesian approach, as outlined in the following formula:

$$P(A_i|x,t) = \frac{Lognormal(t, \mu_{x,A_i}, \sigma_{x,A_i}) P(x|A_i) P(A_i)}{\sum_j P(x|A_j) P(A_j)}$$

where A_i is the HETUS activity, x is the tuple (user's condition, user's age class, TUS PLACE of the POI) and t is the duration of the stop, while μ_{x,A_i} and σ_{x,A_i} are respectively the mean and standard deviation associated with an activity for a user's condition, user's age class, and TUS PLACE of the POI.

In accordance with the chosen approach, the model parameters μ_{x,A_i} and σ_{x,A_i} and the probabilities $P(x|A_i)$ and $P(A_i)$ are estimated using aggregated counts from the TUS survey data.

For clarification, an example is presented showing the aggregated TUS counts. The example shows that in the TUS survey, the HETUS activity 3.6.1 'Shopping (including online/ e -shopping)' performed by employed respondents (condition=1), aged 25-44 years (age class=2), carried out at a shop (TUS place=12), was recorded 2151 times with an average duration of 39 minutes and a standard deviation of 35 minutes.

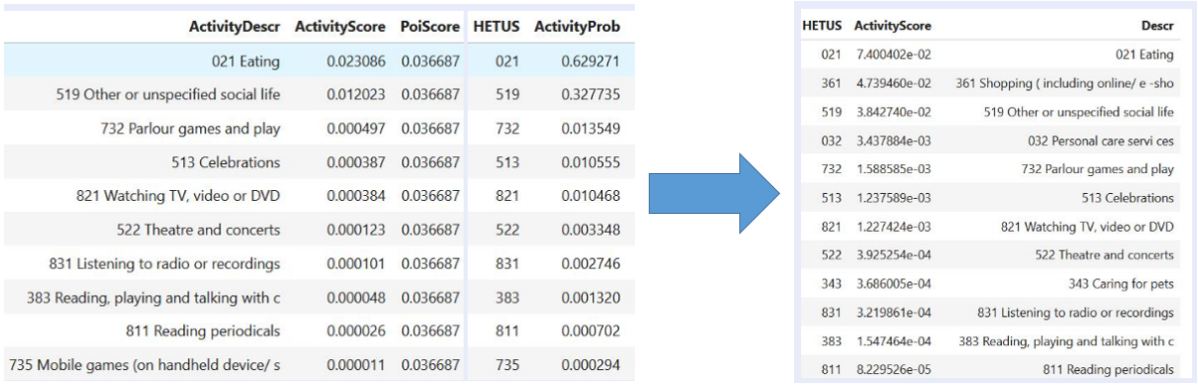
condition	age class	TUS place	HETUS	COUNT	DURATION mean	DURATION sd
1	2	12	3.6.1	2151	00:39	00:35

Assignment of a rank to the HETUS activities

Finally a rank (ActivityScore) of the HETUS activities is assigned to the stop, based on a final score calculated aggregating the probabilities of the activity weighted by the POI-score associated with the activity for each POI in the shortlist.

$$ActivityScore_A = \sum_{POI}^{POI\_shortlist} Score_{POI}\ ActivityScore_{A,POI}$$

Figure 35: Example of final ranking of predicted activities



ActivityDescr	ActivityScore	PoiScore	HETUS	ActivityProb
021 Eating	0.023086	0.036687	021	0.629271
519 Other or unspecified social life	0.012023	0.036687	519	0.327735
732 Parlour games and play	0.000497	0.036687	732	0.013549
513 Celebrations	0.000387	0.036687	513	0.010555
821 Watching TV, video or DVD	0.000384	0.036687	821	0.010468
522 Theatre and concerts	0.000123	0.036687	522	0.003348
831 Listening to radio or recordings	0.000101	0.036687	831	0.002746
383 Reading, playing and talking with c	0.000048	0.036687	383	0.001320
811 Reading periodicals	0.000026	0.036687	811	0.000702
735 Mobile games (on handheld device/ s	0.000011	0.036687	735	0.000294

HETUS	ActivityScore	Descr
021	7.400402e-02	021 Eating
361	4.739460e-02	361 Shopping (including online/ e -sho
519	3.842740e-02	519 Other or unspecified social life
032	3.437884e-03	032 Personal care servi ces
732	1.588585e-03	732 Parlour games and play
513	1.237589e-03	513 Celebrations
821	1.227424e-03	821 Watching TV, video or DVD
522	3.925254e-04	522 Theatre and concerts
343	3.686005e-04	343 Caring for pets
831	3.219861e-04	831 Listening to radio or recordings
383	1.547464e-04	383 Reading, playing and talking with c
811	8.229526e-05	811 Reading periodicals

Figure 35 describes an example of how the algorithm returns to the platform a score associated with the most likely HETUS activities given the spatiotemporal characteristics of the stop, the user's characteristics, and the type of POIs around the stop.

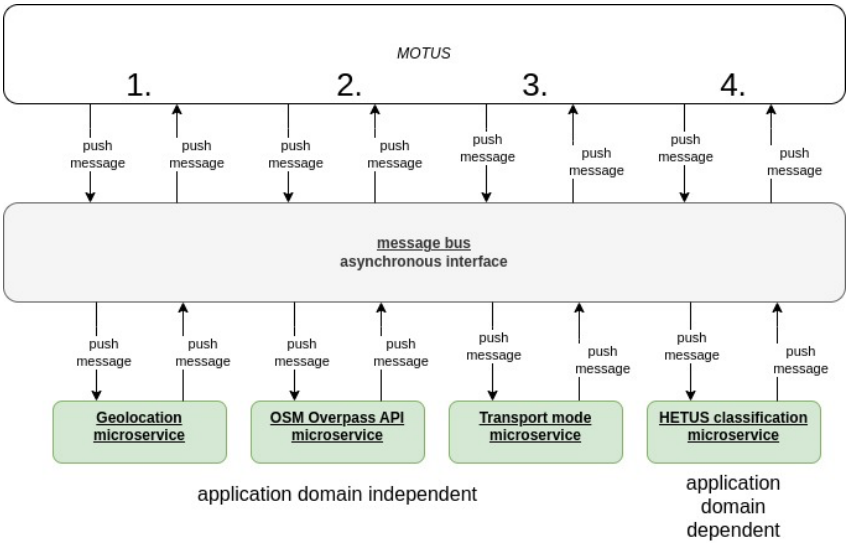
5.4.2. Implementation

An implementation in Python has been created and is available on Github at link *https://github.com/essnet-ssi/HETUS-classification-microservice*. Modifications to the model are planned with the addition of input features useful for identifying the activity, such as the start time of the stop or the day of the week. This implies a modification to the formula for calculating the model based on the aggregated TUS data.

5.4.3. Integration

In MOTUS, the HETUS classification microservice is integrated as an independent microservice as illustrated in Figure 36:

Figure 36: Microservices integration in relation to MOTUS



The classification is performed in step 4. of the geo pipeline.

5.4.3. Test

The HETUS Classification algorithm was initially tested in its complete configuration, utilizing all available input data and features. The algorithm's performance - accuracy and reliability - was thoroughly evaluated under these conditions. The validation process was conducted using the available annotated data.

Subsequently, to identify the relevance of the considered features for predicting HETUS activities, the algorithm's performance was further assessed by varying the input data settings. This involved testing different combinations of features to assess their impact on prediction accuracy and overall model performance. This was designed to simulate situations in which NISs do not have available or do not want to use data from a time-use survey, or there are constraints on the use of personal data in the microservice.

The test was conducted on the HETUS Classification algorithm after the implementation of the previous microservice (Geolocation microservice for determining stop clusters and adding context to stops - external sources consulted to add a list of nearby places/shops).

Pipeline

The pipeline for the test is the following:

1. Data collection: GPS data collected using the app CBS-Odin (made available by CBS for downloading from Google Store), for 4 respondents over 4 days
2. Stop identification: from GPS traces data are segmented using the algorithm ATS-OPTICS
3. POI identification: selection of the points of interest inside the stop by REST query to Map Service (Google Place)
4. Labelling: For the stops identified the users labelled the place and the activities carried out
5. Work and home identification: stops related to place of work and home are excluded from the analysis

6. HETUS activity prediction: For each stop the algorithm returns the most likely activity prediction and the place type
7. Prediction evaluation: calculate metrics for:
    - HETUS activity three-digit
    - Number of Match Top 1: Number of stops where the activity with the highest prediction score is correctly predicted by the algorithm;
    - Number of Match Top 2: Number of stops where one of the two highest-scoring activities is correctly predicted by the algorithm;
    - Number of Match Top 3: Number of stops where one of the three activities with the highest scores is correctly predicted by the algorithm.

Results of algorithm validation

Table 23 summarises the results obtained by analysing the total number of identified stops and their matches with the Google Places map. First, the stops that are useful for HETUS activity prediction (containing some POIs) were identified. The table highlights, among the stops with identified POIs, those where the predicted activity matches the assigned label. The analysis considers the rank assigned to the HETUS activities for each stop, which is determined by a final score combining the activity probabilities weighted by the POIs score in the shortlist.

Table 23: Stops detection and matching with HETUS activities on the basis of assigned rank

	STOPS /Google Places
Stop segmented by ATS/OPTICS	111
Stop with duration > 5 minutes	83
Stop with POIs useful for activity prediction	71
Stops with identified POIs (not labelled home/work)	21
Stops with identified POIs (not labelled home/work)	
MATCH TOP 1	11
NO MATCH TOP 1	10
MATCH TOP 2	12
NO MATCH TOP 2	9
MATCH TOP 3	13
NO MATCH TOP 3	8

The evaluation of activity prediction accuracy reveals several cases, both correct and incorrect prediction, depending on the context of the stop.

Below is a detailed breakdown of these cases:

- *Cases of Correct Activity Prediction*

1. Correct POI and Activity:
   The correct POI is identified in the stop and the first activity listed in the shortlist matches the actual activity. This represents an ideal scenario where both the location and activity are accurately predicted.
2. Incorrect POI but Correct Activity:
   Although the identified POI is incorrect, the activity is still correctly assigned. This occurs when multiple similar POIs are present around the stop, and the algorithm assigns the correct activity despite the POI mismatch.

- *Cases of Incorrect Activity Prediction*

3. Private Residence with Nearby POIs:
   The stop corresponds to a private house (e.g., a friend's home) that is not the respondent's residence. While some POIs may be present nearby, the activity prediction is incorrect due to the identification of an irrelevant POI. In such cases, the respondent is expected to correct the place during validation.
4. Correct POI but Atypical Activity:
   The correct POI is identified, but the predicted activity is atypical for that location (e.g., a concert in a public park). This results in an incorrect activity assignment due to the unusual nature of the event.
5. Ambiguous POIs with Multiple Activities:
   The map provides several POIs in the stop, each associated with different activities. The algorithm selects an incorrect activity due to the ambiguity in the available choices.
6. POI Without Tags:
   In cases where the POI lacks sufficient tags or metadata, classification becomes impossible, leading to an inability to predict the activity accurately.

In the following figures some examples of cases 1, 4 and 5 are shown.

Figure 37: Correct POI and Activity: the respondent was in a restaurant, the activity was 0.2.1 eating, and the predicted activity is 0.2.1 eating

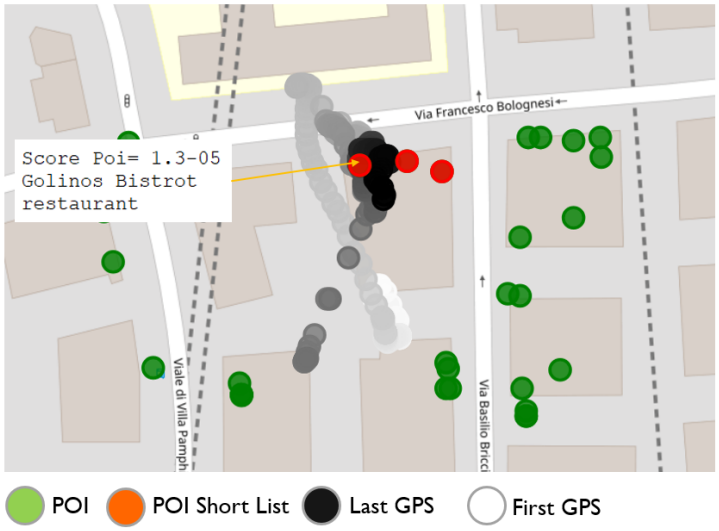


Figure 38: Correct POI but atypical activity: the respondent was in a school, the activity was 6.1.0 Other or unspecified sports or outdoor activities but the predicted activity is 2.1.1 Classes and lectures

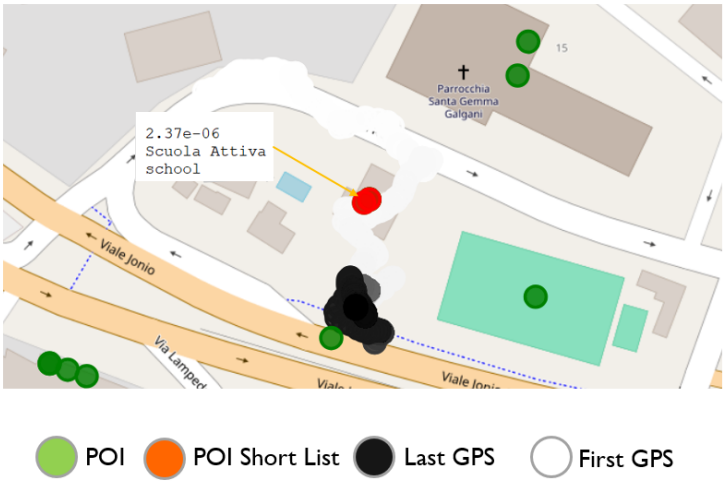
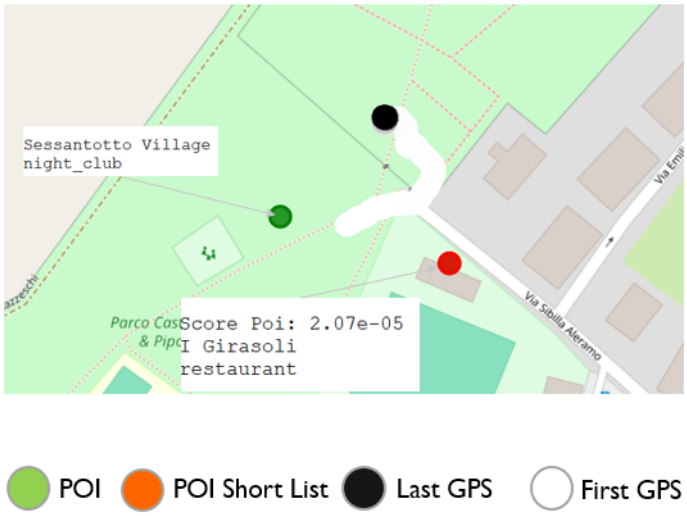


Figure 39: Ambiguous POIs with multiple activities: the respondent was in the park, the activity was 6.1.9 Other or unspecified sports or outdoor activities but 0.2.1 eating associated with the wrong place restaurant



Quality assessment

To evaluate the performance and quality of the algorithm, various configurations of the input data and features were examined. These configurations included:

- Different map services: Utilizing alternative map services (e.g., OSM and GP) as input data sources.
- Varied use of TUS data: Exploring different ways of incorporating information from the Time Use Survey (TUS) data to assess its impact on the algorithm's performance.

Depending on the research and privacy rules, it may not be possible to use the respondent's information (age class, condition) in the microservice. In this case, the prediction will be carried out with the other available information. The impact of this lack of information on the quality of the HETUS activity prevision was measured in this assessment phase.

Table 24 shows the performance of the algorithm when using OSM as context information. From the results (matches pass from 11 to 7) it can be seen that OSM is a lower-quality auxiliary source than GP for the activity prediction objective. In fact, it contains fewer POIs, i.e. it is much less complete than GP. Besides, from a direct personal knowledge of places (in Rome), OSM also is much less updated than GP. This fact would deserve an in-depth study, which is difficult to carry out in every country.

Table 24: Stops detection and matching with HETUS activities on the basis of assigned rank

	STOPS /Open Street Map
Stop segmented by ATS/OPTICS	111
Stop with duration > 5 minutes	83
Stop with POIs useful for activity prediction	56
Stops with identified POIs (not labeled home/work)	24
Stops with identified POIs (not labelled home/work)	
MATCH TOP 1	7
NO MATCH TOP 1	17
MATCH TOP 2	8
NO MATCH TOP 2	16
MATCH TOP 3	8
NO MATCH TOP 3	16

Table 25 shows the performance of the algorithm when using different algorithm settings for Google Places map service:

- Rule based: the algorithm does not use the probabilities based on TUS data but only the context information linked directly to activities, using a table linking places to HETUS activities.
- Probabilistic, no personal data: the algorithm uses the probabilities based on TUS data (only time slot and places) but not the personal user data (age group and employment status).
- Probabilistic, no time slot: the algorithm uses the probabilities based on TUS data with the personal user data but without time slot (distributions that also include frequencies for time slots may not be very meaningful as based on few data).

Table 25: Performance of the algorithm when using different algorithm settings

	Top 1		**Top 2**		**Top 3**	
Algorithm setting	MATCH	NO MATCH	MATCH	NO MATCH	MATCH	NO MATCH

Rule based	10	10	11	9	11	7
Probabilistic, no personal data	11	9	13	7	13	7
Probabilistic, no time slot	6	14	11	9	13	7
Probabilistic, full setting	11	9	13	7	13	7

Table 25mainly shows that the use of data referring to the respondent (age group and employment status) does not seem to improve the quality of the prediction, although the analysis is based on a few data. Exploiting the time slot seems instead very relevant: excluding it, in fact, produces the worst results, even if using it requires, in general, the availability of the TUS survey data.

Summary

A prototypal algorithm for predicting HETUS activities to be provided as tentative data has been developed by exploiting stop location information, context information, TUS data and personal data.

From the testing phase, it emerged that a necessary condition for the algorithm to work well is that there is good-quality auxiliary information. In fact, if the stop identified by the segmentation does not contain useful POIs, the prediction is not obtainable. Furthermore, the use of Google Place as an auxiliary source improves the prediction compared to the use of Open Street Map. Therefore, in fact, the possibility of predicting the activity is based on the accuracy and completeness of the map service.

The accuracy of the algorithm is a function of the quantity and quality of the GPS points of the stop and the quality of the stop segmentation has an impact on the quality of the prediction.

The assessment phase also evaluated different scenarios that concern both the availability of TUS survey data and aspects related to privacy, i.e. the use of the respondent's personal data within the microservice.

We can conclude that the most relevant features for the quality of the prediction are the place type and the time slot, while the personal data do not particularly improve the prediction, although the limited set of data used for the evaluation does not allow us to draw unequivocal conclusions on this matter. This indicates that the time slot and the duration of the stop provide in any case a significant contribution to the prediction and its accuracy. Finally, the probabilistic approach seems to perform slightly better than the *rule-based* approach.

6. Energy Microservice

The Energy Microservice is the last environment being considered within the SSI-project. While also for this microservice the journey starts with outlining both the functional and non-functional requirements (6.1.), the development remains at the PoC level (6.2.).

6.1. Functional and non-functional requirements of the Energy Microservice

Historically, collecting precise data on a homeowner's energy usage has been challenging due to several factors, the most significant being respondent reluctance to participate. This hesitation is primarily driven by privacy concerns related to data collection methods, fears of user habit profiling, and apprehensions regarding data security. As a result, any solution designed to gather insights into energy usage must carefully address these concerns. Additionally, the process of effective data collection is complex and requires a certain level of technical expertise from participants. With these considerations in mind, Statistics Netherlands has initiated a small-scale exploratory in-house Proof of Concept (PoC) to evaluate these challenges.

We decided to start with a PoC because there are currently too many unknowns to already create a full smart service. First more information is needed on the technical possibilities of smart meters and smart plugs, what data is measured and what can be detected using this data. As the ideal smart service we think about collecting real-time data on energy, gas and water, determining the appliances used and what the respondents are doing using AI/ML models, and involve the respondents in filling in the gaps. However, as said, there are too many unknowns. As a result we decided to start with existing commercial solutions on the market and see what can be measured and detected using these. The rest of this document describes the business and functional requirements and setup of the PoC.

The PoC requires participants to receive and install a smart plug, following a step-by-step in-app guide to successfully connect the device to their home Wi-Fi network. The smart plug will commence real-time monitoring of energy consumption, gas usage, and, where applicable, the energy returned to the grid from solar panel systems. Simultaneously, participants are asked to maintain a manual diary for eight consecutive days, recording the number of people present in the household at different times, as well as the usage of power-intensive appliances, including their start time, end time, and duration. Additionally, participants are requested to document any factors that could affect the accuracy of insights, such as the presence of house guests. For the PoC, this will be documented in a separate document.

6.1.1. Business requirements

1. The system should gather real-time energy usage data from participants to support the daily energy consumptions, analysis of consumption patterns and help identify home appliances, including high-power appliances, like electric vehicles or energy storage, by correlating the data with user-reported inputs.
2. The collected data must undergo comprehensive analysis to evaluate the viability of employing smart plugs for energy monitoring, identify potential challenges, and provide insights to support future large-scale implementations.

3. The Proof of Concept (PoC) must establish standardized guidelines for participants to accurately document their energy usage, ensuring consistency in data collection and enhancing the reliability and validity of the findings.
4. The solution must implement stringent data protection measures to ensure that all collected information is anonymized, securely stored, and safeguarded against unauthorized access, thereby addressing participant privacy concerns.
5. The system should ideally incorporate a robust, secure, and well-documented API to facilitate seamless integration with third-party applications, research platforms, and analytical tools, thereby enhancing data interoperability and enabling extended analysis.

6.1.2. Functional requirements

With regards to data collection;

- The system must collect real-time energy consumption data from smart meters using a smart plug and/or API's that can be configured to collect data at specific intervals (e.g. every 15 mins, hourly)
- The manual data collection process is designed to ensure a highly qualitative and consistent method for recording logbook entries.

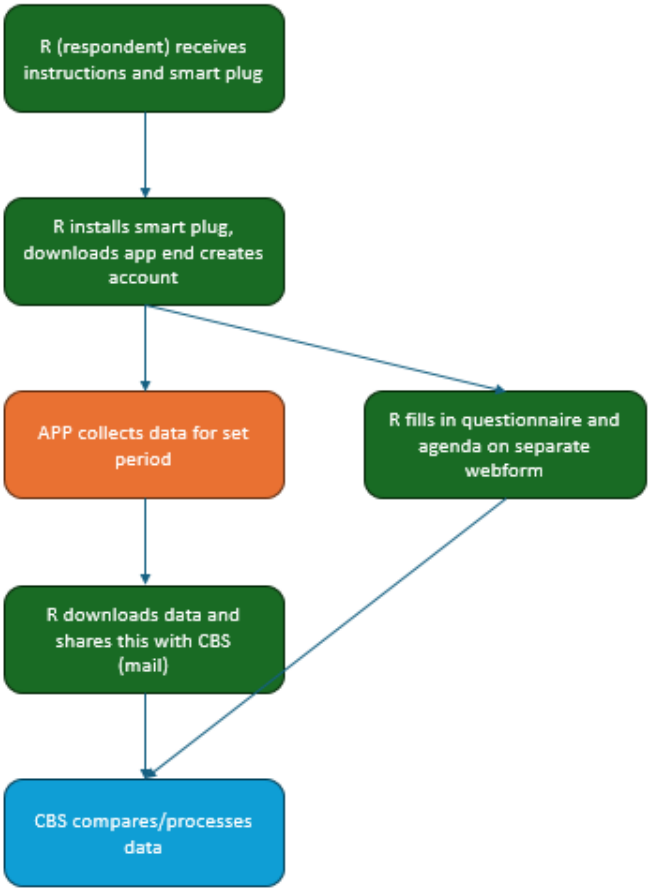
With regards to data processing, analytics and reporting;

- The system must aggregate, normalize, and store energy usage data.
- Energy consumption needs to be reported regularly (collected real-time, transferring data needs to be determined)
- The system provides an interactive dashboard displaying real-time energy usage. Ideally the system should provide a method to download periodic usage summaries.

With regards to third-party microservices and/or API

- REST/GraphQL APIs must be exposed for integration with external platforms.
- The API should give external parties (e.g. Statistics Netherlands) access to aggregated, anonymized data while ensuring secure authentication and authorization (OAuth 2.0).
- The microservice solution must be deployable at any given institute (for example through the use of containerization)
- The microservice solution is designed to be scalable according to the anticipated traffic levels, ensuring that performance is not significantly impacted.

Figure 40: Energy Microservice feasibility setup functional requirements – flow diagram



The diagram gives an overview of the main functional requirements for the PoC:

- functionality related to *user handling* is indicated by the green boxes.
- functionality related to the *app* is indicated by the orange boxes.
- functionality related to the CBS is indicated by the blue boxes.

User handling

Respondent handling (green boxes)	
1a	R needs to install smart plug
	Based on a 'plug-and-play' principle compatible with 'smart meters' which are already installed in the respondents' homes. The smart plug will connect with the WiFi-network.
1b	R downloads and installs mobile app
	This step includes creating an account and pay for the plus-service (subscription)
2	R fills ins questionnaire
	The questionnaire will contain a day-to-day overview of the usage of electronic devices
	The questionnaire will also contain some more general questions about the household
	Within this PoC the questionnaire will also contain some review question about the usage of the app and the process
3	R downloads data from app
	Because of the subscription, R will be not only be able to see the real-time usage but also to download an overview (.csv file) of the usage.
	After downloading the file, R will submit this data to CBS. In this PoC the file will be provided via e-mail

6.1.3. Non-functional requirements

- Performance
  The app shall respond to user interactions within an acceptable speed

- Security
  All user credentials are provided by the organization and stored securely using encryption
  Authentication must use secure token-based methods (e.g. OAuth2), with automatic session expiration after a defined inactivity period.
  All network communications shall use HTTPS with up-to-date TLS protocols.
  Sensitive data (such as usage patterns or location) must not be stored unencrypted on the device.
  The microservice solution is built on the principles of security by design.
  End-to-end encryption must be implemented for all (anonymized) data transmissions.
  The solution must comply with GDPR and ISO 27001 regulations.

- Usability
  The app must have a clean, intuitive user interface, designed according to mobile UX best practices.
  Users shall be able to view daily, weekly, and monthly energy usage in no more than 3 taps.
  The app must support dark mode and adapt to the device's accessibility settings (e.g. large fonts, voiceover).

- Reliability
  The app shall maintain an average crash rate of less than 1 crash per 10,000 sessions.
  In case of connectivity issues, the app shall cache the most recent data locally for offline viewing.

Automatic background syncing shall resume when network connectivity is restored.

- Compatibility
  The app shall support the latest 3 major versions of iOS and Android.
  It must display correctly across different device sizes, including phones and small tablets (4.7” and up).

- Maintainability
  The app codebase shall follow modular design principles and include proper inline documentation.
  The system shall include monitoring tools for performance and error tracking.

- Scalability
  Data load (e.g. charts, usage summaries) must be efficiently handled for both individual and aggregated usage metrics.

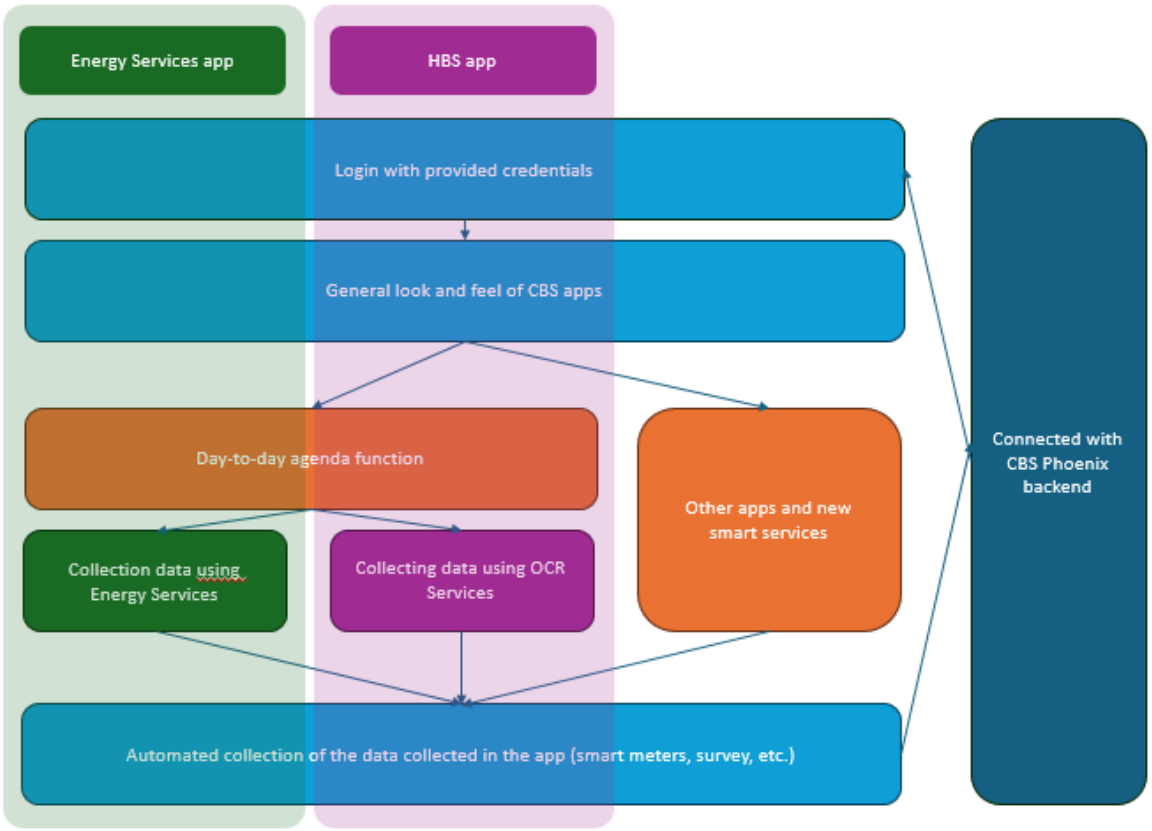
6.2. PoC Energy Microservice

At this moment we only conducted a PoC by using an external service. As a result, we have no integration into an app or our systems. However, with the knowledge and architecture already in place from the development of previous apps, it is possible to incorporate this service within the existing structure/architecture. Connections such as logging in and delivering data back into our (Phoenix) channel have already been delivered (securely). In addition, we can use the styling and design of existing apps as shown in the diagram below.

For the user, this will mean more ease of use. In fact, steps 1b and 3, as presented in the previous chapter, will be changed drastically.

- No account needs to be created
- No need to pay for a subscription
- No need to download the data
- No need to upload the data

Figure 41: Microservices integration in relation to CBS



With regards to security and compliance

- The microservice solution is built on the principles of security by design.
- End-to-end encryption must be implemented for all (anonymized) data transmissions.

The solution must comply with GDPR and ISO 27001 regulations

7. MOTUS platform

This chapter describes the MOTUS architecture, the deployment options and the pentest that has been carried out in relation to the MOTUS platform.

7.1. MOTUS architecture

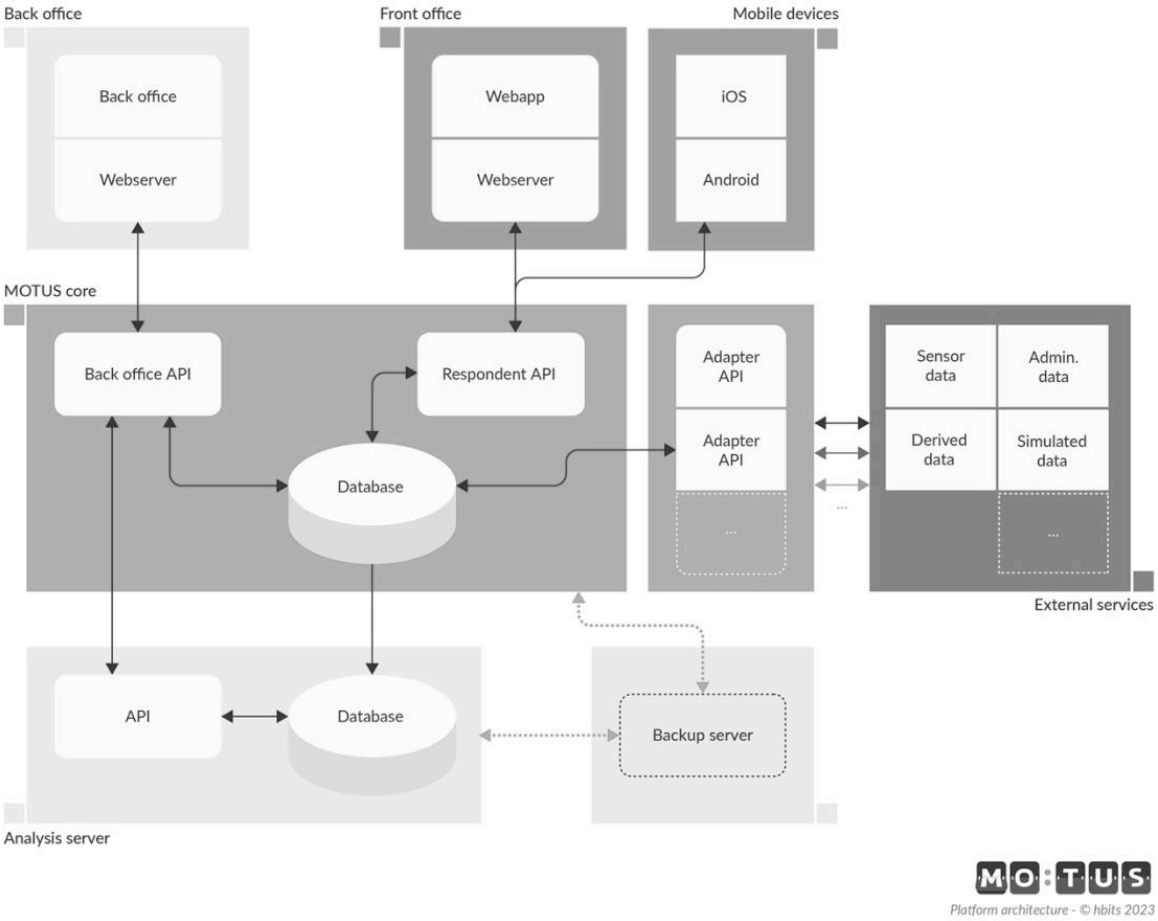
The figure shows the platform architecture of MOTUS. The MOTUS data collection platform consists of a front office as well as a back office.

The front office relates to the collection tool or application, with which the users can interact via a user interface (UI) and which delivers, through its functionalities, a user experience (UX). The MOTUS application is available as a web version for browsers (https://app.motusresearch.io) and in iOS and Android mobile versions for smartphones and tablets. The purpose of the application is to make it easier for the respondent to carry out all tasks of a (time use or other) survey.

The back office serves to build a study, to facilitate data collection and monitoring, and to process the data. To this end, the back office, which is accessible via a web environment, contains several MOTUS-builders with specific functionalities for e.g. survey, diary and communication.

Both the front office and back office connect to the MOTUS core (“the core”) through Application Programming Interfaces (APIs). The core holds the database with all information required to build a study and collect data. A separate analysis server (can) hold(s) a replica of the database from the core and facilitates the processing of information in the back office. The back-up server ensures the gathered data cannot get lost. Adapter APIs serve to adapt external information so that it can be processed in the core, thereby allowing the ingestion of, for example, passive data gathered via integrated sensors or connected devices, administrative/secondary data available via external data sources, or other processed data. For reasons of optimization, data security and privacy, these data are handled and organised in an anonymized way in stand-alone microservices. All input provided by the user is sent encrypted via https communication to the server and is immediately available to all devices of the user via the respondent API. As a result, the MOTUS web and mobile applications can be used interchangeably.

Figure 42: MOTUS architecture



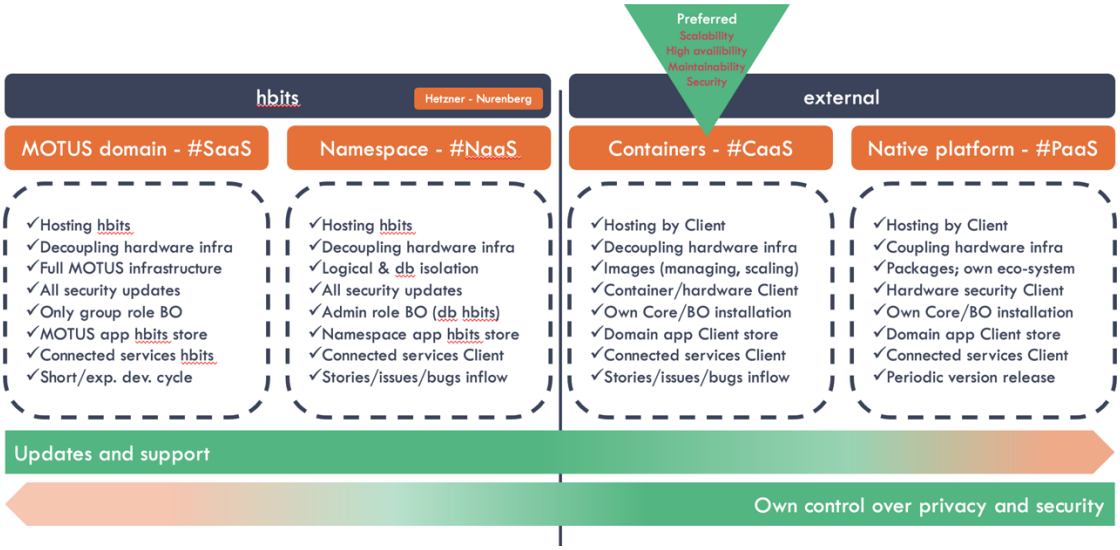
7.2. MOTUS deployment strategies

In designing deployment strategies for MOTUS, various models are considered based on hosting, infrastructure, security, and client control. These strategies fall under two main categories: hosting solutions under control of hbits (hardware is at Hetzner – Nuremberg and is ISO27001 certified) and hosting solutions where the hosting is external to the control of hbits (the adopting institution, organisation or Client). For each category two options are available, which brings the number of options to four in total.

Each model offers distinct advantages based on the level of control, scalability, and privacy required by the client. By offering a range of deployment models, organizations, institutions and clients can select the strategy that best aligns with their infrastructure, security, and operational needs.

The models are discussed below, while Figure 43 provides visual support. More tailored version based on these four models are possible.

Figure 43: MOTUS deployment strategies



7.2.1. MOTUS Domain - #SaaS

MOTUS domain operates under the Software as a Service (SaaS) model, meaning that all infrastructure, hosting, and updates are managed by hbits. This approach ensures seamless integration with the full MOTUS infrastructure while minimizing client-side responsibilities. Additionally, MOTUS provides automatic security updates, ensuring a high level of protection against vulnerabilities.

Access to the back-office is defined on the user level and is initiated by the platform administrator upon invitation to create an account and to activate the 2FA-login procedure. A non-admin user can only be part of one group at the time. Per group different studies can be developed. Per study back-office users can be given a role. With access restricted to groups and studies, security and role management are streamlined to maintain controlled access.

The MOTUS applications are available via the hbits domain or via the hbits application store, simplifying deployment and access. Furthermore, all connected services are managed directly by hbits, leading to a highly efficient and cohesive ecosystem. One of the biggest advantages of the MOTUS domain model is its short and expedited development cycle, making it ideal for organizations that require rapid iterations and continuous improvements without the complexities of self-managed infrastructure.

7.2.2. Namespace - #NaaS

Namespace follows the Namespace as a Service (NaaS) model, providing a structured approach where hosting and infrastructure are managed by hbits, but with logical and database isolation for each organization, institution or client. This approach ensures that different users have their own separate environments while still benefiting from the shared infrastructure managed by hbits.

One of the key differences from the MOTUS domain is that the back-office environment is separated from other back-offices (copies) and is managed by a Namespace administrator giving organizations, institutions and clients more control over their data while still receiving security updates and system

maintenance from hbits. Another added value is the strict separation of the database(s). Only role limited and technical access by hbits to the database remains.

The namespace applications can become available via the namespace domain or via the namespace application store while it remains an option to let hbits provide support and so to facilitate to a controlled yet flexible ecosystem. Additionally, connected services are configured by the organization, institution or client, allowing greater customization and integration with their workflows. Namespace deployment is well-suited for organizations, institutions or clients that require more administrative control and are in need of an own database while still leveraging cloud-based infrastructure.

7.2.3. Containers - #CaaS

Containers is seen as a Containers as a Service (CaaS) model and is designed for organizations, institutions and clients who want full control over their infrastructure while still leveraging containerized environments for scalability and manageability. Under this model, hosting and infrastructure are entirely managed by the client, allowing for complete autonomy over hardware and software configurations.

One of the primary advantages of this approach is containerized management (orchestration), which facilitates easier scaling, deployment, and resource allocation. Clients are responsible for the installation and maintenance of their core and back-office systems, giving them greater flexibility but also increasing their operational responsibilities.

Applications are stored and managed within the organization's, institutions or client's domain or store, ensuring independence from third-party hosting services. Connected services are also fully controlled by the client, making this model an excellent choice for organizations that prioritize customization, data sovereignty, and independent issue management.

7.2.4. Native platform - #PaaS

Native platform is seen as a typical Platform as a Service (PaaS) model that represents another deployment strategy, where organizations, institutions and clients manage their hosting and infrastructure entirely. Unlike the previous container-based solution, a native platform runs the MOTUS software directly on the operating system of dedicated hardware.

With a native platform clients operate within their own ecosystem and can receive software packages for updating their environment, creating a tailored environment that aligns with their specific requirements. Core and back-office installation is handled exclusively by the organisation, institution or client, offering complete control but requiring more internal IT resources. Applications are handled via an own domain and or an own store, and connected services are fully managed in-house. One defining characteristic of native platform is its periodic version release cycle.

7.2.5. Advise from hbits

A key aspect of deployment strategy is balancing updates and support with control over privacy and security. The choice between the different options depends on the level of autonomy and security requirements of the organization, institution and client.

In general, there are several critical attributes to be taken into account. These attributes guide the selection of deployment models to meet organizational needs while ensuring stability and performance, and are:

- Scalability: The ability to handle growing workloads and increasing demand efficiently.
- High Availability: Ensuring minimal downtime and continuous service operation.
- Maintainability: Simplified system updates and ease of managing infrastructure.
- Security: Robust protection mechanisms to safeguard data and applications against threats.

Although MOTUS can be deployed in various ways, the advice to NSIs is to follow a cloud-based deployment strategy in which all MOTUS components (core, back-office...) and microservices as well run in their own container. The benefits of a containerized environment are better scalability, improved application monitoring and decoupling from the underlying infrastructure. Several container management and container orchestration applications exist from commercial to open-source solutions, from simple to complex orchestration platforms. Examples are: docker compose, kubernetes, rancher, redhat openshift.

7.3. MOTUS platform pentest(s)

The MOTUS platform has undergone various pentests since the platform is in use for different studies by hbits and by the VUB (as hbits is a spin-off). The most recent pentests were conducted together with Destatis carried out by external consultants in the area of cyber and application security like T-Systems, Atos and SEC Consult. In total 3 pentests were conducted with each time some retests after the findings were resolved. These pentests were done on a native installation of MOTUS on the premisses of IT.NRW, as being the host to install MOTUS with their own eco system.

The first part will discuss the approach and findings of the native installation with the focus on the last pentest, and with MOTUS running on Laravel 10. The second part has a focus on MOTUS being deployed as a containerised solution, and with MOTUS running on Laravel 11. A third part will address the Load and Performance (LaP) test and is carried out via the open-source tool K6. The findings are independent from the deployment strategy.

7.3.1. Pentest native installation MOTUS

The last pentest on the (still running) native installation of MOTUS with IT.NRW as host was carried out by SEC Consult. The pentest was conducted during the period Q4 2023 – Q1 2024 and mitigations were finalised Q2 2024. The previous pentests were conducted between 2020 and 2023. First an short overview is given about these tests before continuing to the most recent test.

Short summary of the 2020-2023 pentests

The pentests that were conducted over the years 2020-2023 by T-Systems and Atos resulted in many findings being classified as Critical, High, Medium and Low. The underlying platform for the web applications was Kosever, for the mobile application Ionic and Angular. The pentest was also accompanied with a Source Code Analysis of MOTUS. At that point in time MOTUS was used solely

for TUS. A broad overview of findings is provided in the CRŒSS report[5] (B5460-2020-INNOVTOOLS-HBS-TUS).

The short summary is as follows - For the web application, critical vulnerabilities such as command injection were found and subsequently resolved. Several high-severity stored and reflected cross-site scripting (XSS) issues were also identified and closed. Cross-site request forgery (CSRF), weak password policies, and session fixation were marked as approved after mitigation. Medium and low-severity issues like cacheable responses, outdated libraries, and denial of service risks were either resolved or approved after review.

For the mobile application, most identified vulnerabilities, including missing certificate pinning, XSS in survey answers, and information disclosure in HTTP responses, were remediated. However, the missing root/jailbreak detection remained unresolved with a medium rating, as it was deemed a low-priority issue due to its limited impact on security compared to user participation concerns.

The static source code analysis for MOTUS, conducted using Fortify Static Code Analyzer, identified 1,992 security findings across (at that time) 245,131 lines of code. After reviewing false positives, 121 actual security issues remained. These included critical vulnerabilities like Cross-Site Scripting (XSS), Open Redirects, and Hardcoded Passwords, along with high and low-severity issues.

Following a mitigation period and consultation, all issues were either resolved, deemed irrelevant, or fixed, with the final status approved by Destatis.

Setup of the 2023-2024 pentest

The pentest was carried out by SEC Consult, with Laravel 10 used for the web applications and (besides version upgrades unchanged) Ionic and Angular as underlying platforms. Of importance is that MOTUS grow from a TUS to a cross-domain platform also including HBS.

The security assessment of MOTUS aimed to identify (new) vulnerabilities that could impact confidentiality, integrity, and availability. The previous pentest results were taken into balance. The assessment covered the Android and iOS mobile applications, the Front Office Web App and API, and the Back Office Web App and API.

The penetration test was carried out in accordance with the OWASP Web Security Testing Guide (WSTG) and the OWASP Mobile Security Testing Guide (MASTG) to ensure thorough coverage. Vulnerability severity was evaluated using the CVSS framework (https://www.first.org/cvss/). Additionally, each identified vulnerability includes a subjective assessment from the tester ("tester rating"), reflecting their expertise and experience. The following test categories were included:

For the web application testing having included the back-office and front-office web applications:

- Evaluated authentication, authorization, and session management mechanisms.
- Verified data encryption during transport and checked certificate validity.
- Tested for unauthorized access to restricted areas and inadequate authorization controls.

[5] B5460-2020-INNOVTOOLS-HBS-TUS. Minnen, J., Olsen, J., & Sabbe, K. (2022). CRŒSS: Establishing a Cross-domain data collection platform for the ESS (European Statistical System). Statistics Belgium, Destatis, hbits CV and Vrije Universiteit Brussel.

- Analyzed platform configuration, backup files, outdated software, and HTTP security headers.
- Examined user input handling to detect injection vulnerabilities and denial-of-service risks.
- Checked for weaknesses in business logic and process flows.
- Conducted client-side testing, including cross-origin sharing and error handling for information leaks.

For the web application testing having included the front-office mobile Applications (iOS & Android):

- Assessed data storage security and interactions with third-party services.
- Verified cryptographic mechanisms, ensuring proper key management and avoiding deprecated algorithms.
- Tested authentication mechanisms, including two-factor authentication and stateless authentication.
- Examined session management for improper rights assignment and session takeover risks.
- Evaluated network communication security, testing against MITM attacks and TLS settings.
- Reviewed code quality, debugging configurations, and reverse engineering protections.

Findings of the 2023-2024 pentest

A total of six vulnerabilities were identified, categorized based on their severity. The overall system risk was assessed as Medium, primarily due to insecure programming practices.

The key vulnerabilities and risks are according to the OWASP classification (https://owasp.org):

- Broken Access Control (F-3.1)
    - Users could access and modify other users' data by manipulating API requests.
    - This flaw affects both web and mobile applications.

  Risk: Unauthorized access to sensitive user data, leading to potential data breaches.
- Logout Does Not Terminate User Session (F-3.2)
    - User sessions remain active even after logging out.
    - Attackers with access to session IDs can hijack accounts.

  Risk: Persistent unauthorized access even after user-initiated logout.
- User Credentials Saved After Application Is Closed (F-2.1, iOS App)
    - Credentials persist in the login mask upon app closure, enabling potential theft.

  Risk: Account takeover if an attacker gains access to the device.
- User Enumeration in Password Reset (F-3.3, F-4.1)
    - The application's error messages reveal whether a user account exists.

  Risk: Attackers can compile valid usernames for brute-force attacks.
- Weak Password Policy (F-4.2)
    - Only an 8-character minimum length is enforced.
    - No checks for commonly used weak passwords.

  Risk: Increased likelihood of successful password-guessing attacks.

All vulnerabilities were solved besides the User Enumeration finding where Destatis deemed it too burdensome to the respondents in case the listed issue would have been solved. Eg. in case a respondent would like to reset his/her password the respondent has to provide a username. In case the username is known by MOTUS a password reset request is send to the linked email address. In case a username does not exist, no email can be sent. The fact that there is no email send could inform attackers that a certain username does not exist.

7.3.2. Pentest containerized installation MOTUS

Over the past year, hbits has transitioned to a fully containerized environment. Scalability, high availability, maintainability, and security were key considerations in shaping the deployment strategy.

In June 2025 Destatis will be shifted from a native installation to a containerized solution. This transition is driven by the same factors but also aims to better accommodate Smart Data.

Smart Data is collected and processed externally from MOTUS through microservices. These microservices generate outputs that are integrated into the MOTUS platform via Adaptor APIs. From there, information flows to front-office applications through internal APIs, with potential bidirectional communication. These microservices leverage the generic architecture developed within SSI and are deployed in a containerized environment. By this the end-to-end flow now incorporates Smart data. The first aim for Destatis is to include the Receipt Scanning Microservice (OCR and COICOP classification).

Below the testing plan is being described. Results are expected in May 2025.

Objectives

The goal of this penetration test is to identify and mitigate security vulnerabilities introduced by:

- The transition to a containerized environment.
- The upgrade from Laravel 10 to Laravel 11.

This penetration testing plan aims to ensure that the MOTUS platform remains secure after its transition to containerization and Laravel 11. Testing will be conducted periodically and upon major updates to maintain security compliance.

Scope

During the pentest different layers/components are tested

- Application Layer: Laravel-based MOTUS application.
- Containerization Layer: Docker, Kubernetes, or any container orchestration setup.
- Network Layer: API communication, internal and external network configurations.
- Infrastructure & Deployment: CI/CD pipelines, environment security, logging, and monitoring.

Methodology

The penetration test will follow the OWASP Testing Guide and the MITRE ATT&CK framework.

Reconnaissance

- Enumerate exposed services, endpoints, and technologies used.
- Identify third-party dependencies and integrations.
- Review Laravel 11’s breaking changes for potential vulnerabilities.

Threat Modeling

- Map attack vectors related to containerization (e.g., container escape, misconfigurations).

- Identify risks associated with Laravel framework changes, and most importantly new or modified libraries.
- Assess security of API authentication and authorization mechanisms.

Testing Areas

Application Security (Laravel 11-Specific)

- Authentication & Authorization: Verify role-based access controls, API tokens, OAuth configurations.
- Validation & Input Handling: Test for SQL injection, XSS, CSRF vulnerabilities.
- Session Management: Ensure proper handling of user sessions and cookies.
- Error Handling & Logging: Check for exposure of sensitive error messages.

Container Security

- Image Vulnerabilities: Scan container images for CVEs (using Trivy, Clair, etc.).
- Misconfigurations: Ensure minimal permissions, non-root user execution, and proper file system restrictions.
- Secrets Management: Verify that environment variables and secrets are stored securely.
- Networking & Isolation: Ensure proper segmentation between containers and prevent lateral movement.

API & Network Security

- API Security: Test for broken authentication, improper authorization, and data exposure.
- Rate Limiting & DoS Protection: Assess rate limiting measures.
- Man-in-the-Middle (MITM) Attacks: Ensure HTTPS enforcement and certificate pinning where applicable.
- Internal API Calls: Test inter-container API communications for security risks.

Infrastructure & CI/CD Security

- CI/CD Pipeline Security: Review for exposed credentials, security misconfigurations.
- Deployment Security: Ensure secure use of environment variables and secrets management.
- Logging & Monitoring: Verify that security-relevant logs are being captured and monitored.

Tools & Techniques

Category	Tools Used
Reconnaissance	Nmap, Nikto, Sublist3r
Web/App Security	OWASP ZAP, Burp Suite, sqlmap
API Security	Postman, JWT.io, Insomnia
Container Security	Trivy, Clair, kube-hunter
Infrastructure	OpenSCAP, Lynis, Falco

7.4. MOTUS Load and Performance test

Load and performance testing plays a crucial role in ensuring that a system functions efficiently under different conditions. These tests help identify bottlenecks, stability issues, and scalability limits, allowing developers to optimize their applications before deployment.

Load testing evaluates how MOTUS and the hosting behaves under both normal and peak usage conditions. By simulating multiple users accessing the system simultaneously, it helps measure critical performance metrics such as response time, throughput, and resource utilization. By this performance testing focuses on measuring the speed, responsiveness, and stability of a system. It assesses factors such as page load times, server response times, and error rates to ensure the application delivers a seamless user experience.

7.4.1. Scope in view of the SSI project

Two sorts of LaP tests are conducted. The first leans toward the participation journey of a respondent within a 'typical' TUS or HBS study with multiple task a respondent have to undertake. The second is linked to the Adaptor API that serves as the integrator of output data from the microservice to the core component (/container) of the MOTUS data collection platform.

Login phase

In a TUS or HBS study multiple phases are apparent in the participation journey of a respondent. The most 'stressfull' phase is however the login phase, certainly if thousands of respondents receive an invitation email (or letter) at the same moment/day. It should be expected that MOTUS and the hosting can handle at least 500 concurrent user logins without experiencing slowdowns or failures.

Adaptor API process request

In this test is the 'integration' process of passing output data from the microservice to the MOTUS environment important. The Adaptor API is therefore tested under different traffic loads to determine how quickly it can process requests and return responses.

This integration testing is accompanied by a scalability testing to assesses whether the microservice can efficiently handle increased workloads by adding more resources, such as additional container instances. This type of testing is essential to see eg. how many tickets or geo points the microservice can handle before it starts to fail.

Methodology

For this test the K6 tool was used. K6 is an open-source performance testing tool designed for load testing web applications, APIs, and microservices. It enables developers and testers to create scripts in JavaScript and execute them to evaluate system performance under various conditions. On top, K6 offers real-time insights while it is easy to integrate into CI/CD pipelines.

Load and Performance test results

The document is based on configurations and optimizations performed to the MOTUS system. Key configurations and settings that led to the final test setup include:

- Initial Setup & Baseline Tests
  - Pre-forking was tested to diagnose potential PHP-FPM issues.
  - Performance was compared using simple PHP files versus the full MOTUS system.
  - Tests were conducted with K6 and an inhouse tool of the company ZOTT, simulating different user loads.
  - HAProxy was used but later deactivated to check its impact.
- Server Configurations & Adjustments
  - Apache settings were tuned, increasing maxrequestworkers from 200 to 2,500.
  - Redis caching was tested and adjusted for better performance.
  - Load balancer configurations were changed and reverted based on impact.
- PHP-FPM & Database Adjustments
  - max_children settings were increased to handle more concurrent users.
  - Database load was analyzed to rule out bottlenecks.

Below an output table is provided.

Table 26: Performance results per type of phase for 500 and 1000 simultaneous users using MOTUS

	Numbers	Time after configuration(s)	Numbers	Time after configuration(s)
Visit start page	500	419 ms	1.000	652 ms
Login	500	6.727 ms	1.000	15.166 ms
Open questionnaire	500	151 ms	1.000	216 ms
Send 20 questions	500	421 ms	1.000	530 ms
Open HBS diary	500	680 ms	1.000	802 ms
Make 30 expenses	15.000	449 ms	30.000	456 ms
Send HBS diary	500	190 ms	1.000	189 ms
Logout	500	341 ms	1.000	342 ms
Send 20 questions	500	29 ms	1.000	37 ms

The results are:

- Final System Capability
  - The system stabilized at handling 2,500 requests and 200 concurrent logins without performance issues.
  - Higher loads did not cause system failure but required further optimizations.

Overall, the setup involved multiple iterations of configuration tuning and performance testing to optimize response times and handle increasing user loads efficiently.

8. CBS platform

This chapter outlines the architecture of the CBS data collection platform, designed to support scalable, shareable, and reusable services within a secure and robust environment. Leveraging a modular microservices approach, the platform enables independent scaling, dynamic resource allocation, and seamless integration across mobile and web applications. The architecture ensures secure data handling, efficient processing pipelines, and flexible deployment strategies while supporting end-to-end workflows for data ingestion, OCR-based processing, and classification.

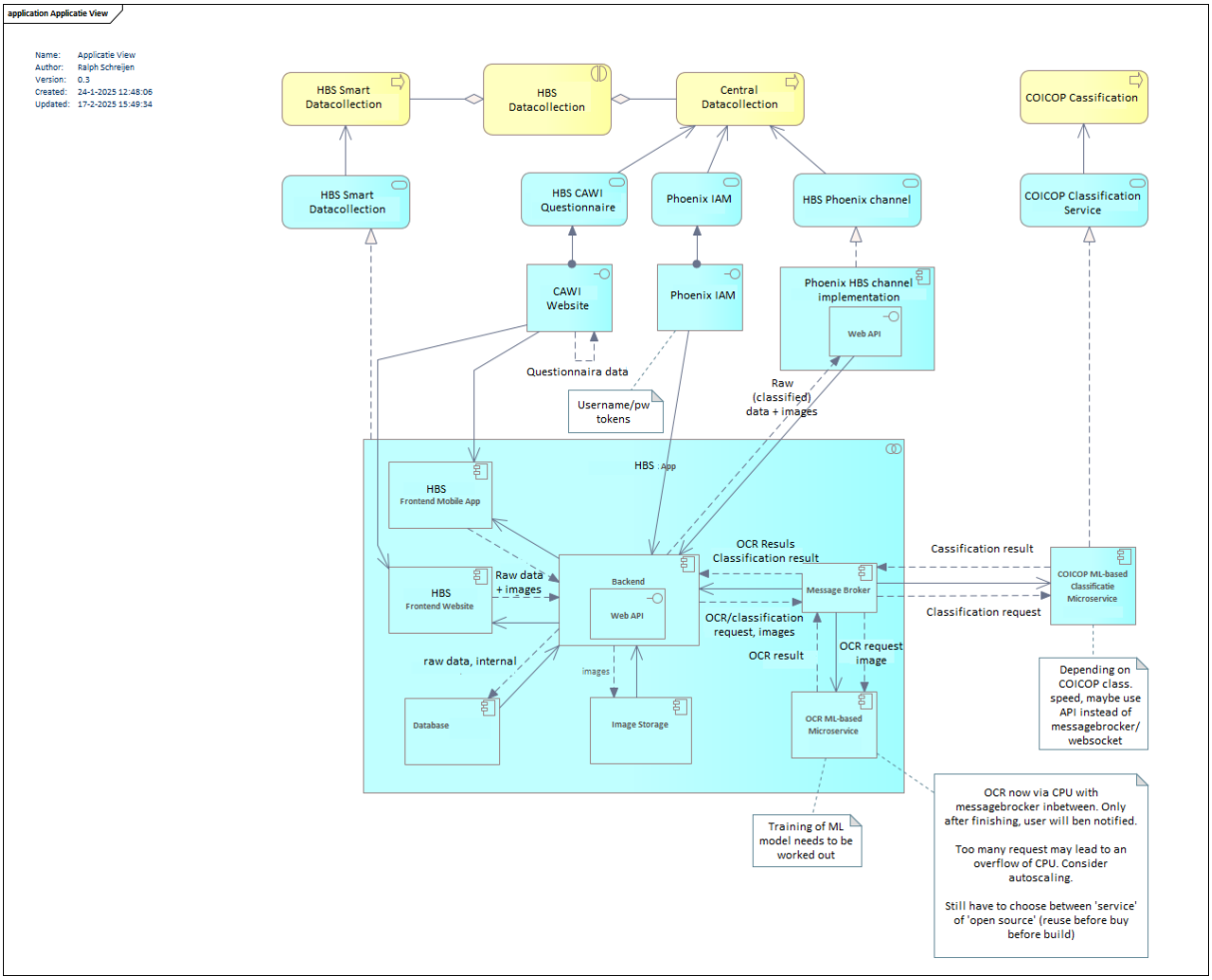
8.1. CBS architecture

The following architecture ensures scalability, shareability, and reusability while providing a robust and secure platform for data collection, processing, and classification.

- Scalability is inherent to a microservices architecture since each microservice can be scaled independently when required. Furthermore, these microservices are deployed in a distributed environment (K8s) which allows for dynamic allocation of resources. This also helps in ensuring robustness of the solution: if one service goes down, it does not necessarily affect other services.
- Shareability is another big advantages of using loosely coupled microservices. Each microservice contains fully independent code, logic and data thus making it far more shareable than application with for example a centralized database on which multiple services rely. When we talk about microservices we actually mean a collection of API's (not tied to any specific frameworks, programming languages or databases) which are designed to communicate to different API's. This independence makes for shareable code.
- Reusability: the total solution is broken up into small loosely coupled modules that provide specific functionality. E.g. the OCR service is a separate API module, the COICOP service likewise, each functionality within the app is a specific API. Etc. All these modules or API's can be deployed at will, independent of each other.

In the architecture process picture below, you can see that the OCR and classification service (SSI services) are only shown on the sideline (The end to end architecture shows part of the platform that the solution integrates with (Phoenix). The integral solution/platform has not been specifically named however. These will be developed, further trained and and seperately implemented later.

Figure 44: CBS data collection platform architecture



8.1.1. Frontend Applications

Data is collected through both an app as well as a web application. Both are similar in functionality but provide a totally different user experience while the backend synchronizes so household will be able to use the webapp and the mobile app simultaneously.

Mobile App (HBS App)

- Manual Data Input: Users can manually enter data (e.g., expenses, categories) through a daily diary.
- Receipt Scanning: Users can scan receipts using the device camera. The scanned images are temporarily stored locally and then uploaded to the backend (this makes it possible to use the app offline and synchronize when a network connection is reestablished).
- Authentication: Respondents log in via OAuth2 or JWT to ensure secure access to the backend. The credentials provided can come from an external identity provider or from the mobile backend.

Website App

- Data Input: Users can manually input data through web forms, similar to the mobile app.
- Authentication: Uses the same authentication mechanism as the mobile app for consistency.
- Shared Backend: Connects to the same backend services as the mobile app to ensure data consistency and reusability.

8.1.2. Backend Services

API Gateway

- Entry Point: Acts as a single-entry point for all requests from both the mobile app and the website app.
- Routing: Routes requests to the appropriate microservices (e.g., data ingestion, OCR service, classifier service).
  Route: app → back-end → microservice → back-end → app
- Authentication & Authorization: Validates JWT tokens or OAuth2 credentials to ensure secure access.

Data Ingestion Service

- Receipt Upload: Accepts receipt images from the mobile & web app and stores them in a temporary on-premises storage). Since the data could contain sensitive information, we have opted to store this data in-house (also taking into account GDPR). However, switching to a cloud solution for receipt-storage would be fairly low-effort.
- Manual Data Input: Accepts structured data from both the mobile app and website app, validates it, and stores it in the database.
- Validating in this context means that (frontend) data presented does not violate the logical model at database level. E.g. a string cannot be mapped onto an integer etc. Sidenote: datamodels in the front end and the backend are identical so an unhappy flow (whereby validating goes wrong during ingestion) could only be the case if the data is manually manipulated. If this happens, the data is flagged and logged as error.
- Event Publishing: Publishes events (e.g., receipt_uploaded, manual_data_received) to a message broker (RabbitMQ) for asynchronous processing.

8.1.3. Machine Learning Services

OCR Service (ML-Based)

- Image Processing: Receives receipt images from the temporary storage via a message broker or direct API call.
- Text Extraction: Uses a pre-trained ML model to extract text from the receipts.
- Output Formatting: Converts the extracted text into a structured JSON format and publishes an event (receipt_processed) to the message broker.
- More information regarding the OCR microservice can be found in chapter 5.

COICOP Classifier Service

- Data Classification: Receives structured data from the OCR service (or manual input via the message broker).
- COICOP Classification: Uses a machine learning model or rule-based system to classify the data into COICOP categories. This is an ongoing process, and we still have some ways to go before the resulting output is satisfactory.
- Output Storage: Stores the classified data in the database and publishes an event (data_classified) for downstream services.

8.1.4. Data Storage

Relational Database (Microsoft SQL)

- Structured Data: Stores manually input data, OCR-processed data, and classified data.
- Relationships: Maintains relationships between users, receipts, and classified data for querying and reporting. The data model is based on the relation between users, devices,

receipts, products and images. The data is organized according to an Entity Relationship Diagram (ERD), which is managed by both the frontend and the backend (see previous comment on this topic). This relational model links users, devices, receipts, and products through primary and foreign keys.

8.1.5. Message Broker (RabbitMQ)

- Event-Driven Architecture: Facilitates asynchronous communication between services.
- Events:
    - receipt_uploaded: Triggered when a receipt is uploaded.
    - receipt_processed: Triggered when the OCR service completes processing.
    - data_classified: Triggered when the classifier service completes classification.
- Decoupling: Ensures services are decoupled and can scale independently.

8.1.6. Shared Services

- User Management Service
- Centralized Authentication: Manages user authentication and authorization for both the mobile app and website app. We also have the option to use an external Identity broker that is closely related to the process of data collection which is our implemented option for the time being.

8.1.7. Deployment & Infrastructure

Cloud Provider (Azure Devops)

- Scalability: Uses auto-scaling groups and load balancers to handle varying loads.
- Serverless Components: Uses serverless functions

Containerization (Kubernetes)

- Microservices: Each service (e.g., OCR Service, COICOP Classifier Service) is containerized for easy deployment and scaling.
- Orchestration: Kubernetes manages the deployment, scaling, and networking of containers.

8.1.8. Shareability & Reusability

Shared Libraries

- Common Utilities: Shared libraries for logging, authentication, and data validation are used across services.
- API Contracts: Well-defined API contracts ensure consistency between the mobile app, website app, and backend services.

Modular Design

- Microservices: Each service is designed to be independent and reusable (e.g., OCR Service can be reused in other projects).
- Event-Driven Architecture: Enables easy integration of new services or replacement of existing ones without disrupting the system.

API Documentation

Swagger/OpenAPI: Comprehensive API documentation ensures developers can easily integrate with the backend services.

8.1.9. Security

Data Encryption

- In Transit: Uses HTTPS/TLS for secure communication between clients and the backend.
- At Rest: Encrypts sensitive data (MSSQL databases to store data which provide built-in encryption features. We specifically use Column level encryption through the use of AES) stored in databases and storage systems.

Access Control

- Role-Based Access Control (RBAC): Ensures users and services have access only to the data and functions they need.

8.2. CBS deployment strategies

Deployment Strategy (Work in Progress). We are in the process of refining our deployment strategy to ensure a structured and automated approach for rolling out new features and updates while maintaining stability and reliability. Our goal is to implement a robust CI/CD pipeline that supports efficient testing, controlled rollouts, and seamless updates.

8.2.1. Code Commit & Versioning

- Developers push changes to a feature branch, following GitFlow principles.
- We are working on standardizing our code review process and integrating static code analysis.
- Once approved, changes are merged into the develop branch and prepared for testing.

8.2.2. Testing & Build

- Our CI pipeline triggers automated tests (unit, integration, and E2E), but we are exploring ways to enhance test coverage.
- If tests pass, the application is built and (if applicable) containerized for deployment.
- We are reviewing our build optimization strategy to speed up this process.

8.2.3. Staging Deployment

- Updates are deployed to a staging environment for further validation.
- We are refining our QA process to improve functional, regression, and performance testing.
- User Acceptance Testing (UAT) is being incorporated where necessary.

8.2.4. Approval & Release Candidate

- If staging tests pass, a release candidate is created and reviewed.
- We are working on improving our release approval process and defining clear release criteria.

8.2.5. Gradual Rollout & Monitoring

- Our goal is to implement a canary deployment strategy to roll out updates to a subset of users first.
- We are enhancing our monitoring capabilities to track performance and potential issues in real-time.

8.2.6. Full Deployment & App Store Release

- Once validated, the update is published to the app store.
- Users will have the flexibility to install the update at their convenience.

- Post-deployment monitoring is an area where we aim to improve incident response and rollback mechanisms.

8.3. CBS platform pentest

On March 31 and April 1, 2025, Secura conducted a security assessment of the mobile CBS Expenditure Application at the request of Statistics Netherlands (hereinafter referred to as CBS). The CBS Expenditure Application is crucial for CBS because users voluntarily input their expenditures through this app for the CBS Expenditure Survey. This survey helps determine how much more expensive life in the Netherlands becomes annually. Consequently, any compromise or disruption could lead to reputational damage for CBS.

The security assessment was carried out as a gray-box review of the mobile application, examining both the Android and iOS versions along with their associated infrastructure.

8.3.1. Summary of the results

The study aimed to address the main question posed by CBS: "Is user data sufficiently protected?" Throughout the investigation, Secura identified effective security measures but also areas for improvement.

One example of an effective measure observed during the review is that administrative functionalities, such as administrator registration, are disabled in the production environment.

At the same time, several points of concern were noted. Specifically, additional defense-in-depth measures are lacking. In terms of authentication, there was a lack of controls to prevent one user from impersonating another. A user with knowledge of another user's valid ID-code could immediately assume that identity and potentially undermine the reliability of the survey by submitting false data. While these IDs are not easily guessable, this remains a concern for further security enhancement.

Additionally, the investigation revealed opportunities to enhance input validation within the application. Special characters were used in store names and product descriptions, which, while not posing direct vulnerabilities within the app itself, could lead to issues in later processing phases where such data is handled.

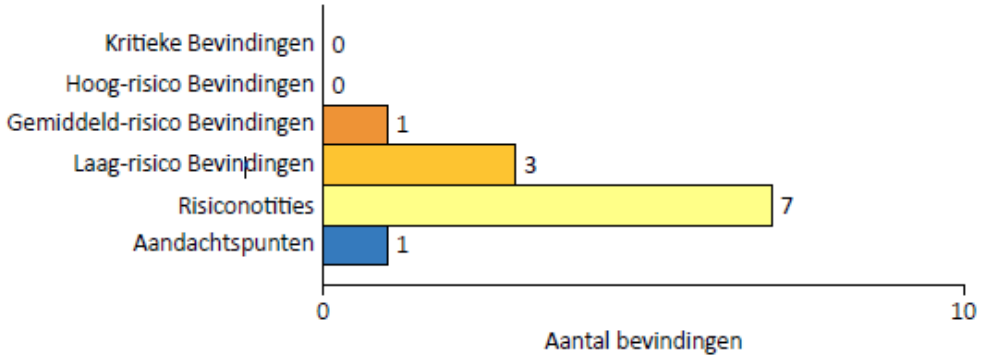
Improvements were also identified regarding system interaction. The application stores data unencrypted, posing a risk to sensitive information confidentiality. Moreover, it was discovered that logging features inadvertently provide access to authentication materials, increasing the risk of unauthorized access.

Secura notes that both the mobile app and backend/web application are still actively under development. For instance, some functionality shown in the mobile app—such as receipt scanning via OCR technology—is yet to be implemented. If further functionalities or modifications are made before the expenditure survey commences, it may warrant an additional test on these.

8.3.2. Risk Management Findings

Secura has classified each finding according to the risk profile that emerges for CBS. The following chart displays the number of findings per risk category:

Figure 45: Risk Management Findings per category (in Dutch)



8.4. CBS platform Load and Performance test

No Load and Performance tests were conducted by the CBS.

Annex 1: COICOP Classification

See .pdf Annex_WP3 SSI_COICOP.

SSI Annex: ML Experiments

Chris Lam, Tim de Jong, Marco Puts

CBS Heerlen
CBS-weg 11
6412 EX Heerlen
P.O. Box 4481
6401 CZ Heerlen
+31 45 570 60 00

www.cbs.nl

31 oktober 2024

1. Introduction

The objective of Work Package 3 in Smart Survey Implementation (SSI-WP3) was to explore two approaches for the automatic classification of product descriptions from receipts to product categories. The first is by using Machine Learning (ML) and the second is by using (a combination of) string-matching techniques. The focus of this annex is on the former: we explore the capabilities and limitation of the ML approach.

The problem is to classify textual product descriptions on receipts to their appropriate product category. We are specifically concerned about the classification according to the COICOP standard 2018 (Classification of Individual Consumption According to Purpose). An example here is the text “milk” that should be assigned the 5-digit code “011410” indicating category “Raw and whole Milk”. We emphasize that the unit of classification is the individual product texts and not whole receipts which may contain multiple products. For ease of reading, we will use the term “receipt text” to specifically refer to the products descriptions on receipts in the remainder of the text. The need for such classification can be seen in statistics like the household expenditure, where spending is analyzed by product category. While the COICOP standard contains categories for a wide variety of consumer products, we focus our problem to the classification of receipt texts from supermarkets exclusively. The receipt texts primarily consist of food products.

ML offers several advantages over traditional string-matching techniques. Compared to these, ML does not require word-by-word matching and can achieve similar performance while being more computationally efficient. It can further explore patterns in the data allowing interpolation between similar texts whereas string matching primarily relies on matching through (approximate) character-level matching. In addition, some ML techniques can be trained to classify through semantic meanings which makes them more robust for synonyms and abbreviations. ML also allows us to use transfer learning, the act of further training a model that was previously trained on other data. This theoretically allows models to improve classification performances as it has access to information otherwise not available from training. ML can further utilize other types of features such as characteristics of the consumer (e.g. gender, age, household composition) for prediction.

When assessing ML in the context of classifying receipt texts, we are primarily interested in two criteria. The first criterion is that its performance should remain stable over time, meaning that its overall ability to classify should not deteriorate as new products or receipt texts are introduced. The second criterion is the ability to classify on receipt texts from unseen supermarkets. Since we are not able to get our hands on the data of all supermarkets in the country, it is important to assess the scope and limitations of the classifiers.

To simulate the relevant conditions and assumptions for testing these criteria, we evaluate ML by various train-test splits of the data. The first way of splitting the data is by time, where we train an ML model on data up to some date and then test it on periods after this date. The second way is to split the data by supermarket, where we simulate the scenario in which an ML model encounters supermarkets absent in the training set.

Before we can start evaluating various ML algorithms, however, we first evaluate the feature extractors. A feature extractor is a preprocessing component that converts textual data into a

numerical representation. Such component is necessary because ML algorithms are typically designed to operate on numerical input data only. Not all feature extractors are equal however, as some are better at preserving and representing the original data than others. This therefore highlights the need to first find the most suited feature extractor for representing receipt data, of which the best performing one will be used in the benchmark for the ML algorithms. All feature extractors will also be assessed on the two types of train-test splits mentioned above.

The research questions that drive the experiments are as follows:

1. What feature extraction technique is most suited for receipt texts?
2. How well can ML models classify receipt texts into COICOP product categories?
    a. How well can a ML classifier maintain a stable performance over time?
    b. How well can a ML classifier classify receipt texts from unseen stores?

The data set that will used in the experiments consists of receipt texts of three major supermarket chains in the Netherlands: Albert Heijn (AH), Lidl, and Plus. All data were provided by the supermarkets themselves except for the COICOP-labels necessary for training, these were sourced from an internal department at Statistics Netherlands.

This project is related to the Household Budget Research at Statistics Netherlands, in which a new smart survey application is developed to accommodate a new ways of data collection for the Household Budget Survey. Its development ran in parallel with the SSI WP3-project and had overlapping themes and objectives.

The remainder of this annex is structured as follows. First are the Methods in Section 2, where we first describe the data set in detail, the ML components featured in the benchmarks, the train-test splits, and finally the procedure in which we assess the benchmarks. Next, we present the results of the benchmarks in Section 3, and then the conclusion and discussion in Section 4.

2. Methods

Two benchmarks will be evaluated as mentioned in the introduction: one for the feature extractors, the other for ML algorithms. Both benchmarks will be performed using the same data set, which is described in Section 2.1. Section 2.2 outlines the list of feature extractors and ML algorithms included in the benchmarks. More detail on the train-test splits for benchmarking will be performed are described in Section 2.3. Finally, we will describe the assessment procedure in Section 2.4.

2.1 Data

2.1.1 Raw data

The data used for the benchmarks consists of the product inventories from three supermarkets in the Netherlands: LIDL, Plus, and AH (Albert Heijn). The data is organized by week, showing all the products offered by the supermarkets each week. This structure enables us to track which products were introduced or discontinued over time.

The data set has been supplemented with COICOP 1999 label data, obtained from the department for CPI (Customer Price Index) at Statistics Netherlands. This is indeed an older standard than the intended classification standard of COICOP 2018. Consequently, all evaluations were done with the COICOP 1999.

The two relevant features for the ML evaluations are the receipt texts and the COICOP-labels. A description and an example of the features are shown in Table 1. A COICOP code consists of six numbers, with each digit corresponding to a hierarchical level (also called COICOP-level). The first level is an exception of this rule by consisting of two digits. As an example, the product labeled with code 011410 indicates that it belongs to category “01: food and non-alcoholic drinks” in COICOP level 1, and in “011: food” looking at the proceeding digit.

Table 2 gives a full overview of the number of records, unique products and receipts of the raw data. The data consists mainly of food and non-alcoholic drink products (65% of the data), followed by the label “CPI Internal Label” (15%). These were marked by the CPI-group indicating either that they still needed labeling or that they were irrelevant for their own research. We emphasize that although CPI marks them irrelevant, they might be relevant for another research. All other categories form just 20% of the remaining products. In the same table, we see that the number of receipt texts is considerably lower than the number of unique products. This means that there are many products sharing the same receipt text.

Table 3 shows the number of records in the data set as by COICOP category. Being a disproportionally large category, “01: Food and non-alcoholic drinks” also consists of many more subcategories, with as many as 58 subcategories in COICOP level 5 compared to others. While the table shows the COICOP distribution for the whole data set, we also note that the distribution differs significantly among the three stores.

In comparison with the receipt data set used by DeStatis, similarities include that both data sets are limited to products from domestic supermarkets, which excludes foreign supermarkets and other types of stores such as DIY or clothing stores. Differences include the fact that DeStatis has an additional set of real receipts for testing aside from the CPI-data. These were scanned

using OCR (Optical Character Recognition) and were then manually labeled with the COICOP-category. Having a set of real receipts allows better testing against OCR errors and other real-world issues.

Feature Name	Description	Example
Receipt Text	Used as input feature / independent variable. The maximum number of characters of a receipt texts differs per supermarket.	Milk
COICOP Label	Used as classification label / dependent variable. According to the COICOP 1999 categorization standard.	011410

Table 1: The features used in the ML benchmarks.

Store Name	Number of records	Number of unique products	Number of unique receipt texts
Plus	7.119.674	342.340	204.840
Albert Heijn (AH)	1.214.392	66.578	13.559
Albert Heijn Franchise	1.866.896	62.715	12.737
Lidl	413.280	31.748	31.748
Total	**10.792.125**	**445.047**	**248.279**

Table 2: Number of records, product and receipt texts in the data set, broken down by supermarket.

COICOP Level 1 Category	Number of records	Proportion of whole (rounded)	Number of COICOP Level 5 categories
01: Food and Non-Alcohol Beverages	7.044.019	65%	58
99: CPI Internal label	1.638.121	15%	1
09: Recreation and Culture	703.551	7%	16
02: Alcoholic Beverages, Tobacco, and Narcotics	552.531	5%	12
05: Furnishings Household Equipment and Routine Household Maintenance	352.381	3%	12
12: Personal Care	247.260	2%	4
03: Clothing and Footwear	92.355	1%	8
06: Health	88.214	1%	3
11: Restaurant and accommodation services	67.754	1%	1
08: Communication	5.939	<1%	2
Total	**10.792.125**	**100%**	**117**

Table 3: Number of records per category in COICOP Level 1.

2.1.2 Preprocessing steps

The raw data described in the previous section were subject to the following preprocessing steps:

1. Remove instances if their receipt text entry is missing.
2. Remove instances where the receipt text consists of only one character.
3. Remove instances with COICOP label 999999. (indicates that the product is irrelevant for the internal department at Statistics Netherlands).
4. The records belonging to “AH Franchise” were merged into “AH”. AH Franchise includes the smaller, convenient store branding of the chain, such as AH To-Go.
5. Drop duplicates instances by receipt texts and COICOP-labels. This means that one instance with the exact same combination of receipt texts and COICOP-labels is kept, but instances may share the same receipt text as long as the COICOP-labels differ.

These steps are performed separately on the training and test sets after the train-test splits. This combined with Preprocessing Step 5, therefore means that some products in the training set may also occur in the test sets. This is particularly true if we split the data by some date, as the product inventory of a store before the period is likely to overlap with the inventory after it. Whereas overlap between training and test sets is normally avoided in ML, this was deliberately done for reasons which will be described in the sections about the train-test splits in Section 2.3.

Instances with label 99999 “CPI Internal Label” were also removed. The naïve reasoning behind this was that we believed that these products were “irrelevant”. Further, we misjudged that they could easily be filtered out beforehand. But this neglects new receipt texts with this label, which cannot be filtered out since they are unknown. Neglecting these receipts could heavily influence the evaluation results. In hindsight, the right way of testing was to include these instances at least in the test sets.

2.2 ML Components in the Benchmarks

2.2.1 Feature Extractor Benchmark

Table 4 lists all feature extractors considered in this benchmark. The feature extractors included can be categorized into two types. The first type of extractors is based on token frequencies, which means that they convert a receipt text consisting of tokens (words, or clusters of characters) into a numerical vector representing their occurrences. Tokenization on character-level is particularly relevant for receipt texts, as they typically contain few words and abbreviations. Each receipt text is tokenized into *character n-grams,* where the *n* in *n-gram* refers to the number of characters in each token. For example, the tokenization of the word “cheese” by *character 4-grams* would yield the following tokens: “chee”, “hees”, “eese”. N-gram tokenization can also be done on word-level, which means that the tokens consist of combinations of words, e.g. a word 2-gram tokenization of the text “hamburger cheese cheddar” yields the following tokens: “hamburger cheese”, “cheese cheddar”. Feature extractors with tokenizers on both word-level as well as character-level are included in the benchmark.

The second type of extractors are based on pretrained word-embeddings. Word-embeddings involve the clustering of a set of tokens in a high-dimensional space, where semantically similar tokens are clustered together, and dissimilar ones are placed further from each other. For example, consider the word-tokens cheese, milk, and lettuce. Since cheese and milk are dairy products, they will be closer to each other in the high-dimensional space than with lettuce. They are therefore semantically more similar to each other. When used for receipt texts, the feature extractors convert the texts into high-dimensional vectors, where similar products should be mapped close to each other.

Since it takes a considerable amount of data and processing power to train word-embeddings from scratch, we have only included pre-trained embeddings downloaded available online. Although it is possible to further fine-tune the embeddings by "re-training" or calibrating them on our data set, we were unfortunately not able to do so due to a lack of computational resources.

Since the performances of the feature extractors cannot be evaluated in isolation, i.e. without an ML algorithm, each feature extractor will be evaluated using the same classifier: the SGD-variant of the Logistic Regressor (SGDClassifier in Scikit-Learn). We use this algorithm because it handles the full data set in batches, therefore bypassing the issues above. The best performing feature extractor in this benchmark will be used on all candidates in the ML algorithm benchmark. These are shown in Appendix A.

Feature Extractor	**Description**
Count Vectorizer (Word) Count Vectorizer (Character n-grams)	Based on token frequency.
Hashing Vectorizer (Word) Hashing Vectorizer (Character n-grams)	Based on token frequency. Approximation of the count vectorizer using the hashing. It provides similar performance but has faster fit and transformation time.
Tfidf Vectorizer (Word) Tfidf Vectorizer (Character n-grams)	Based on token frequency. Fundamentally the same as the count vectorizer, but with normalization of the tokens by relative occurrence on the whole data set.
Spacy Model (BeRT)	Based on word-embeddings. We used the pretrained nl_core_news Spacy models in sizes small, medium, and large[1]. These are only evaluated by out-of-box performance. They are therefore not finetuned on the data.

Table 4: The feature extractors included in the feature extractor benchmark.

2.2.2 ML Algorithm Benchmark

Table 5 lists the candidates for the ML algorithms benchmark. We have tried to include a wide variety of ML algorithms ranging from simple regression models to decision tree-based ones. Some non-ML algorithms were included in the list for comparison: the exact string-matching algorithm, the constant-predictor and the string-matching pipeline developed by DeStatis. The pipeline by DeStatis is a string-matching pipeline involving a combination of exact and fuzzy

[1] https://spacy.io/models/nl

string-matching techniques. We have included two variants of this pipeline: one including the procedure that matches products based on article IDs and one without. The procedure was included in the original implementation, but provides an unfair advantage over other algorithms by using the article ID as an auxiliary variable.

Not all candidates in Table 5 are trained on the same amount of data. These algorithms are marked with an asterisk and are trained on a random sample of 100.000 instances from the training data rather than the full data set. This due to computational constraints of the Scikit-Learn library, where most algorithms were designed to handle around 100.000 instances during training.[i] Although it seems unfair to compare models trained on differing amounts of data, it does show the inherent strengths and limitations of the various ML algorithms. We have further fixed the randomization seeds to ensure similar random samples across algorithms.

The only ML-algorithm in the list trained on the full data set is the SGD-variant (Stochastic Gradient Descent) of Logistic Regression (SGDClassifier in Scikit-Learn). Since it fits models in batches of a data set as opposed to the entire set at once, it bypasses the memory-related issues.

Like in the feature extractor benchmark, all ML algorithms here will be evaluated with the same feature extractor. The feature extractor used here is determined by results of the feature extraction benchmarks.

ML Algorithms	**Description**
Logistic Regression*	Fit on only 100.000 instances.
Logistic Regression SGD (Stochastic Gradient Descent)	Can handle the full data set.
Naïve Bayes (Multinomial)*	Fit on only 100.000 instances. Classifies entries based on the prior distribution in the training data.
Random Forest Classifier *	Builds an ensemble of decision trees, where each is fitted on a random sample of a max. 100.000 instances.
Multilayer Perceptron Classifier*	Fit on only 100.000 instances.
Constant Predictor	Fits the prior of a class given an instance. Fit on only 100.000 instances.
Exact String-matching Algorithm	Can handle the full data set. Involves a simple look-up procedure in the training set: if there is a one-on-one match with a receipt text in the training set, the same COICOP-class will be assigned as the matching entry. Otherwise, the string-matching algorithm assigns a dummy class.
DeStatis String-Matching Pipeline	Uses the full data set as corpus. Features a combination of exact and fuzzy string-matching techniques, but also a procedure for matching products by article ID.
DeStatis String-Matching Pipeline (w/o article-ID matching)	Same as above, but skips the matching procedure by article ID.

Table 5: The list of algorithms considered in the benchmarks of ML algorithms. Some non-ML algorithms are also included.

2.3 Train-test Splits for Benchmark Assessment

The benchmarks will be assessed on two train-test splits: the split by time and a split by supermarket. The former tests ML on the degree of performance stability over time; and the latter on the ability to deal with receipt texts from previously unseen supermarkets.

2.3.1 Train-test split by Date: performance stability over time

Supermarkets perpetually introduce new products. Consequently, new receipt texts are also continuously being created. One important criterion that a ML model must possess is therefore robustness against this changing environment. We can simulate this scenario by splitting the data on some period *t*, where all receipt texts occurring before period *t* will be used for training, and the receipt texts after this period will be used for testing. The subsequent periods *t+1, t+2 …* will be tested on individually to track the performance over time. If a model performs well on all periods, it can be said that the model shows stable performance. If this is not the case, then we conclude that the model performance is not robust over time.

The period we have chosen for the split was June 1st, 2023 – all receipt texts before this date formed the training set and the data from periods June, July, and August 2023 each form independent test sets. The split was chosen in such a way that all supermarkets are reasonably represented in all training and test sets. Table 6 shows the number of instances in the data sets after this split and the preprocessing procedure as described in Section 2.1.2. The table shows that the distribution by supermarket differs significantly in the sets, with AH occupying a larger share in the test sets. The sets contain more instances than the number of the unique receipts as they may contain duplicate records with matching receipt texts but different COICOP-label.

With this assessment procedure, we hold the following assumptions:

- Firstly, we test only on receipt texts from known supermarkets (AH, Lidl, Plus), i.e. the test sets do not include supermarkets not found in the training set. The results therefore indicate the performance stability for these supermarkets only – and not on supermarkets not encountered by the model.
- We allow an overlap of receipt texts between the training and test sets, meaning that some receipt texts may occur in both sets. This was a deliberate assumption despite going against ML convention – as we expect a reasonable number of products, like eggs or milk, to remain on the shelves all the time, therefore better reflecting the real-world scenario.

The degree of overlap between the test sets and training sets heavily determines the difficulty of the tests – the greater the overlap, the easier the test set. This is because an ML model would have already seen the instances in the training set, and a simple string-matching algorithm would also have sufficed. To investigate the difficulty of the test sets, we have listed this degree of overlap in Table 7. It shows that more than 80% of the receipt texts in the text sets are also found in the training set, indicating that the test sets are relatively easy. However, this overlap does decrease rapidly over time: from 86% in June to 81% in August. This means that as time passes, the difficulty also increases since more and more new unseen products are introduced. There are also differences across supermarkets, with Plus having a lower rate of overlap (< 80%) than AH and Lidl (both >90%), meaning that Plus is harder to classify than the others.

Supermarket	Training set (< June 2023)		Test set June 2023		Test set July 2023		Test set Aug. 2023	
	n	%	n	%	n	%	n	%
Plus	399.315	89	49.119	67	64.179	72	49.620	67
AH	21.189	5	19.632	27	20.177	23	20.265	27
Lidl	24.964	6	4.362	6	4.385	5	4.361	6
Total	445.468	100	73.113	100	88.741	100	74.248	100

Table 6: The number of unique receipt texts in the training and test sets and the breakdown per supermarket after preprocessing.

	Instances in Test set	Instances overlapping with training set	
	n	n	%
Test set Jun 2023			
Plus	49.119	43.429	79
AH	19.632	19.306	97
Lidl	4.362	4.125	95
Total	73.113	66.860	86
Test set Jul 2023			
Plus	64.179	55.747	95
AH	20.177	19.508	90
Lidl	4.385	3.961	77
Total	88.741	79.216	83
Test set Jul 2023			
Plus	49.620	42.370	94
AH	20.265	19.449	85
Lidl	4.361	3.698	76
Total	74.248	65.518	81

Table 7: The number and percentage of receipt texts in each test sets overlapping with the training set, i.e. the receipt texts that are also found in the training set.

2.3.2 Train-test split by supermarket: performance on unseen supermarkets

Testing how well ML can classify receipt texts from previously unseen supermarkets is very important with regards to the real-world usage. Especially since the nature of the receipt texts can differ significantly across supermarkets in terms of character length, and the types of products that are sold (de Jong, Lam, & Puts, 2024). If a ML model performs well on supermarkets not encountered during training, then it means that it can be used on a wide range of supermarket, therefore defining the scope on which it can be used.

To simulate the conditions in which an ML model is faced with receipt texts from previously unseen stores, we perform multiple train-test splits where repeatedly one supermarket is left out for testing and the remaining ones are used for training. Table 8 shows all three combinations of splits by supermarket. The ratio between the training and test sets can differ significantly by size differ per split due to Plus having an unproportionally large number of instances. Theoretically, this means that split 3 is the hardest since there is much less data to train from.

	Training set	Test set

Split nr.	Supermarkets	n	Supermarket	n
1	Lidl, Plus	476.946	AH	26.198
2	AH, Plus	477.425	Lidl	25.719
3	AH, Lidl	51.917	Plus	451.227

Table 8: All combinations of train and test sets after splitting by supermarket. The numbers shown are the results after preprocessing.

2.4 Assessment Procedure

All training and test sets in the splits above are preprocessed individually with the procedure described in Section 2.1.2.

We perform a grid search to find the best hyperparameters for the training set, where one month of receipt texts for the split by time. Similarly for the split by supermarket, one supermarket in the training set is held out as validation set. The log-loss score was used as optimization criterion.Once the best performing hyperparameter is found for each ML component, they are fitted on the full training set and then evaluated on the test sets by accuracy, recall, and precision, balanced accuracy, f1-score, and log-loss. All training and testing are done in COICOP level 5. Analyses on higher COICOP levels are done by aggregating the labels from level 5.

Quantification

Aside from analyzing the individual classifications, the quantifications are often more important with regards to the statistical purposes. Quantification refers to estimating the class distributions of an unlabeled test set (Forman, 2005). In practice, it entails classifying an unlabeled data set using an ML model, and then counting the number instances that was assigned to each class. Quantification is especially relevant if the main purpose is the statistical analysis on an aggregate level, which is the case for our current problem.

The COICOP classification scheme is hierarchical, meaning that analysis on various levels of aggregation is possible. With this purpose in mind, the first question is whether the quantifications done by ML are good enough. At which level does over- and underestimation start to become problematic? To determine this, we examine the quantifications made by the best ML algorithm in our benchmark on various COICOP levels. In our evaluations, we aggregate labels by first letting the best ML model classify a test set as usual on COICOP level 5 and then truncating the category codes to the desired level, e.g. aggregating from level 5 to levels 1 and 3 for code 012340 will result in codes 01 and 0123.

The levels of aggregation that we assess are level 1, level 3 (only the subcategories under 01: Food), and lastly the worst over- and underestimated in COICOP level 4. The test sets on which this analysis will be performed, will be the most challenging test sets for the two types of train-test splits described above. The August 2023 test set will be selected for the train-test split by time due to it being the “newest” of the test sets, and the Plus test set will be used for the train-test split by supermarket because of its comparatively large size.

We assess quantifications by comparing the estimated number for the classes with the actual number according to the ground truth. The further apart the numbers are, the worse the quantifications. To assist the analysis for this comparison, we introduce the comparability ratio (CR) metric (Harteloh, 2020). Translated to our COICOP classification problem, a CR is defined

as: the estimated number of a COICOP category (x) by the ML model divided by the number of the same COICOP category (x) according to the ground truth of the same test set:

$$CR(x) = \frac{Estimated\ number\ of\ x\ by\ ML\ model}{Number\ of\ x\ in\ ground\ truth}$$

The CR is a measure of misestimations by the ML model relative to the ground truth. A CR close to 1.00 means that the estimated number is close to the ground truth. The greater or lower the number, the sever the overestimation or underestimation by the ML model, respectively.

3. Results

In this section we first report the results of the benchmark for the Feature Extractors in Subsection 3.1 and then for the ML algorithms in Subsection 3.2. The best performing feature extractor in the former section will then be used in all evaluations of the benchmark in the latter section. The benchmark is first assessed on the performance stability over time and then on the performance on receipt texts from previously unseen stores. Both benchmarks will be evaluated on two types of train-test splits: first the splits by time and then by supermarket.

3.1 Feature Extractor Benchmark

3.1.1 Train-test split by time: Performance Stability over Time

Table 9 shows the accuracy scores from benchmarking the feature extraction techniques on test sets from three consecutive months. Being the accuracy scores on the whole test sets, the scores reflect the proportion of all receipt texts that were correctly classified. All feature extractors were trained and evaluated using the same classifier: the Stochastic Gradient Descent variant of Logistic Regression (SGDClassifier). Additional scores, such as the precision, can be found in Appendix B.

The frequency based-extractors (i.e. Tfidf, Count, and Hashing Vectorizers) perform best, of which the character n-grams ones perform better than the word-based ones. Overall, the scores do not differ much over the different test sets, suggesting that all feature extractors yield stable performances over the months even though the overlap between test sets is less than 50%. The best scoring feature extractor in this evaluation is the Tfidf vectorizer with character n-grams. The Spacy Model performs worst. This is, however, because it had not been finetuned on the data, making it an unfair comparison. Other metrics characterizing the results are included in Appendix .

Since the test sets has more receipt texts from some stores than the others (on average: Plus 70%, AH 25%, Lidl 5% as presented earlier) we further show the accuracy scores of each test set with respect to each store in Table 10. Here, we observe that receipt texts from AH are on average more difficult to classify. Character n-grams extractors seem to consistently (85% - 88%) yield better results for this supermarket than when word tokenizers are used (82% - 83%). This falls in line with the observation that AH has shorter receipt texts than the others (de Jong, Lam, & Puts, 2024), where the longest text for AH consists of 12 characters, while Lidl and Plus respectively have 35- and 20-character limits. Feature extractors using character n-grams can therefore significantly improve the scores on shorter receipt texts.

Feature Extractor	Jun 2023	Jul 2023	Aug 2023	Mean tests
Tfidf Vectorizer (Char. n-grams)	89,9	89,4	88,8	89,4 ± 0,6
Count Vectorizer (Char. n-grams)	88,7	88,2	87,6	88,2 ± 0,6
Hashing Vectorizer (Char. n-grams)	88,8	88,3	87,7	88,3 ± 0,6
Hashing Vectorizer (Word)	86,4	86,0	85,3	85,9 ± 0,6
Tfidf Vectorizer (Word)	86,7	86,1	85,5	86,1 ± 0,6
Count Vectorizer (Word)	86,2	85,7	85,1	85,7 ± 0,6
Spacy Model (BeRT)	37,6	37,0	37,3	37,3 ± 0.3

Table 9: The accuracy scores of the feature extractors on the train-test split by time. The test sets shown are June, July and August 2023. The column Mean tests show the mean and standard deviations (prefixed with the ± symbol) of one candidate over all three tests.

Supermarket	AH			Lidl			Plus		
Test sets (2023)	Jun	Jul	Aug	Jun	Jul	Aug	Jun	Jul	Aug
Tfidf Vectorizer (Char. n-grams)	88	87	87	91	91	89	91	90	90
Count Vectorizer (Char. n-grams)	86	85	85	90	89	88	90	89	89
Hashing Vectorizer (Char. n-grams)	86	85	85	90	89	88	90	89	89
Hashing Vectorizer (Word)	83	82	82	87	86	84	88	87	87
Tfidf Vectorizer (Word)	83	83	82	87	86	85	88	87	87
Count Vectorizer (Word)	82	82	82	87	86	85	88	87	86
Spacy Model (BeRT)	34	34	34	46	46	45	38	37	38

Table 10: : The accuracy scores of the feature extractors per supermarket on the test sets June, July and August 2023.

3.1.2 Train-test split by supermarket: Performance on Unseen Supermarkets

Table 11 shows the accuracy scores of the feature extractors when tested according to the train-test splits by supermarket. Again, other metrics of the same tests can be found in Appendix B. Like in the previous subsection, all feature extractors were trained and evaluated with the same classifier: the Stochastic Gradient Descent variant of Logistic Regression (SGDClassifier). We first notice that the scores overall are low, with none of the feature extractors scoring over 50%. But the ones based on character n-gram tokens perform better than the ones based on word tokens: both in raw accuracy scores (46% - 48% for character n-grams tokens compared to 43% - 44% for word tokens) and the standard deviation over the various supermarkets (2% versus 4%). The winner in this assessment is also the Tfidf-vectorizer with character n-grams.

Generally, the results show that Plus was harder to classify than the other supermarkets. This is not surprising considering the great imbalance in size of the training and test data sets. The feature extractors had substantially less data to train on and more data that had to be classified, making the evaluation of Plus a difficult task. Meanwhile, the evaluations on the other supermarkets allowed models to train on the large data set of Plus while having a relatively small (and therefore easier) test set.

Feature Extractor	Split 1: AH (n=26.198)	Split 2: Lidl (n=25.719)	Split 3: Plus (n=455.227)	Mean tests
Tfidf Vectorizer (Character n-grams)	50,0	49,7	45,5	48,4 ± 2,5
Hashing Vectorizer (Character n-grams)	48,4	49,3	45,0	47,6 ± 2,2
Count Vectorizer (Character n-grams)	46,7	46,5	43,6	45,6 ± 1,7
Tfidf Vectorizer (Word)	47,7	44,8	39,2	43,2 ± 4,0
Count Vectorizer (Word)	46,4	45,0	39,0	43,9 ± 4,3
Hashing Vectorizer (Word)	47,0	43,6	39,1	43,5 ± 3,9
Spacy Model (BeRT)	13,6	14,0	17,5	15,0 ± 2,1

Table 11: The accuracy scores of the feature extractor benchmark on the train-test split by supermarket

3.1.3 Best Feature Extractor for Receipt Texts

From the evaluations of the two train-test splits in the previous subsections, we can conclude that the Tfidf-vectorizer with character n-gram tokens is best suited for handling receipt texts. The use of character n-grams appears to be the key. We suspect that this is due to the nature of the receipt texts, where the texts are short, typically consisting of only a handful of words and abbreviated words. This is particularly evident in the performance differences among supermarkets. Character n-gram ones may provide up to a 5% increase in accuracy compared to word-based ones on AH, where its receipt texts are distinctively shorter than the other supermarkets.

3.2 Benchmark ML Algorithms

In this section we benchmark the ML algorithms against each other. All ML algorithms are evaluated with the character n-gram Tfidf vectorizer, the winner of the feature extractor benchmark in the previous section. Similar to what was done there, we first evaluate the ML algorithms on the train-test split by time in Section 3.2.1, and then on the train-test split by supermarket Section 3.2.2.

3.2.1 Assessment Train-test split by time: Performance Stability over Time

Table 12 shows the accuracy scores of the ML algorithms over all the three test sets. Table 13 shows the same results by supermarket in the test sets. All algorithms were evaluated with the same feature extractor: the Tfidf-vectorizer fit on character n-gram tokens. We remind the reader that some ML algorithms were trained on a random sample of 100.000 receipt texts rather than the full data set as described earlier in Section 2.2. More scores can be found in Appendix B.

We first notice that a simple exact string-matching algorithm can already achieve reasonable performance (> 80% on all three tests), indicating that the tests are not too difficult due to the considerable overlap between training and test sets. However, the same algorithm does show a performance degradation of 1% - 2% per month caused by the introduction of new products.

While a similar trend is observed for the ML algorithms, the decline is less rapid, with an overall performance decrease of around 0,5% per month. But aside from being able more stable, the best ML-algorithm also performs better than string-matching (mean accuracy of 89% compared to 83%). The most significant improvement can be seen on the set of Lidl receipts in Table 13. The pipeline by DeStatis scored quite similarly to the best ML algorithm, even outperforming them on some store-specific evaluations in Table 13. But once the article ID matching is removed, the pipeline scored similarly to Random Forest.

However, among the ML-algorithms, we do notice that the models trained on a random sample (Regular Logistic Regression, Multilayer Perceptron, and Naïve Bayes) performed worse than the ones trained on the full data. This is especially evident for receipt texts from AH or Lidl as shown in Table 13. We suspect that this is because these stores form a small part of the training data, which means that their probability from being sampled in the random sample is even smaller.

Table 14 and Table 15 show the quantifications of the SGD-variant of Logistic Regression on the August 2023 test set in COICOP levels 1 and 3 (Food and non-alcoholic drinks only), respectively. The tables show the estimated and actual number in each category. Here we see that the estimations in the higher COICOP levels are well within 5% from the actual numbers, barring some minority categories. However, as soon as we zoom in to COICOP level 4 as shown in Table 16, we observe some worst cases of over- and underestimations. This suggests that the information on receipt texts seems to be insufficient for the required preciseness of some categories in COICOP level 4.

Upon manual inspection of the misclassifications, we have found the following cases:

- Ambiguity in receipt text
    - Receipt text consists of only a brand name, where brands may produce various types of products.
    - Receipt text contains insufficient information for the required preciseness of the target category. Some examples:
        - "AH Garnalen" (AH Shrimps) to *fresh or chilled seafood* or *fresh or chilled seafood*;
        - "AH zalmfilet" (AH Salmon filet) to *Frozen Fish* or *Fresh or chilled fish*;
        - "Boek" (Book) to category fiction or non-fiction;
        - "AH pasta" to *pasta products and couscous* or *ready-made meals.*
- Receipt texts with incorrect labels (ground truth)
    - "Proteinbar wit choc" labeled as *Magazines and periodicals.*
- Ambiguity due to miscellaneous subcategories:
    - "Olijfolie" (Olive oil) were sometimes labeled as *Other food products* while classified *Olive Oil*.

ML Algorithm	Jun 2023	Jul 2023	Aug 2023	Mean tests
Logistic Regression (SGD)	89,9	89,4	88,8	89,4 ± 0.5
DeStatis Pipeline	89,9	88,8	88,2	89,0 ± 0,7
Random Forest Classifier**	89,9	89,2	88,6	89,2 ± 0.7
DeStatis Pipeline (w/o art. ID matching)	88,3	87,4	86,7	87,5 ± 0,7
Exact String-Matching Algorithm	84,6	82,7	81,5	82,9 ± 1.5
FastTextClassifier	82,2	82,1	80,4	81,6 ± 0.4
Multilayer Perceptron Classifier*	81,4	81,7	80,8	81,3 ± 0.4
Logistic Regression*	77,7	78,0	77,4	77,6 ± 0.5
Naïve Bayes (Multinomial)*	77,4	77,6	76,8	77,3 ± 0.4
Constant Predictor	9,9	10,5	9,9	10,1 ± 0.3

Table 12: The accuracy scores in % of the ML-algorithms on the three test sets: June, July and August 2023. The column "Mean tests" show the mean and standard deviations (prefixed with the ± symbol) over all three tests.

	AH			Lidl			Plus		
	Jun	Jul	Aug	Jun	Jul	Aug	Jun	Jul	Aug
Logistic Regression (SGD)	88	87	87	91	91	89	91	90	90
DeStatis Pipeline	93	92	92	93	92	89	88	87	87
DeStatis Pipeline (w/o art. ID matching)	88	87	87	92	91	88	88	87	87
Random Forest	87	87	86	91	90	89	91	90	89
Exact String-Matching	88	87	86	88	84	79	83	81	80
FastTextClassifier	69	70	71	79	82	80	81	84	82
Multilayer Perceptron	70	69	69	78	77	76	86	86	86
Logistic Regression	63	63	63	73	73	72	84	83	83
Naïve Bayes (Multinomial)	66	66	66	75	75	74	82	81	81
Constant Predictor	7	6	6	7	8	8	12	12	12

Table 13: The accuracy scores of the ML algorithms per supermarket on the test sets June, July and August 2023.

COICOP Level 1	Actual Aug (n)	Est. Aug (n)	CR
1: Food and Non-Alcohol Beverages	60.078	60.084	1,00
2: Alcoholic Beverages, Tobacco, and Narcotics	6.120	6.202	1,01
3: Clothing and Footwear	415	415	1,00
5: Furnishings Household Equipment	2.463	2.481	1,01
6: Health	207	230	1,11
8: Communication	15	14	0,93
9: Recreation and Culture	3.444	3.388	0,98
12: Personal Care	1.506	1.434	0,95

Table 14: The number of estimated and actual number of receipt texts COICOP categories, aggregated by COICOP-level 1.

COICOP Level 3 (*01: Food & Non-alcoholic drinks* only)	Actual Aug (n)	Est. Aug (n)	CR
0111: Bread and cereals	12.599	12.710	1,01
0112: Meat	10.715	10.785	1,01
0113: Fish and seafood	1.059	1.096	1,03
0114: Milk	7.164	7.252	1,01
0115: Oils and fats	366	368	1,01
0116: Fruit	2.218	2.252	1,02
0117: Vegetables	6.642	6.580	0,99
0118: Sugar, jam, honey, chocolate, and confectionary	3.505	3.688	1,05
0119: Food products	12.153	11.669	0,96
0121: Coffee, tea, and cocoa	1.123	1.149	1,02
0122: Mineral waters, soft drinks, fruit and vegetable juices	2.534	2.535	1,00
Total	60.078	60.084	

Table 15: The number of receipt texts classified to the COICOP categories belonging under category 01: Food & Non-alcoholic drinks.

COICOP Level 4	Actual Aug (n)	Est. Aug (n)	CR
01153: Olive oil	38	54	1,42
09311: Pre-recorded diskettes and CD-ROMs …	3	4	1,33
09521: Newspapers	93	121	1,30
09549: Toner and ink-cartridges	253	328	1,30
12329: Other personal effects (miscellaneous)	22	27	1,23
…	…	…	…
09530: Miscellaneous printed matter	165	68	0,41
09511: Fiction books	182	63	0,35
01134: Frozen seafood	9	3	0,33
01132: Frozen fish	12	3	0,25
09513: Other non-fiction books	36	3	0,08

Table 16: The worst instances of over- and underestimated categories in COICOP Level 4.

3.2.2 Assessment Train test split by supermarket: Performance on Unseen Supermarkets

In this subsection, we benchmark the ML algorithms against each other to test their performances on receipt text from unseen supermarkets. Table 17 shows the accuracy scores of the algorithms. Again, all algorithms were evaluated along with the same feature extractor: the Tfidf Vectorizer on character n-gram tokens. The algorithms that were fit on a random sample of a maximum of 100.000 instances are marked with an asterisk. More scores can be found in Appendix B.

Overall, we see that none of the algorithms could score higher than 50%. The highest average accuracy over all three splits is 48,1% by SGD Logistic Regression. However, all ML algorithms performed the the string-matching algorithms by a large margin, including the string-matching pipeline by DeStatis. This therefore indicates that, string-matching is not very effective on unseen supermarkets due to the large differences in receipt texts across supermarkets.

When comparing the various supermarkets, we again observe that the results in Plus are generally lower than the other supermarkets. But as explained in the previous section, this is likely due to the differences in sizes of the test sets.

Table 18 and Table 19 show the estimated and actual number instances by counting classifications and aggregating them to COICOP levels 1 and 3. These tables show that substantial over- and underestimations may occur on level 1, especially for non-food products. However, zooming in on only food-products in COICOP level 3, we see that most categories are estimated to be within 20% from the ground truth. In the worst over- and underestimations of COICOP level 4 in Table 20, however, we see that some categories in the finer grained levels may still be severely over- or underestimated. Therefore, again supporting the argument that receipt texts are not sufficient for disambiguating between the finer grained categories in COICOP level 4 and lower.

ML Algorithm	Split 1: Test AH (n=26.198)	Split 2: Test Lidl (n=25.719)	Split 3: Test Plus (n=455.227)	Mean tests
Logistic Regression (SGD)	49,6	45,3	49,5	48,1 ± 2,0
Multilayer Perceptron Classifier*	43,7	45,4	45,0	44,7 ± 0,7
Random Forest Classifier**	46,7	44,9	41,8	44,5 ± 2,0
Naïve Bayes (Multinomial)*	47,3	45,3	43,4	45,3 ± 1,6
Logistic Regression*	42,0	45,1	44,7	43,9 ± 1,4
FastTextClassifier	43,2	33,7	39,0	38,7 ± 3,9
DeStatis pipeline	35,9	10,0	10,9	18,9 ± 12,0
DeStatis pipeline (w/o article matching)	21,6	10,0	8,2	13,3 ± 5,9
Constant Predictor	6,4	6,9	13,1	8,8 ± 3,8
Exact String-Matching	4,7	2,9	0,7	2,8 ± 1.6

Table 17: The accuracy scores of the ML algorithms on the train-test splits by supermarket.

COICOP Level 1	Actual Plus (n)	Est. Plus (n)	CR
01: Food and Non-Alcohol Beverages	374.870	360.399	0,96
02: Alcoholic Beverages, Tobacco, and Narcotics	37.931	40.834	1,08
03: Clothing and Footwear	2.960	2.097	0,71
05: Furnishings Household Equipment	8.560	12.230	1,43
06: Health	3.406	2.100	0,62
08: Communication	14	3	0,21
09: Recreation and Culture	18.333	25.163	1,37
12: Personal Care	3.932	7.181	1,83
Total	**450.007**	**450.007**	

Table 18: The number of estimated and actual number of receipt texts COICOP categories, aggregated by COICOP-level 1.

COICOP Level 3 (01: Food & Non-alcoholic drinks)	Actual Plus (n)	Est. Plus (n)	CR
0111: Bread and cereals	99.387	99.899	1,01
0112: Meat	70.687	68.461	0,97
0113: Fish and seafood	4.756	5.671	1,19
0114: Milk	36.664	32.265	0,88
0115: Oils and fats	1.212	1.397	1,15
0116: Fruit	13.870	14.473	1,04
0117: Vegetables	31.533	41.157	1,31
0118: Sugar, jam, honey, chocolate, and confectionary	19.696	22.292	1,13
0119: Food products (miscellaneous)	81.244	57.647	0,71
0121: Coffee, tea, and cocoa	3.757	3.535	0,94
0122: Mineral waters, soft drinks, fruit and veg. juices	12.064	13.602	1,13
Total	**374.870**	**360.399**	

Table 19: The number of receipt texts classified to the COICOP categories belonging under category 01: Food & Non-alcoholic drinks.

COICOP Level 4	Actual Plus (n)	Est. Plus (n)	CR
01143: Other milk and cream	127	1.069	8,42

01116: Pasta products and couscous	548	3.422	6,24
01123: Meat, dried, salted, in brine or smoked	29	178	6,14
01111: Cereals	313	1.162	3,71
01136: Other seafood preparations	1.253	3.771	3,01
…	…	…	…
01133: Bread and bakery products	488	111	0,23
01193: Salt, condiments and sauces	1023	152	0,15
01125: Meat, offal, blood and other parts of … animals	3254	285	0,09
01142: Skimmed milk	1911	143	0,07
01135: Other seafood, dried, salted, in brine or smoked	714	11	0,04

Table 20: The worst instances of over- and underestimated categories in COICOP Level 4.

4. Conclusions and Discussion

In the first benchmark we tested various feature extractors against each other. We concluded that a Tfidf vectorizer on character n-grams is best suited for handling on receipt texts. Which is sensible given that receipt texts are typically short and are filled with abbreviations.

In the second benchmark we first tested ML algorithms on the train-test splits by time for testing the performance stability over time. When ML models only have to classify receipt texts from known supermarkets (i.e. supermarkets are also found in training set), we observe that ML models can provide better and more stable performances than regular string-matching. The best performing ML model, the SGD-variant of the Logistic Regression, scored 90% accuracy on the first month with a performance drop of 0.5% per month. This means that 90% of all receipt texts in the tests were classified into the right COICOP Level 5 class. The regular string-matching algorithm, for comparison, scored 85% on the same month and dropped 1,5% in accuracy per month.

However, when the algorithms are expected to classify receipt texts from previously unseen supermarkets (i.e. not included in training set), we observed that none could score higher than 50% accuracy on any of the supermarkets. This means that fewer than half of the receipt texts in the tests were correctly classified. While the results showed that ML algorithms perform significantly better than the string-matching alternatives, it suggests that the ML does not classify receipt texts from unseen supermarkets well.

From examining the quantifications of various COICOP levels, we further observed that receipt texts were insufficient for distinguishing some of the finer-grained categories in COICOP level 4. In some cases, it led to substantial over- and underestimations of some specific categories. In comparison, COICOP levels 1 to 3 generally showed less misestimation.

In conclusion, ML models can classify receipts well (90% accuracy) with a relatively stable performance (0,5% drop per month), but only on receipt texts from known supermarkets, i.e. supermarkets that are included in our training set. When this is not the case, then the expected performance drops to 50% accuracy.

There are various reasons as to why ML models perform poorly on unseen supermarkets. The first is that the products can vary significantly across supermarkets. This is not only true for the types of products offered, e.g. Lidl regular sells non-food products such as laptops, but also for the brands that are featured in the stores. The second reason is due to the nature of the receipt texts. The maximum number of characters in receipt texts varies per supermarket, from 12 characters for AH, to 35 characters for Lidl. Moreover, supermarkets tend to adopt different naming and abbreviation conventions when coming up with receipt texts.

Although our results indicate that the performance degradation rate is 0,5% per month, we are aware that this conclusion has been drawn from only three months' worth of test data. Two questions arise from this. The first is whether this rate is constant over a longer period, and the second is whether the performance will keep degrading to 0% or if there is some lower bound to which it will never drop under. While the first question remains unanswered for now, we may deduce an estimate for the lower bound using the current results. If we assume that the product inventory of a supermarket at some point diverges to such an extent that it essentially

becomes a different supermarket, i.e. becoming an unseen supermarket, then we may use the results from the other evaluations to deduce a lower bound accuracy of 50%.

Some other limitations of this study include the fact that the tests on performance stability only featured three consecutive summer months. Results may therefore hold limited external validity. Preferably, we would have enough data to test on a whole year to properly evaluate the rate of decline and the effect of seasonality. Further, the experiments could be improved by adding auxiliary features of the respondent, e.g. gender, age, that may help the classification task.

We plan on examining the following points in the future:

- testing the robustness against OCR artefacts. Since the ML models are expected to be used on digitized receipt texts from OCR scans, typos are therefore bound to occur in the receipt texts. The current results are only valid under the assumption that receipt texts are perfectly scanned;
- re-evaluating the tests by incorporating instances labeled with 999999;
- evaluating the effect of stratification strategies during training, e.g. stratification by supermarket or COICOP-class on the classification performance;
- examining correction methods to account for bias introduced by stratification strategies (Puts & Gootzen, 2024).
- weighing instances during training based on their product occurrences.
- adding features such as price, quantity, or respondent characteristics.
- testing other ML techniques based on word-embedding, e.g. FastText and GenSim. especially with regards retraining pretrained models.
- Testing the effect of seasonality. This can be done on Lidl and Plus as more data is available for these supermarkets.

5. References

de Jong, T., Lam, C., & Puts, M. (2024). *Store product and dynamics in product texts used for COICOP classification.* Internal report.

Forman, G. (2005). Counting positives accurately despite inaccurate classification. *European conference on machine learning.*

Harteloh, P. (2020). The implementation of an automated coding system for cause-of-death statistics. *Informatics for Health and Social Care*, 1-14.

Puts, M., & Gootzen, Y. (2024). Correcting for Misclassification Bias: an Bayesian Ideal Classification Approach.

6. Appendix A: Best Hyperparameters

6.1 Feature extractors

Feature Extractor	Hyperparameter setting	Best Hyperparameter value
Tfidf Vectorizer (Char. n-grams)	ngram_range	(2, 4)
	lowercase	True
	analyzer	char
	binary	True
Count Vectorizer (Char. n-grams)	ngram_range	(2, 4)
	Lowercase	True
	analyzer	Char_wb
	binary	True
Hashing Vectorizer (Char. n-grams)	ngram_range	(2, 4)
	Lowercase	True
	analyzer	Char_wb
	binary	True
Hashing Vectorizer (Word)	lowercase	True
	analyzer	word
	binary	True
Tfidf Vectorizer (Word)	lowercase	True
	analyzer	word
	binary	True
Count Vectorizer (Word)	lowercase	True
	analyzer	word
	binary	True
Spacy Model (BeRT)	model_name	"nl_core_news_md"

Table 21: The best hyperparameters for the feature extractors from grid search hyperparameter optimization.

6.2 ML Algorithms

Algorithm	Hyperparameter Name	Hyperparameter value
FastTextClassifier	Epoch	20
	Lr	0.5
	Loss	Softmax
	bucket	4.000.000
Logistic Regression (SGD)	Loss function	Log loss
	Alpha	1e-6
Logistic Regression	C	10
	Solver	saga
Random Forest Classifier	Max features	Log2
	Max samples	100.000
Multilayer Perceptron	Alpha	0.01
Multinomial Naïve bayes	alpha	0.0001
Constant Predictor (DummyClassifier)	Strategy	prior

Table 22: The best hyperparameters for the from grid search hyperparameter optimization.

7. Appendix B: More in-depth results

Algorithm	Jun 2023			Jul 2023			Aug 2023		
Metric	**ACC**	**PREC**	**REC**	**ACC**	**PREC**	**REC**	**ACC**	**PREC**	**REC**
Logistic Regression (SGD)	89,9	86,5	83,2	89,4	85,3	84,3	88,8	86,4	83,1
DeStatis Pipeline	89,9	85,4	88,1	88,8	85,9	87,5	88,2	87,3	86,3
Random Forest Classifier**	89,9	87,5	81,4	89,2	87,7	81,7	88,6	88,6	79,8
DeStatis Pipeline (w/o art. ID matching)	88,3	83,2	83,8	87,4	83,9	83,4	86,7	86,0	82,1
FastTextClassifier	82,2	73,6	73,9	82,1	74,9	75,8	80,4	75,1	74,9
Exact String-Matching Algorithm	84,6	87,5	80,3	82,7	86,3	80,1	81,5	88,0	77,9
Multilayer Perceptron Classifier*	81,4	80,9	57,9	81,7	81,3	58,2	80,8	81,4	57,5
Logistic Regression*	77,7	76,3	53,1	78,0	75,6	53,6	77,4	75,8	53,1
Naïve Bayes (Multinomial)*	77,4	75,5	60,2	77,6	72,8	62,1	76,8	74,7	61,5
Constant Predictor	9,9	99,1	1,0	10,5	99,1	1,0	9,9	99,1	1,0

Table 23: Additional accuracy, precision and recall scores of the algorithms on the train-test split by time

Algorithm	Split 1: Test AH			Split 2: Test Lidl			Split 3: Test Plus		
Metric	**ACC**	**PREC**	**REC**	**ACC**	**PREC**	**REC**	**ACC**	**PREC**	**REC**
Logistic Regression (SGD)	49,6	61,9	44,8	45,3	54,9	48,1	49,5	31,9	54,7
Multilayer Perceptron Classifier*	43,7	58,5	38,6	45,4	53,5	45,4	45,0	32,5	53,0
Random Forest Classifier**	46,7	55,9	44,2	44,9	45,6	51,5	41,8	27,2	52,6
Naïve Bayes (Multinomial)*	47,3	53,2	45,2	45,3	44,0	51,7	43,4	31,2	53,2
Logistic Regression	42,0	54,9	39,6	45,1	53,0	45,7	44,7	32,8	53,3
FastTextClassifer	43,2	44,3	43,3	33,7	25,8	52,7	39,0	24,9	50,8
DeStatis pipeline	35,9	61,7	36,0	10,0	48,5	26,0	10,9	39,0	34,7
DeStatis pipeline (w/o article matching)	21,6	53,0	25,4	10,0	49,9	24,5	8,2	34,9	30,4
Constant Predictor	6,4	98,9	1,2	6,9	98,7	1,4	13,1	98,9	1,3
Exact String-Matching	4,7	70,1	6,14	2,9	61,4	9,8	0,7	43,3	24,9

Table 24: Additional accuracy, precision and recall scores of the algorithms on the train-test splits by supermarket

[i] Scikit-learn chart: https://scikit-learn.org/stable/machine_learning_map.html

Annex 2: Mode of transport

See .pdf Annex_WP3 SSI_GEO-TRANSPORTMODE.

Development and performance of a transport mode classification algorithm for smart surveys

Jonas Klingwort, Yvonne Gootzen, Jurgens Fourie

31.03.2025

Contents

1 Introduction

An integral component of smart time-use, travel, and mobility surveys is the ability to predict respondents' modes of transportation, thereby minimizing the necessity for manual data labeling and reducing the response burden. This report documents the development of a transport mode prediction algorithm specifically designed for integration with smart surveys. The algorithm is based on a decision tree, using smartphone GPS data and infrastructure information from OpenStreetMap (OSM). The development of the algorithm was based on data collected by Statistics Netherlands (Schouten et al. 2024). The algorithm was also evaluated on open geo-data that is publicly available in the SSI Git-repository (https://github.com/essnet-ssi/geoservice-ssi) and was collected within the scope of the SSI project. This document provides a comprehensive overview of the algorithm's development, describing the development procedures, the datasets utilized, the underlying methodology, and the resulting outcomes.

2 Background

Transport mode prediction currently lacks a universally established algorithm. Existing methods predominantly rely on manual rule-based approaches, decision trees, or machine-learning techniques. Within this project's scope, an extensive review of existing methodologies and algorithms was conducted by Fourie (2025). For developing a transport mode classification algorithm within the SSI project, it was decided to base it on a decision tree due to its simplicity and interpretability. We briefly discuss their advantages and disadvantages and compare manual rule-based approaches, decision trees, and machine-learning models. Some of the benefits of decision-tree models are listed below:

1. Interpretability: Decision trees provide a transparent, easy-to-understand decision-making process, making them ideal for explaining predictions.
2. Non-linearity: They can model complex relationships between input features without requiring linear assumptions.
3. Feature importance: Decision trees naturally rank features based on their importance, helping understand key factors affecting transport mode choices.
4. Categorical & numerical data: They can process different data types (e.g., numeric, timestamps, categorical travel modes) without complex preprocessing.
5. Computational efficiency: Training and prediction are relatively fast, making them suitable for real-time transport mode prediction in smart surveys.
6. Missing data: Decision trees can handle missing values better than some machine learning models using surrogate splits.

Some of the disadvantages of decision-tree models are listed below:

1. Overfitting: Decision trees can overfit the training data, leading to poor generalization unless pruning techniques are applied.
2. Sensitivity to noisy data: Small variations in input data can lead to different splits, making the model unstable.
3. Limited expressiveness: Decision trees can handle complex patterns but may struggle with highly complex relationships between features compared to deep learning models.
4. Bias in splitting criteria: Splitting criteria like the Gini index or Information Gain tend to favor features with more levels, which might lead to biased predictions.

We also give a brief comparison of decision trees with rule-based approaches and with machine-learning approaches. Decision trees are more flexible and scalable than rule-based approaches but may overfit the data. Rule-based methods are static and rely heavily on expert knowledge, which may not generalize well. Decision trees can be trained automatically while rule-based approaches are handcrafted. This comparison is summarized in Table 1.

Theme	**Decision Trees**	**Manual Rule-based Approaches**
Flexibility	Adapts to patterns in data automatically	Requires manually defined rules
Interpretability	Easy to understand	Easy to understand
Scalability	Scales well with data size	Becomes complex with increasing rules
Handling New Data	Can retrain to adjust	Needs manual updates
Accuracy	Higher with enough data	Limited by predefined rules

Table 1: Comparison of decision trees and rule-based approaches

While decision trees offer simplicity and interpretability, they may not be as accurate as ensemble methods (like Random Forests) or deep learning models. Table 2 gives a brief summary:

Theme	**Decision Trees**	**Random Forest**	**Neural Networks**
Interpretability	High	Medium (ensemble of trees)	Low (black box model)
Accuracy	Moderate	High	Very High
Computational Cost	Low	Medium	High
Handling Overfitting	Pruning needed	Less prone (ensemble effect)	Requires regularization
Training Speed	Fast	Slower than single tree	Slowest

Table 2: Comparison of decision trees with machine learning models

Alongside this project, and as an integral component of its development, a rule-based algorithm for transport mode classification was also developed by Fourie (2025). The results obtained during the development of this algorithm contributed to the advancement of this project. The algorithm with results is included in the Appendix D. The findings derived from the work by Fourie (2025) are expected to be published soon by Fourie et al. (expected 2025). This simple rule-based algorithm performed reasonably well but had the drawback of resulting in multiple classifications. The developed rule-based approach is a non-nested system that evaluates conditions independently and

selects the best match without a strict hierarchy. This fact can lead to overlapping or conflicting conditions requiring priority handling. No priority handling is needed using a decision tree because each track will be assigned one predicted transport mode.

3 Data

Two datasets were used to develop and evaluate the transport mode prediction algorithm. First, data from a large-scale field study conducted by Statistics Netherlands was used to develop and test the algorithm's internal validity. This dataset is not publicly available. Second, data was collected within the SSI project to create an open, publicly available geo-dataset with error-free labels. This dataset was used to evaluate the generalizability of the algorithm and establish its external validity.

3.1 Development data

The dataset is based on a Dutch general population sample collected from 2022 to 2023. Data from 255 participants were used for the development. The dataset contains 4,298 tracks and a total of about 20 million observations. An observation consists of a timestamp and geo-location (longitude and latitude). We refer to Schouten et al. 2024 for general details about the dataset.

3.1.1 Data processing

The dataset underwent preprocessing steps to ensure quality and consistency. Gootzen et al. expected 2025 describes the initial and general data cleaning procedures and informs on the data quality. For the specific development of the transport mode classification algorithm, tracks exceeding 10 hours were excluded. Second, tracks containing fewer than ten GPS observations were removed. Third, labels for similar transport modes were grouped: 'car (driver)' and 'car (passenger)' were merged into a single transport mode. The categories 'bike' and 'e-bike' were also merged into one transport mode. After preprocessing, the average number of GPS observations per track was 843, although the median was notably lower at 402, indicating a skewed distribution. Similarly, the average track duration was 53 minutes, but the median was 12 minutes, reflecting the skewness. Regarding track length, the mean was 15 km, while the median was considerably shorter at 2.8 km, again indicating a skewed distribution in the data.

3.1.2 Transport modes

The target variable of the classification task is the transport mode used during a track. The distribution of track labels across transport modes in the entire dataset is presented in Table 3. The labels indicate that the developed algorithm will only classify a single mode. Multi-modal tracks cannot be classified. This fact is not a shortcoming caused by the algorithm, but the labels to develop/train the algorithm do not contain multi-modal tracks. The target variable also contains the label 'Other'. This label is expected to reduce the overall quality of the algorithm as it is not a defined mode of transport and can contain a wide variation of transport modes. This

label was excluded to only classify well-defined transport modes in the study by Fourie (2025).

Mode	Count	Percentage
Car	1,892	44.02
Walk	1,002	23.31
Bike	946	22.01
Train	161	3.75
Other	111	2.58
Bus	96	2.23
Metro	58	1.35
Tram	32	0.74
Total	4,298	100

Table 3: Distribution of transport modes in development data. Rows ordered by count.

3.1.3 Train and test splits

The dataset was partitioned at the user level, ensuring that each user was assigned exclusively to the training or testing set, but not both. This approach was chosen to evaluate the model's ability to generalize to entirely unseen users, providing a strict and realistic assessment of generalization in scenarios where new users are encountered in future surveys. By separating users in this manner, the risk of overfitting is mitigated, as the model is prevented from learning user-specific patterns from the training set that could influence predictions in the test set. However, this approach introduces specific challenges. The number of users in the dataset is limited, and some users contribute disproportionately, with a large number of tracks attributed to a single individual. As a result, the train-test splits may become imbalanced across labels, potentially affecting the robustness and generalizability of the algorithm. Therefore, it was decided to split the dataset by partitioning users into separate subsets for training (70%) and testing (30%), ensuring that no user appeared in both sets. Stratification was applied based on each user's dominant mode of transport to maintain a balanced representation of transport modes. The public transport modes bus, metro, and tram were grouped for the train and test splits (they were used as individual classes for the remainder of the development). This practical solution prevented the case that tram was once only the most prominent mode, and therefore, no split could have been applied because the stratification would require at least two occurrences of a transport mode. This approach preserved the variation and distribution of transport modes across both subsets, ensuring that the test set accurately reflected the training data's characteristics while preventing user overlap between the two sets. Tables 4 and 5 show the train and test splits.

3.2 Open geo-data

This dataset was reserved exclusively for testing the developed algorithm, with no portion used during the development or training phases, ensuring an unbiased evaluation of the algorithm's generalization capabilities. The dataset was collected in the

Mode	Count	Percentage
Car	1,333	44.43
Walk	684	22.80
Bike	643	21.43
Other	103	3.43
Train	99	3.30
Bus	75	2.50
Metro	45	1.50
Tram	18	0.60
Total	3,000	100

Table 4: Train set. Rows ordered by count.

Mode	Count	Percentage
Car	559	43.07
Walk	318	24.50
Bike	303	23.34
Train	62	4.78
Bus	21	1.62
Tram	14	1.08
Metro	13	1.00
Other	8	0.62
Total	1,298	100

Table 5: Test set. Rows ordered by count.

summer of 2024 using the most recent version of the CBS smartphone app available at that time. This app version employed a revised sensor configuration compared to the app used to collect the development data. The updated configuration reduced the number of sensors used in the smartphone to collect GPS but prioritized collecting more detailed data from a single sensor type. The data was collected to obtain data with high-quality labels without errors for the transport mode. This data was collected by a small group of CBS staff and staff from the University of Utrecht. Furthermore, the data contains tracks within the Netherlands and Germany. Accordingly, this test set will inform how well the algorithm generalizes to a different app version/sensor configuration and data collected in a different country. Data from 5 users with 137 tracks are available. The transport mode distribution is shown in table 6.

Mode	Count	Percentage
Walk	78	0.61
Tram	27	0.21
Bike	10	0.08
Train	9	0.07
Bus	5	0.04
Metro	5	0.04
Ferry	3	0.02
Total	137	100

Table 6: Transport modes in open geo-data. Rows ordered by count.

Note that the decision tree was not trained on data containing the 'ferry' label. Thus, the algorithm will fail to predict this label. However, it was collected to evaluate the algorithm's decision for this label. Lastly, the most prominent mode in the development data, 'car', is not included. This dataset is publicly available in the SSI Git repository (https://github.com/essnet-ssi/geoservice-ssi).

4 Methods

The methods section contains the construction of GPS features (Section 4.1), the construction of OSM features (Section 4.2), the pre-processing of GPS and OSM feature (Section 4.3), development of the decision tree (Section 4.4), and the transport mode prediction algorithm (Section 4.4).

4.1 Feature construction GPS

This section describes the required GPS features for the algorithm. In particular, features from five themes were created and evaluated: speed (speed, acceleration, jerk, snap), GPS (accuracy, frequency), direction (bearing, altitude), trip (length, duration), and time (weekday, weekend indicator). For most features, several variants were created based on different statistics. A list of all GPS features is given in Appendix A. Even more GPS features were evaluated by Fourie et al. (expected 2025). However, as they have been found not to be relevant, they were not implemented here. The required Python code can be found in the accompanying script **gps_feature.py**. A list with a short description of all Python scripts can be found in Appendix C.

4.2 Feature construction OSM

This section describes the required OSM features for the algorithm. A list of all used OSM features is given in Appendix B. The features are based on publicly available OpenStreetMap (OSM) data obtained from the official Geofabrik download portal (https://download.geofabrik.de/). A documentation of all OSM infrastructure contained in the database can be found at: https://wiki.openstreetmap.org/wiki/Map_features. OSM data complements the GPS features described in Section 4.1 by providing details on infrastructure such as road networks, transit routes, and stations. Integrating this data improves usually the quality of the transport mode classifications (Fourie 2025; Gong et al. 2012; Sadeghian et al. 2022; Smeets et al. 2019). Fourie (2025) systematically studied which OSM features have the most considerable potential to improve the transport mode classifications. The main findings by Fourie (2025) were, a) that OSM did not help to improve the classification quality for the transport modes walk, bike, and car. Based on these findings, it was decided not to create features using OSM-specific information for these three transport modes, and b) that although OSM provides a variety of data on transportation and travel infrastructure – including features such as roundabouts, traffic junctions, stop signs, speed cameras, and street lamps, these did not help improve the classification performance. Another reason to limit the number of OSM features is computational efficiency. Accordingly, in the development of the algorithm, only OSM features about bus, metro, train, and tram stops and/or routes were created. The required Python code can be found in the accompanying script **osm_features.py**. Even more OSM features were evaluated by Fourie et al. (expected 2025). However, as they have found not to be relevant, they were not implemented here.

Track buffering

Buffering a GPS track when calculating features using OSM data is beneficial because it helps include relevant spatial context around the track, improving feature extraction and accuracy. In the following, we explain why this step is helpful. First,

it accounts for GPS inaccuracies and noise. GPS tracks often have errors due to signal loss, multipath effects, or device inaccuracies. A buffer helps compensate for small deviations and ensures relevant OSM features are included even if the track is slightly misaligned. Second, it captures nearby infrastructure and context. Many transport mode features depend on proximity to roads, paths, and transit stops. A buffer ensures that all relevant OSM data is considered within a reasonable distance of the track. This is especially important in urban environments where GPS can jump between nearby roads. Third, it enables more robust feature engineering. By including OSM features within the buffer, one can calculate more informative features such as, for example, pathway availability (e.g., bike lanes, sidewalks, pedestrian zones) or transit accessibility (e.g., nearest bus/tram stops). Fourth, it handles transport mode variability. A narrow track-only approach may miss important context (e.g., a train passenger may be slightly off the designated rail network). To conclude, buffering the GPS track will likely improve spatial accuracy, feature information, and mode classification robustness when integrating OSM data.

The buffering process takes the GPS coordinates representing the track and generates a buffer zone around it. This buffer is defined by a specified radius or distance, which determines how far the area extends from the track's centerline. For instance, a buffer with a radius of 25 meters would create a region 25 meters wide on either side of the track. This is the radius so that the diameter will be 50m. An example of this procedure is shown in Figure 1.

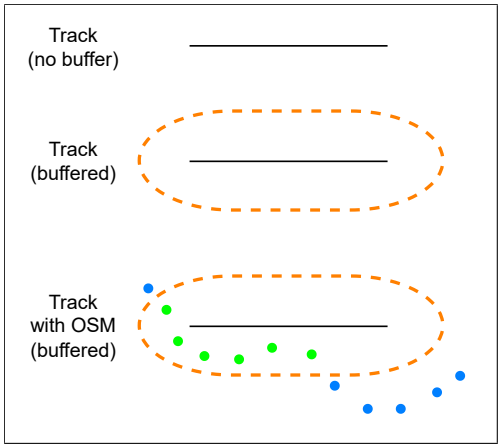


Figure 1: Simplified example of track buffering: a single track (black solid line), a buffered track (black solid line with surrounding orange dashed line), and a buffered track with mapped OSM infrastructure (black solid line with surrounding orange dashed line and mapped OSM infrastructure. Green points in the buffer are considered for feature construction, blue points outside the buffer are not.)

Once the buffer is constructed, spatial operations are performed to identify which OSM coordinates or features lie within the buffered area. This is achieved using spatial indexing and intersection techniques, which compare the locations of OSM features to the buffer's boundaries. The accompanying Python script **osm_features.py** contains the code to apply the buffering. Representing the track as a linestring object instead of considering the individual measurements enormously increased computational efficiency. Points-of-interest that fall within or intersect the buffer are retained

for further analysis. For the OSM count features, the total buffer was also used to normalize features for a fair comparison between shorter and longer tracks. In contrast to Fourie (2025), who used a 10 meter buffer, a buffer of 25 meters was used. It was tested whether different buffer sizes (10, 20, 50, 75, and 100 meters) were having an effect. No noticeable changes in results were observed (Fourie et al. expected 2025). The lack of an impact might be due to the fixed thresholds used in the rule-based algorithm.

4.3 Pre-processing of GPS and OSM features

Some GPS calculations did not result in reasonable numeric values. If the calculation of a feature resulted in an infinite value, the infinite value was replaced with twice the maximum value (inf → 2 * max). A negative infinity value was set to zero (-inf → 0). Missing values remained unchanged since a decision tree can handle missing data. String variables were factorized for the decision tree. For the OSM features, some variables contained missing values. This occurs when there is no OSM infrastructure in the buffer of a track. Here, the missing data was replaced with a zero count, reflecting this feature's actual absence.

4.4 Decision tree development

A decision tree is a supervised learning algorithm used for classification and regression tasks. It is a tree-like model where each internal node represents a decision based on a feature, each branch represents an outcome of that decision, and each leaf node represents a final prediction. In the following, we briefly explain the components of the tree. The tree starts with the root node, the topmost node of the tree. It represents the entire dataset and the first decision point based on a selected feature. The decision nodes are intermediate nodes that split data based on a condition. Each decision node applies a rule (e.g., speed > 30km/h?) and branches accordingly. The branches (or edges) represent possible outcomes of a decision. They connect nodes and direct the data down the tree. The leaf nodes (or terminal nodes) represent the final outcome/classification (e.g., 'Car' or 'Bike'). The process of dividing a node into two or more sub-nodes is based on feature conditions.

Optimizing hyperparameter space of decision tree

Grid search was done which is a hyperparameter tuning technique used to find the best combination of parameters that optimize the model's performance. It systematically searches through a predefined set of hyperparameters by testing all possible combinations and selecting the best one based on a scoring metric. The hyperparameter search was conducted using the following space:

$$
\begin{aligned}
\text{Maximum depth:} &\quad d \in \{3, 5, 7\},\\
\text{Minimum samples split:} &\quad m_{\text{split}} \in \{10, 25, 50\},\\
\text{Minimum samples leaf:} &\quad m_{\text{leaf}} \in \{5, 10, 15, 20, 25\},\\
\text{Criterion:} &\quad c \in \{\text{gini}, \text{entropy}\},\\
\text{Maximum features:} &\quad f \in \{1, 3, 5, 7\}.
\end{aligned}
$$

The output for the optimal decision tree and the transport mode prediction algorithm respectively is shown below. The required Python code for the algorithm can be found in the accompanying script **train_decision_tree.py**.
The file **decision_tree_ssi.pickle** contains the trained decision-tree model.

Transport mode prediction algorithm

```
node=0 is a split node with value=[anders: 0.034, auto: 0.444, bus: 0.025, fiets: 0.214, metro: 0.015, tram: 0.006,
     trein: 0.033, voet: 0.228]: go to node 1 if bus_route_mean_distance <= 1287.4715576171875 else to node 64.
        node=1 is a split node with value=[anders: 0.022, auto: 0.288, bus: 0.016, fiets: 0.297, metro: 0.005, tram
            : 0.007, trein: 0.004, voet: 0.36]: go to node 2 if speed_percentile_10 <= 4.045245885848999 else to
            node 39.
                node=2 is a split node with value=[anders: 0.029, auto: 0.255, bus: 0.013, fiets: 0.235, metro:
                    0.007, tram: 0.005, trein: 0.003, voet: 0.453]: go to node 3 if speed_iqr_value <=
                    5.45417857170105 else to node 16.
                        node=3 is a split node with value=[anders: 0.014, auto: 0.062, bus: 0.003, fiets: 0.089,
                            metro: 0.006, tram: 0.002, voet: 0.824]: go to node 4 if speed_percentile_80 <=
                            10.346889019012451 else to node 15.
                                node=4 is a split node with value=[anders: 0.013, auto: 0.066, bus: 0.003, fiets:
                                    0.048, metro: 0.007, tram: 0.002, voet: 0.862]: go to node 5 if
                                    jerk_percentile_85 <= 7493698.0 else to node 10.
                                        node=5 is a split node with value=[anders: 0.022, auto: 0.231, fiets:
                                            0.088, metro: 0.011, tram: 0.011, voet: 0.637]: go to node 6 if
                                            altitude_percentile_85 <= 2.445638060569763 else to node 9.
                                                node=6 is a split node with value=[anders: 0.027, auto: 0.178,
                                                    fiets: 0.055, metro: 0.014, voet: 0.726]: go to node 7 if
                                                    altitude_percentile_85 <= 0.14158499240875244 else to node 8.
                                                        node=7 is a leaf node with values=[auto: 0.303, fiets:
                                                            0.121, metro: 0.03, voet: 0.545].
                                                        node=8 is a leaf node with values=[anders: 0.05, auto:
                                                            0.075, voet: 0.875].
                                                node=9 is a leaf node with values=[auto: 0.444, fiets: 0.222, tram:
                                                     0.056, voet: 0.278].
                                        node=10 is a split node with value=[anders: 0.012, auto: 0.037, bus: 0.004,
                                             fiets: 0.041, metro: 0.006, voet: 0.902]: go to node 11 if
                                            metro_route_min_distance <= 0.024851143825799227 else to node 14.
                                                node=11 is a split node with value=[anders: 0.012, auto: 0.038, bus
                                                    : 0.004, fiets: 0.042, voet: 0.903]: go to node 12 if
                                                    speed_percentile_5 <= 0.2018592208623886 else to node 13.
                                                        node=12 is a leaf node with values=[anders: 0.028, auto:
                                                            0.099, bus: 0.014, fiets: 0.057, voet: 0.801].
```

```
                                                node=13 is a leaf node with values=[anders: 0.006, auto:
                                                    0.014, fiets: 0.037, voet: 0.944].
                                            node=14 is a leaf node with values=[metro: 0.143, voet: 0.857].
                                    node=15 is a leaf node with values=[anders: 0.029, fiets: 0.824, voet: 0.147].
                            node=16 is a split node with value=[anders: 0.042, auto: 0.429, bus: 0.021, fiets: 0.367,
                                metro: 0.008, tram: 0.008, trein: 0.006, voet: 0.118]: go to node 17 if speed_stddev <=
                                10.804237842559814 else to node 32.
                                    node=17 is a split node with value=[anders: 0.052, auto: 0.297, bus: 0.018, fiets:
                                        0.487, metro: 0.008, tram: 0.006, trein: 0.004, voet: 0.128]: go to node 18 if
                                        bus_route_max_distance <= 144.82131958007812 else to node 25.
                                            node=18 is a split node with value=[anders: 0.114, auto: 0.217, fiets:
                                                0.408, metro: 0.005, tram: 0.005, trein: 0.005, voet: 0.245]: go to node
                                                 19 if accuracy_percentile_85 <= 14.302633285522461 else to node 22.
                                                node=19 is a split node with value=[anders: 0.18, auto: 0.261,
                                                    fiets: 0.486, voet: 0.072]: go to node 20 if proportion_10_30 <=
                                                     0.280659481883049 else to node 21.
                                                        node=20 is a leaf node with values=[anders: 0.179, auto:
                                                            0.464, fiets: 0.179, voet: 0.179].
                                                        node=21 is a leaf node with values=[anders: 0.181, auto:
                                                            0.193, fiets: 0.59, voet: 0.036].
                                                node=22 is a split node with value=[anders: 0.014, auto: 0.151,
                                                    fiets: 0.288, metro: 0.014, tram: 0.014, trein: 0.014, voet:
                                                    0.507]: go to node 23 if proportion_15_30 <= 0.2102891132235527
                                                    else to node 24.
                                                        node=23 is a leaf node with values=[anders: 0.018, auto:
                                                            0.123, fiets: 0.193, metro: 0.018, trein: 0.018, voet:
                                                            0.632].
                                                        node=24 is a leaf node with values=[auto: 0.25, fiets:
                                                            0.625, tram: 0.062, voet: 0.062].
                                            node=25 is a split node with value=[anders: 0.016, auto: 0.344, bus: 0.028,
                                                 fiets: 0.533, metro: 0.009, tram: 0.006, trein: 0.003, voet: 0.06]: go
                                                to node 26 if acc_kurt <= inf else to node 29.
                                                node=26 is a split node with value=[anders: 0.029, auto: 0.076,
                                                    fiets: 0.829, voet: 0.067]: go to node 27 if sd_time_diff_s <=
                                                    1.9156306385993958 else to node 28.
                                                        node=27 is a leaf node with values=[anders: 0.03, fiets:
                                                            0.925, voet: 0.045].
```

```
30                                                  node=28 is a leaf node with values=[anders: 0.026, auto:
                                                        0.211, fiets: 0.658, voet: 0.105].
31                                              node=29 is a split node with value=[anders: 0.009, auto: 0.476, bus
                                                    : 0.042, fiets: 0.387, metro: 0.014, tram: 0.009, trein: 0.005,
                                                    voet: 0.057]: go to node 30 if speed_iqr_value <=
                                                    20.525142669677734 else to node 31.
32                                                  node=30 is a leaf node with values=[anders: 0.01, auto:
                                                        0.171, bus: 0.01, fiets: 0.733, metro: 0.01, tram: 0.01,
                                                         voet: 0.057].
33                                                  node=31 is a leaf node with values=[anders: 0.009, auto:
                                                        0.776, bus: 0.075, fiets: 0.047, metro: 0.019, tram:
                                                        0.009, trein: 0.009, voet: 0.056].
34                                  node=32 is a split node with value=[anders: 0.019, auto: 0.741, bus: 0.028, fiets:
                                        0.085, metro: 0.009, tram: 0.014, trein: 0.009, voet: 0.094]: go to node 33 if
                                        speed_percentile_85 <= 30.46554946899414 else to node 34.
35                                      node=33 is a leaf node with values=[anders: 0.061, auto: 0.306, fiets:
                                            0.327, tram: 0.02, voet: 0.286].
36                                      node=34 is a split node with value=[anders: 0.006, auto: 0.871, bus: 0.037,
                                             fiets: 0.012, metro: 0.012, tram: 0.012, trein: 0.012, voet: 0.037]: go
                                             to node 35 if bus_route_std_distance <= 595.4812316894531 else to node
                                            38.
37                                          node=35 is a split node with value=[anders: 0.009, auto: 0.897,
                                                fiets: 0.017, metro: 0.009, trein: 0.017, voet: 0.052]: go to
                                                node 36 if speed_percentile_20 <= 0.006204613484442234 else to
                                                node 37.
38                                              node=36 is a leaf node with values=[anders: 0.077, auto:
                                                    0.615, fiets: 0.077, trein: 0.077, voet: 0.154].
39                                              node=37 is a leaf node with values=[auto: 0.932, fiets:
                                                    0.01, metro: 0.01, trein: 0.01, voet: 0.039].
40                                          node=38 is a leaf node with values=[auto: 0.809, bus: 0.128, metro:
                                                 0.021, tram: 0.043].
41                  node=39 is a split node with value=[anders: 0.004, auto: 0.38, bus: 0.027, fiets: 0.47, tram: 0.01,
                         trein: 0.006, voet: 0.103]: go to node 40 if proportion_45_80 <= 0.024996007792651653 else to
                        node 49.
42                          node=40 is a split node with value=[auto: 0.078, bus: 0.01, fiets: 0.757, tram: 0.007,
                                trein: 0.003, voet: 0.145]: go to node 41 if proportion_5_15 <= 0.6839917302131653 else
                                to node 48.
```

```
43                          node=41 is a split node with value=[auto: 0.091, bus: 0.012, fiets: 0.825, tram:
                                0.008, trein: 0.004, voet: 0.06]: go to node 42 if proportion_15_30 <=
                                0.04568206053227186 else to node 43.
44                              node=42 is a leaf node with values=[voet: 1.0].
45                              node=43 is a split node with value=[auto: 0.095, bus: 0.012, fiets: 0.863,
                                    tram: 0.008, trein: 0.004, voet: 0.017]: go to node 44 if
                                    avg_time_diff_s <= 1.8849532008171082 else to node 47.
46                                  node=44 is a split node with value=[auto: 0.058, bus: 0.01, fiets:
                                        0.903, tram: 0.005, trein: 0.005, voet: 0.019]: go to node 45 if
                                         accuracy_percentile_20 <= 5.499175310134888 else to node 46.
47                                      node=45 is a leaf node with values=[auto: 0.03, bus: 0.012,
                                             fiets: 0.929, tram: 0.006, trein: 0.006, voet: 0.018].
48                                      node=46 is a leaf node with values=[auto: 0.184, fiets:
                                            0.789, voet: 0.026].
49                                  node=47 is a leaf node with values=[auto: 0.324, bus: 0.029, fiets:
                                         0.618, tram: 0.029].
50                          node=48 is a leaf node with values=[fiets: 0.364, voet: 0.636].
51                  node=49 is a split node with value=[anders: 0.01, auto: 0.848, bus: 0.052, fiets: 0.026,
                        tram: 0.016, trein: 0.01, voet: 0.037]: go to node 50 if altitude_percentile_80 <=
                        0.8099796772003174 else to node 57.
52                          node=50 is a split node with value=[anders: 0.019, auto: 0.837, bus: 0.058, trein:
                                0.019, voet: 0.067]: go to node 51 if jerk_skew <= -1.5975200533866882 else to
                                node 56.
53                              node=51 is a split node with value=[anders: 0.011, auto: 0.832, bus: 0.063,
                                     trein: 0.021, voet: 0.074]: go to node 52 if jerk_percentile_90 <=
                                    21673985.0 else to node 53.
54                                  node=52 is a leaf node with values=[auto: 0.765, bus: 0.235].
55                                  node=53 is a split node with value=[anders: 0.013, auto: 0.846, bus
                                        : 0.026, trein: 0.026, voet: 0.09]: go to node 54 if
                                        busway_normcount <= 3.5516588923201198e-06 else to node 55.
56                                      node=54 is a leaf node with values=[auto: 0.778, bus:
                                            0.222].
57                                      node=55 is a leaf node with values=[anders: 0.014, auto:
                                            0.855, trein: 0.029, voet: 0.101].
58                          node=56 is a leaf node with values=[anders: 0.111, auto: 0.889].
59                  node=57 is a split node with value=[auto: 0.862, bus: 0.046, fiets: 0.057, tram:
                        0.034]: go to node 58 if train_route_mean_distance <= 1428.7440795898438 else to
                         node 63.
```

```
                                    node=58 is a split node with value=[auto: 0.89, bus: 0.012, fiets: 0.061,
                                        tram: 0.037]: go to node 59 if snap_min_value <= -2761052520448.0 else
                                        to node 60.
                                            node=59 is a leaf node with values=[auto: 0.429, fiets: 0.571].
                                            node=60 is a split node with value=[auto: 0.933, bus: 0.013, fiets:
                                                 0.013, tram: 0.04]: go to node 61 if max_time_diff_s <= 32.5
                                                else to node 62.
                                                node=61 is a leaf node with values=[auto: 0.979, fiets:
                                                    0.021].
                                                node=62 is a leaf node with values=[auto: 0.857, bus:
                                                    0.036, tram: 0.107].
                                    node=63 is a leaf node with values=[auto: 0.4, bus: 0.6].
node=64 is a split node with value=[anders: 0.054, auto: 0.693, bus: 0.039, fiets: 0.082, metro: 0.03, tram
    : 0.005, trein: 0.08, voet: 0.017]: go to node 65 if railway_station_normcount <= inf else to node 72.
        node=65 is a split node with value=[anders: 0.03, auto: 0.154, bus: 0.053, fiets: 0.036, metro:
            0.207, tram: 0.024, trein: 0.485, voet: 0.012]: go to node 66 if bus_route_std_distance <=
            8919.11767578125 else to node 69.
                node=66 is a split node with value=[anders: 0.023, auto: 0.138, bus: 0.092, fiets: 0.046,
                    metro: 0.402, tram: 0.034, trein: 0.241, voet: 0.023]: go to node 67 if
                    proportion_80_120 <= 0.017564226873219013 else to node 68.
                        node=67 is a leaf node with values=[auto: 0.114, bus: 0.136, fiets: 0.045, metro:
                            0.614, tram: 0.068, trein: 0.023].
                        node=68 is a leaf node with values=[anders: 0.047, auto: 0.163, bus: 0.047, fiets:
                            0.047, metro: 0.186, trein: 0.465, voet: 0.047].
                node=69 is a split node with value=[anders: 0.037, auto: 0.171, bus: 0.012, fiets: 0.024,
                    tram: 0.012, trein: 0.744]: go to node 70 if jerk_iqr_value <= 47285052.0 else to node
                    71.
                        node=70 is a leaf node with values=[anders: 0.086, auto: 0.343, trein: 0.571].
                        node=71 is a leaf node with values=[auto: 0.043, bus: 0.021, fiets: 0.043, tram:
                            0.021, trein: 0.872].
        node=72 is a split node with value=[anders: 0.058, auto: 0.785, bus: 0.036, fiets: 0.09, tram:
            0.002, trein: 0.01, voet: 0.018]: go to node 73 if speed_average <= 23.857415199279785 else to
            node 76.
                node=73 is a split node with value=[anders: 0.014, auto: 0.274, bus: 0.041, fiets: 0.589,
                    trein: 0.027, voet: 0.055]: go to node 74 if accuracy_percentile_95 <= 16.21677875518799
                     else to node 75.
                        node=74 is a leaf node with values=[anders: 0.021, auto: 0.106, bus: 0.043, fiets:
                            0.745, voet: 0.085].
```

```
77                                  node=75 is a leaf node with values=[auto: 0.577, bus: 0.038, fiets: 0.308, trein:
                                        0.077].
78                              node=76 is a split node with value=[anders: 0.061, auto: 0.826, bus: 0.036, fiets: 0.05,
                                    tram: 0.002, trein: 0.009, voet: 0.015]: go to node 77 if proportion_5_15 <=
                                    0.12189747020602226 else to node 86.
79                                  node=77 is a split node with value=[anders: 0.062, auto: 0.877, bus: 0.021, fiets:
                                        0.025, trein: 0.005, voet: 0.01]: go to node 78 if speed_percentile_90 <=
                                        43.59307289123535 else to node 79.
80                                      node=78 is a leaf node with values=[auto: 0.286, fiets: 0.714].
81                                      node=79 is a split node with value=[anders: 0.064, auto: 0.892, bus: 0.022,
                                            fiets: 0.006, trein: 0.005, voet: 0.01]: go to node 80 if
                                            bus_route_max_distance <= 45133.2890625 else to node 83.
82                                          node=80 is a split node with value=[anders: 0.037, auto: 0.916, bus
                                                : 0.029, fiets: 0.007, trein: 0.003, voet: 0.008]: go to node 81
                                                 if proportion_30_50 <= 0.0958034060895443 else to node 82.
83                                              node=81 is a leaf node with values=[auto: 0.838, bus:
                                                    0.068, fiets: 0.054, trein: 0.027, voet: 0.014].
84                                              node=82 is a leaf node with values=[anders: 0.042, auto:
                                                    0.927, bus: 0.023, voet: 0.008].
85                                          node=83 is a split node with value=[anders: 0.151, auto: 0.816,
                                                fiets: 0.005, trein: 0.011, voet: 0.016]: go to node 84 if
                                                speed_median_value <= 83.79621505737305 else to node 85.
86                                              node=84 is a leaf node with values=[anders: 0.29, auto:
                                                    0.699, voet: 0.011].
87                                              node=85 is a leaf node with values=[anders: 0.011, auto:
                                                    0.935, fiets: 0.011, trein: 0.022, voet: 0.022].
88                                  node=86 is a split node with value=[anders: 0.053, auto: 0.469, bus: 0.142, fiets:
                                        0.23, tram: 0.018, trein: 0.035, voet: 0.053]: go to node 87 if
                                        tram_route_std_distance <= 25382.951171875 else to node 88.
89                                      node=87 is a leaf node with values=[anders: 0.034, auto: 0.138, bus: 0.276,
                                            fiets: 0.345, tram: 0.069, trein: 0.138].
90                                      node=88 is a split node with value=[anders: 0.06, auto: 0.583, bus: 0.095,
                                            fiets: 0.19, voet: 0.071]: go to node 89 if speed_percentile_90 <=
                                            34.240671157836914 else to node 90.
91                                          node=89 is a leaf node with values=[fiets: 0.75, voet: 0.25].
92                                          node=90 is a split node with value=[anders: 0.069, auto: 0.681, bus
                                                : 0.111, fiets: 0.097, voet: 0.042]: go to node 91 if
                                                jerk_percentile_15 <= -90295644.0 else to node 92.
```

```
93                                                    node=91 is a leaf node with values=[auto: 0.333, bus:
                                                          0.083, fiets: 0.5, voet: 0.083].
94                                                    node=92 is a leaf node with values=[anders: 0.083, auto:
                                                          0.75, bus: 0.117, fiets: 0.017, voet: 0.033].
```

Listing 1: Python code for decision-tree based transport mode prediction

4.5 Evaluation metrics

The algorithm's performance will be assessed using precision, recall, F1-score, accuracy, and balanced accuracy, metrics commonly used in transport mode classification. Precision evaluates the model's ability to minimize false positives, while recall measures its ability to capture true positives. The F1-score combines both metrics to provide a balanced evaluation, particularly useful for imbalanced datasets. Accuracy represents the overall correctness of predictions but can be misleading in imbalanced data, where balanced accuracy offers an evaluation by averaging recall across all classes. Key definitions include true positive (TP), when the model correctly predicts the actual class (e.g., predicting 'walking' when correct), false positive (FP), where an incorrect class is predicted (e.g., predicting 'car' instead of 'bike'), false negative (FN), when the correct class is missed, and true negative (TN), when incorrect classes are correctly excluded. The formulas for each metric are:

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

$$\text{F1-Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}$$

$$\text{Balanced Accuracy} = \frac{1}{2}\left(\frac{\text{TP}}{\text{TP} + \text{FN}} + \frac{\text{TN}}{\text{TN} + \text{FP}}\right)$$

5 Results

First, the results based on the development data (see Section 3.1) are described. Second, the results based on the open geo-data (see Section 3.2) are described.

5.1 Evaluation on development set

The results from the confusion matrix in Table 7 and classification report in Table 8 for the training data indicate the following key findings: The confusion matrix reveals that the model performs well in predicting car and walking but struggles with categories like bus and tram, where most instances are misclassified. Bike also shows a strong prediction rate, though some misclassifications occur with cars and walking. The 'Other' category is highly misclassified, with many instances incorrectly labeled as car or bike, indicating potential difficulties distinguishing less common transport modes.

Table 7: Confusion matrix of training data

Observed\Predicted	**Other**	**Car**	**Bus**	**Bike**	**Metro**	**Tram**	**Train**	**Walking**
Other	0	64	0	25	0	0	5	9
Car	0	1161	2	106	5	0	20	39
Bus	0	46	3	15	6	0	3	2
Bike	0	31	0	554	2	0	4	52
Metro	0	4	0	1	27	0	8	5
Tram	0	7	0	7	3	0	1	0
Train	0	10	0	5	1	0	82	1
Walking	0	39	0	48	0	0	2	595

The classification report reveals that the model performs well for high-frequency classes like car and walking, achieving precision, recall, and F1-scores around 0.85–0.87, indicating strong and balanced performance for these categories. Bike and train also show relatively high F1 scores (0.79 and 0.73, respectively), with the train having a high recall (0.83) despite moderate precision. However, the model struggles with underrepresented classes. Other and tram have 0.00 F1-scores, as the model fails to classify these instances correctly. Buses have low performance, with an F1 score of 0.07, mainly due to extremely low recall, meaning most buses are misclassified as other categories (especially cars). Metro performs better, with precision and recall around 0.60–0.61, but still shows room for improvement.

Table 8: Classification report for training data

Class	**Precision**	**Recall**	**F1-Score**	**Support**
Other	0.00	0.00	0.00	103
Car	0.85	0.87	0.86	1,333
Bus	0.60	0.04	0.07	75
Bike	0.73	0.86	0.79	643
Metro	0.61	0.60	0.61	45
Tram	0.00	0.00	0.00	18
Train	0.66	0.83	0.73	99
Walk	0.85	0.87	0.86	684
Accuracy			0.81	3,000
Macro avg.	0.54	0.51	0.49	3,000
Weighted avg.	0.77	0.81	0.78	3,000

Although the overall accuracy is relatively high at 81%, the macro average F1-score of 0.49 and balanced accuracy of 0.51 indicate poor performance in less common classes. The weighted average F1-score of 0.78 is boosted by the well-classified dominant classes, masking the severe misclassification of minority classes. This suggests the model may be biased towards common classes, struggling to capture the nuances of less common transport modes.

The results from the confusion matrix in Table 9 and classification report in Table 10 for the test data indicate the following key findings: The model performs well for high-frequency classes like car, bike, train, and walking, with relatively high precision, recall, and F1-scores. For instance, car has an F1-score of 0.85, and walking achieves 0.84, reflecting consistent performance compared to the training set, where these classes also had high scores. Train maintains strong recall (0.87) and a high F1-score (0.82), showing that the model reliably identifies most train instances. Similarly, bike achieves an F1-score of 0.76, demonstrating the model's ability to generalize reasonably well to this class.

Table 9: Confusion matrix of test data

Observed\Predicted	**Other**	**Car**	**Bus**	**Bike**	**Metro**	**Tram**	**Train**	**Walking**
Other	0	3	0	1	0	0	0	4
Car	0	459	0	67	6	0	9	18
Bus	0	15	0	4	1	0	0	1
Bike	0	12	0	249	6	0	1	35
Metro	0	2	0	1	4	0	5	1
Tram	0	3	0	3	7	0	0	1
Train	0	5	0	2	0	0	54	1
Walking	0	18	0	24	2	0	0	274

However, the confusion matrix highlights various misclassifications, especially for underrepresented classes. For example, 'Other' is never classified correctly, with instances being mistaken for car, bike, or walking – mirroring the training set where 'Other' had an F1-score of 0.00. Bus also performs poorly, with all instances misclassified, mostly as car or bike, leading to a 0.00 F1-score, just like in training data. This suggests the model struggles to learn meaningful patterns for rare classes, likely

because of class imbalance and overlapping features.

The metro and tram classes continue to be problematic. Metro shows a slight improvement over the training set, with a 0.21 F1-score on the test set, but remains low, with many instances misclassified as train. Tram remains entirely misclassified, with an F1-score of 0.00, indicating the model failed to generalize this class from the training set to the test set. These results indicate that minority classes are poorly represented in the decision boundaries, possibly because the model is biased toward more common classes like car and walking.

Table 10: Classification report for test data

Class	**Precision**	**Recall**	**F1-Score**	**Support**
Other	0.00	0.00	0.00	8
Car	0.89	0.82	0.85	559
Bus	0.00	0.00	0.00	21
Bike	0.71	0.82	0.76	303
Metro	0.15	0.31	0.21	13
Tram	0.00	0.00	0.00	14
Train	0.78	0.87	0.82	62
Walk	0.82	0.86	0.84	318
Accuracy			0.80	1,298
Macro avg.	0.42	0.46	0.44	1,298
Weighted avg.	0.79	0.80	0.79	1,298

Despite an overall accuracy of 80%, a macro average F1-score of 0.44 and a balanced accuracy of 0.46 reveal that performance varies widely across classes, with the model performing well on frequent categories but failing on rare ones. The weighted average F1-score of 0.79 is heavily influenced by the well-classified majority classes, masking the poor recognition of smaller classes. The test results confirm the patterns observed in training: the model captures dominant class features well but struggles with minority classes, leading to repeated misclassification patterns across both datasets.

Feature Importance

Table 11 shows the feature importance of the top 20 selected features. In total, 39 were selected. Although the hyperparameter space should have limited the number of features to seven, the final decision tree contains 39 features. This is, because the hyperparameters are not strictly enforced. The feature importance results reveal that the model relies heavily on a few key features, with 'bus route mean distance' as the most influential feature, contributing 22.3% to the decision-making process. This suggests that distance patterns along bus routes play a critical role in distinguishing transport modes. Speed-related features also dominate the model's decisions, with metrics like 'speed IQR' (17.7%), 'speed percentile 10' (5.1%), and 'speed standard deviation' (3.1%) collectively contributing a large share of the importance. This heavy reliance on speed variation could explain the model's struggles with modes with overlapping speed ranges (e.g., bus vs. car or metro vs. train). Interestingly, railway station count (10,4%) is another essential feature, likely helping the model identify train and metro trips. Proportion-based speed features (e.g., proportion 45–80 km/h,

7.9%) also influence predictions, possibly helping differentiate slower modes like walking from faster ones like cycling or driving. Lower-ranked features, like 'jerk percentile 85' (0.9%) and 'tram route standard distance' (0.75%), contribute minimally. Overall, the model leans heavily on speed and distance metrics, explaining its success with frequent modes like cars and bikes and its failures with underrepresented classes. Strengthening the model with more contextual features (see discussion) or refining route-based features for specific transport modes could help improve classification performance, especially for minority classes.

Table 11: Top 20 decision tree feature importance

Feature	Importance
bus_route_mean_distance	0.22
speed_iqr_value	0.18
railway_station_normcount	0.10
proportion_45_80	0.08
speed_percentile_10	0.05
proportion_5_15	0.04
speed_stddev	0.03
speed_average	0.03
speed_percentile_90	0.03
speed_percentile_80	0.02
bus_route_std_distance	0.02
bus_route_max_distance	0.02
proportion_15_30	0.02
speed_percentile_85	0.02
acc_kurt	0.02
accuracy_percentile_85	0.01
jerk_percentile_85	0.01
proportion_80_120	0.01
tram_route_std_distance	0.01
speed_median_value	0.01

5.2 Evaluation on open geo-data

The results from the confusion matrix in Table 12 and classification report in Table 13 for the test on the open geo-data show the following key findings: The confusion matrix reveals considerable misclassification patterns, particularly among specific transport modes. Walking is the most accurately predicted class, with 52 correct classifications, though it is still confused with bike (16) and car (9). Tram shows the highest misclassification rate, frequently being predicted as car (14), bike (7), or metro (5). Bus is rarely identified correctly and is often confused with car, bike, metro, and train. Similarly, the ferry is misclassified as the bike. The label 'ferry' did not appear in the observed labels in the development data. However, this label was kept to study what prediction would result for this label. This is especially interesting, because GPS signals usually get noisy when the smartphone is close to or on the water. Train and metro show moderate accuracy, but car and other categories are never correctly predicted. These results highlight challenges in distinguishing between modes with

similar speed profiles and infrastructure characteristics.

Table 12: Confusion matrix of open geo-data

Observed\Predicted	Other	Car	Bus	Ferry	Bike	Metro	Tram	Train	Walking
Other	0	0	0	0	0	0	0	0	0
Car	0	0	0	0	0	0	0	0	0
Bus	0	2	0	0	1	1	0	1	0
Ferry	0	0	0	0	3	0	0	0	0
Bike	0	2	0	0	7	1	0	0	0
Metro	0	3	0	0	0	1	0	0	1
Tram	0	14	0	0	7	5	0	0	1
Train	0	3	0	0	0	0	0	6	0
Walking	0	9	0	0	16	1	0	0	52

The classification report highlights performance variations across transport modes. Walking and the train achieve the highest F1 scores (0.79 and 0.75, respectively), indicating relatively good performance. The bike also shows moderate recall (0.70) but low precision (0.21), leading to a modest F1-score of 0.32. In contrast, bus, ferry, tram, car, and other categories are never correctly identified, resulting in F1-scores of 0.00. Metro has a low F1-score (0.14) due to poor precision and recall. The overall accuracy of 0.48, the balanced accuracy of 0.32, and the macro F1-score of 0.22 reflect substantial class imbalances and misclassification issues, particularly for underrepresented classes.

Table 13: Classification report for test open geo-data

Class	Precision	Recall	F1-Score	Support
Other	0.00	0.00	0.00	0
Car	0.00	0.00	0.00	0
Bus	0.00	0.00	0.00	5
Ferry	0.00	0.00	0.00	3
Bike	0.21	0.70	0.32	10
Metro	0.11	0.20	0.14	5
Tram	0.00	0.00	0.00	27
Train	0.86	0.67	0.75	9
Walking	0.96	0.67	0.79	78
Accuracy			0.48	137
Macro avg.	0.24	0.25	0.22	137
Weighted avg.	0.62	0.48	0.53	137

6 Discussion

This report describes the development and performance of a transport mode classification algorithm for smart surveys. A decision-tree-based algorithm was developed. The results show that the decision tree model achieves a reasonable overall accuracy of 81% (balanced accuracy 51%) on the training data and 80% on the test set from the development data (balanced accuracy 46%), with a weighted average F1-score of 0.78

and 0.79, respectively. These results indicate that the model generalizes relatively well, but deeper analysis shows considerable class-specific imbalances and misclassification patterns. The model performs well in identifying high-frequency classes like car, bike, and walk, with consistently high F1 scores. In contrast, classes like bus and tram are poorly classified, with F1 scores of 0.00, suggesting that these classes are either rarely predicted or consistently confused with more dominant classes. When tested on the open geo-data, exclusively reserved for model evaluation, the results indicate that the model struggles with generalizability, particularly for less frequent transport modes. While it performs reasonably well for walking and trains, its failure to correctly classify buses, trams, and cars suggests poor generalization to unseen data. The low macro F1-score and imbalanced precision-recall values highlight a strong bias toward dominant classes, leading to frequent misclassifications. This result suggests the developed algorithm may overfit to patterns in the training data rather than learning robust, generalizable decision rules. The feature importance table highlights that OSM-based distance metrics and GPS-based speed features predominantly drive the decision-making process. This heavy reliance on distance and speed could explain the model's difficulty distinguishing between modes with similar speed ranges and infrastructure patterns. The confusion matrices reveal some systematic misclassifications, such as walking and biking. The other class instances are spread across multiple categories, reflecting this group's lack of distinctive patterns.

A non-nested rule-based algorithm was also developed as part of creating the decision-tree-based algorithm (Fourie et al. expected 2025). This rule-based algorithm achieved an overall accuracy of 85%, a balanced accuracy of 70% when evaluated on the test set from the development data. When tested on the open geo-data collected within the SSI project, an accuracy of 80% and a balanced accuracy of 83% were achieved. These results suggest that the manual rule-based algorithm generalizes better unseen data than the decision-tree-based algorithm developed in this paper. The main differences between the decision tree and the rule-based algorithm are a) single vs. multiple transport mode predictions and b) the number of predicted classes (excluding the class 'Other' and 'Ferry').

Limitations and future work

GPS signal

There are commonly known issues with GPS signals, which vary depending on smartphone and sensor configuration (Gootzen et al. expected 2025). Features about the different GPS measurements had predictive power to classify the transport mode. However, the GPS signals are only helpful to a certain extent by capturing variations in speed, but nearly no other GPS-based feature is important. This is a shortcoming of smartphone sensors. GPS-loggers, for example, are typically more accurate, especially high-end devices with better antennas. The influence of GPS signals on features and prediction quality needs more research in the future.

Features

About 200 features based on GPS and OSM were considered in the development. However, in the current version of the decision tree, only 39 features are important, which are predominantly speed (GPS) and distance metrics (OSM). The decision tree prioritized OSM distance metrics over OSM count variables. This means that counting OSM infrastructure is less sufficient for transport mode prediction. This was

also shown by Fourie et al. (expected 2025), that OSM-based proximity features have more predictive power. More research is required on the quality and coverage of the OSM database. The fact that counts were not chosen could also be due to low and incomplete coverage.

Transport mode labels
The class 'Other' complicated the model training as this might contain a huge variability in transport modes. Without additional features, such a category will always remain challenging to predict. Moreover, there are errors in the labels of the development data as explained by Fourie (2025). Future studies must ensure to collect high-quality labels, especially for the public transport modes as these are most challenging to predict.

Machine learning
The decision tree and rule-based model are straightforward algorithms. Machine learning could also be considered to predict the transport mode. However, the issues encountered during development include, for example, the imbalance of data, rare classes, errors in training labels, and the absence of potentially more relevant features that will remain with this dataset. Especially contextual features, for example, owning a car or (e)bike, subscription for public transportation, the smartphone being logged in to WiFi from public transport, or the smartphone using services such as Apple Carplay or Android Auto, or other smartphone background services, are considered to have potential to improve the performance of the algorithm. No machine learning algorithm alone will solve these problems.

Performance
In the literature, sometimes better algorithm performance is reported. However, several papers only report accuracy, not the F1 or balanced accuracy. This report shows that accuracy alone can lead to over-optimistic conclusions about the algorithm's performance.

Comparability and generalization
For a fair comparison of general algorithm performance, a reference public dataset should be utilized, such as the open geo-dataset collected within the scope of this project. Otherwise, there will always be tailored solutions that do not generalize or are not comparable. The rule-based algorithm shown in Appendix D showed that it generalizes to two countries (The Netherlands and Germany) and to different sensor configurations of the smartphone app, see also Fourie et al. (expected 2025).

7 Conclusion

Transport mode prediction is central to improving the functionalities for smart time-use, travel, and mobility surveys. The decision tree model shows some promising results, but also shows issues of not being able to distinguish between transport modes with similar movement and infrastructure patterns. The high performance of dominant classes (like car and walk) comes at the expense of minority classes (bus, metro, tram), leading to poor recall and precision scores. The developed algorithm lays the groundwork for automatic transport mode prediction, while full automation of this

smart feature remains a future goal.

By establishing this algorithm as a benchmark, the development provides a practical starting point for refining future models. The algorithm's simplicity and speed make it a viable option for real-world implementation, potentially alongside user prompts, to reduce respondent burden without compromising accuracy.

References

Fourie, J. J. (2025). Rules for transport mode determination in smart travel surveys [Master Thesis – Statistics and Data Science University Leiden].

Fourie, J. J., Klingwort, J., & Gootzen, Y. (expected 2025). Rule-based transport mode classification in a smartphone-based travel survey [Discussion paper, Statistics Netherlands].

Gong, H., Chen, C., Bialostozky, E., & Lawson, C. T. (2012). A GPS/GIS method for travel mode detection in New York City. *Computers Environment and Urban Systems*, *36*(2), 131–139.

Gootzen, Y., Klingwort, J., & Schouten, B. (expected 2025). Data quality aspects for location-tracking in smart travel and mobility surveys [Discussion paper, Statistics Netherlands].

Sadeghian, P., Zhao, X., Golshan, A., & Håkansson, J. (2022). A stepwise methodology for transport mode detection in gps tracking data. *Travel Behaviour and Society*, *26*, 159–167.

Schouten, B., Remmerswaal, D., Elevelt, A., de Groot, J., Klingwort, J., Schijvenaars, T., Schulte, M., & Vollebregt, M. (2024). A smart travel survey: Results of a push-to-smart field experiment in the netherlands [Discussion paper, Statistics Netherlands].

Smeets, L., Lugtig, P., & Schouten, B. (2019). Automatic travel mode prediction in a national travel survey.

Appendix A GPS features

We provide some information on features that might not be known to a general audience. Speed is the rate at which an object covers distance. It tells how fast an object is moving but does not specify direction. Acceleration is the rate at which velocity changes over time. It describes how quickly an object's speed or direction of motion changes. Acceleration is a vector quantity, meaning it has both magnitude and direction. Jerk is the rate at which acceleration changes over time. Jerk is also a vector quantity that describes how abruptly an object's acceleration changes. Snap is the rate at which jerk changes over time. Snap is used less frequently but can be important when analyzing systems where smooth motion is essential.

Bearing refers to the direction or angle from one point to another, typically measured clockwise from a reference direction (often true north) to the line connecting two points on the Earth's surface. In GPS applications, bearing is used to specify the direction in which an object or location lies relative to another point.

Altitude refers to the vertical position or height of a point above a reference surface, typically mean sea level.

- **Speed, Acceleration, Jerk, and Snap**
  - Shared features:
    - Mean
    - Median
    - Standard deviation
    - Minimum
    - Maximum
    - Interquartile range
    - Skewness
    - Kurtosis
    - 95, 90, 85, 80, 20, 15, 10, 5 percentile
  - Additional speed features:
    - Proportion at very low speed (0 – 5 km/h)
    - Proportion at low speed (5 – 15 km/h)
    - Proportion at low to medium speed (10 – 30 km/h)
    - Proportion at low to medium speed (15 – 30 km/h)
    - Proportion at medium speed (30 – 50 km/h)
    - Proportion at medium to high speed (45 – 80 km/h)
    - Proportion at high speed (80 – 120 km/h)
    - Proportion at very high speed ($\geq$ 120 km/h)
- **Bearing**
  - Features:
    - Mean
    - Median
    - Standard deviation

        - Minimum
        - Maximum
        - Interquartile range
        - Skewness
        - Kurtosis
        - 95, 90, 85, 80, 20, 15, 10, 5 percentile

- **Altitude**
    - Features:
        - Mean
        - Median
        - Standard deviation
        - Minimum
        - Maximum
        - Interquartile range
        - Skewness
        - Kurtosis
        - 95, 90, 85, 80, 20, 15, 10, 5 percentile
        - Proportion below sea

- **Accuracy**
    - Features:
        - Mean
        - Median
        - Standard deviation
        - Minimum
        - Maximum
        - Interquartile range
        - Skewness
        - Kurtosis
        - 95, 90, 85, 80, 20, 15, 10, 5 percentile

- **Time**
    - Features:
        - Trip Length (seconds)
        - Day of the week
        - Weekend indicator

- **Distance**
    - Features:
        - Trip length (km)

- **GPS Frequency**
  - Features:
    * Number of GPS measurements
    * Mean time between subsequent measurements
    * Median time between subsequent measurements
    * Standard deviation time between subsequent measurements
    * Minimum time between subsequent measurements
    * Maximum time between subsequent measurements
    * Number of long GPS gaps (a period of at least 10 min. without GPS observations)

Appendix B OSM features

- **Infrastructure counts and normalized counts**
  - Features:
    * bus station
    * bus stop
    * railway
    * light rail
    * subway
    * tram
    * busway
    * tram stop
    * railway halt
    * railway station
    * bicycle
- **Route proximity for bus, bike, metro, train, and tram routes**
  - Features:
    * Minimum distance of entire track to route
    * Maximum distance of entire track to route
    * Mean distance of entire track to route
    * Standard deviation distance of entire track to route

Appendix C Python scripts

The Git repository (https://github.com/essnet-ssi/geo-transportmode-prediction-ssi0 contains the following Python scripts that contain the code required to implement the transport mode prediction algorithm.

1. transport_mode_main.py
    - The main script for transport mode prediction that will load and run the other scripts.
2. options.py
    - This script contains all options regarding file paths, data preprocessing and model training required in the other scripts.
3. functions_general.py
    - This script contains general functions required for the transport mode prediction process.
4. gps_features.py
    - Contains functions for gps-based feature creation for events and locations data. These features are added to the events dataframe.
5. osm_features.py
    - Contains functions for osm-based feature creation for events and locations data. These features are added to the events dataframe.
6. train_decision_tree.py
    - This script runs a grid search over a hyperparameter set to train the best decision tree model for the given data. The current best result is 20250424_decision_tree_ssi.pickle.
7. decision_tree_ssi.pickle
    - The best decision tree model for the available development data resulting from the combination of options, feature creation and model training.

Appendix D Rule-based transport mode prediction

The rule-based algorithm presented here was developed by Fourie (2025) and will soon be published, and therefore, we refer to Fourie et al. (expected 2025) for a detailed development description of this rule-based algorithm. Alongside the algorithm, we will present and elaborate the results of this algorithm. The algorithm was developed using the dataset described in Section 3.1. The confusion matrix in Table 14 shows the algorithm's results based on the training data. There are a large number of correct predictions for bike (560), car (825), and walk (573) indicating the algorithm performs well for these categories. Train (80) and metro (29) have relatively lower correct predictions but still show some accuracy. There are some misclassification trends. Bike is often confused with walk (62) and car (28). Car is sometimes misclassified as bike (70) and walk (30). Walk is sometimes misclassified as bike (81) and car (31). Metro and Train are occasionally misclassified as cars. For Buses and trams, the classification accuracy is poor. Bus has no correct predictions (all zeros on the diagonal for that row), meaning it is entirely misclassified. Tram has only 13 correct classifications, frequently misclassified as car (7) or walk (2). In conclusion, the algorithm performs well for bikes, cars, and walks but struggles with buses and trams. Misclassification patterns suggest possible feature overlap between cars, walking, bikes, and between metro and trains.

Table 14: Confusion matrix of training data

Observed\Predicted	**Bike**	**Bus**	**Car**	**Metro**	**Train**	**Tram**	**Walk**
Bike	560	0	28	0	6	0	62
Bus	3	0	6	0	2	0	2
Car	70	1	825	2	8	3	30
Metro	5	0	4	29	1	0	8
Train	3	0	11	2	80	0	5
Tram	0	0	7	0	0	13	2
Walk	81	0	31	2	3	2	573

It was found that some misclassifications for bike, walk, and car were due to wrong labels assigned by the user. This issue was analyzed by Fourie et al. (expected 2025) and found that misclassifications are primarily due to incorrect labeling and data quality issues. Bike trips are misclassified as walking and have an unrealistically low average speed (on average, 4.86 km/h), suggesting labeling errors. Conversely, walks misclassified as bikes have an unusually high average speed (on average, 9.54 km/h) with greater speed variation, indicating possible data inconsistencies. Car misclassifications follow similar trends. Car trips are misclassified as walking and have an average speed of 3.22 km/h, likely due to mislabeling. Car trips misclassified as bikes have an average speed of 13.4 km/h, suggesting low-quality data or user errors.

The test set confusion matrix in Table 15 shows similar misclassification trends as the training set, supporting previous findings. There are persistent misclassifications between bike and walk. Bike is often misclassified as walk (17 instances) and walk

as bike (30 instances), consistent with the training set. This aligns with the previous finding that speed-based classification thresholds may be causing incorrect labeling. Car is misclassified as walk (12) and bike (25), similar to the training set pattern. This suggests difficulty distinguishing low-speed car trips from other modes, possibly due to labeling errors or data quality issues. Just like in the training set, bus has zero correct classifications, meaning the model struggles to recognize this mode entirely. There is improved performance for the metro, train, and tram. These categories had limited correct classifications in the training set but showed slightly improved results in the test set. However, some misclassification persists, particularly train being confused with car (8 instances). The algorithm struggles with low-speed distinctions, particularly bike vs. walk and car vs. walk/bike. Bus classification remains a major issue that needs further investigation. The slight improvement in metro, train, and tram suggests some learning transfer but still room for optimization.

Table 15: Confusion matrix of test data

Observed\Predicted	**Bike**	**Bus**	**Car**	**Metro**	**Train**	**Tram**	**Walk**
Bike	243	0	24	0	0	0	17
Bus	2	0	8	0	0	0	0
Car	25	0	360	2	2	2	12
Metro	0	0	1	5	1	0	0
Train	1	0	8	0	32	1	1
Tram	0	0	1	0	0	9	1
Walk	30	1	11	0	1	1	247

Table 16 shows the classification report of the algorithm based on the training set. The algorithm achieves 84% accuracy, indicating good general performance. However, balanced accuracy is much lower (65%), suggesting poor performance on underrepresented classes. There is a strong predictive Performance for the majority classes. Car (F1-score: 0.89), Walk (0.84), and Bike (0.81) are well classified with high precision and recall. These categories have the highest support (sample count), contributing to strong performance. There are severe issues with Bus classifications. Bus has a precision, recall, and F1-score of 0.0, meaning the model fails completely in identifying bus trips. This aligns with the confusion matrix, where bus instances were entirely misclassified. There is moderate performance for Metro, Train, and Tram. Metro (F1: 0.71) and Train (F1: 0.80) show acceptable performance, though metro has a lower recall (0.62), indicating missed detections. Tram has the weakest performance (F1: 0.65) among the non-bus classes, likely due to its small sample size (22). The key takeaways are, that the algorithm performs well for high-frequency classes (Car, Walk, Bike). THe bus classification is completely ineffective. Metro, Train, and Tram need improvement, likely due to lower support and feature overlap. Balanced accuracy (65%) suggests the model struggles with minority classes.

Table 17 shows the classification report of the algorithm based on the training set. The model achieves 85% accuracy, slightly higher than in the train set (84%). Balanced accuracy improves to 70% (from 65%), indicating better handling of class imbalances but still showing weaknesses. There are consistently strong classifications

Table 16: Classification report for training data

Class	**Precision**	**Recall**	**F1-Score**	**Support**
Bike	0.78	0.85	0.81	656
Bus	0.00	0.00	0.00	13
Car	0.90	0.88	0.89	939
Metro	0.83	0.62	0.71	47
Train	0.80	0.79	0.80	101
Tram	0.72	0.59	0.65	22
Walk	0.84	0.83	0.84	692
Accuracy		0.84		2470
Balanced Accuracy		0.65		2470

for the majority classes: Car (F1-score: 0.88), Walk (0.87), and Bike (0.83) perform well, similar to the training set. Precision and recall are stable across both datasets, suggesting the model generalizes well for these major classes. Bus classification still fails. This confirms that the model cannot recognize and misclassifies bus trips entirely. There are slight improvements for minority classes: Tram (F1: 0.75) improves from 0.65 in the train set, showing better recall (0.82 vs. 0.59). Metro (F1: 0.71) now has balanced precision and recall, unlike in the training set where recall was lower. Train (F1: 0.81) has similar performance but a recall drop (0.74 vs. 0.79), meaning some train trips are still misclassified. The key takeaways and comparison to the train set are that major classes (Bike, Car, Walk) maintain high performance. Bus classification failure persists. Minor classes (Metro, Train, Tram) show slight improvements, especially Tram. Balanced accuracy improves (70% vs. 65%), indicating slightly better recognition of underrepresented classes. The model still struggles with class imbalances and distinguishing low-speed modes.

Table 17: Classification report for test data

Class	**Precision**	**Recall**	**F1-Score**	**Support**
Bike	0.81	0.86	0.83	284
Bus	0.00	0.00	0.00	10
Car	0.87	0.89	0.88	403
Metro	0.71	0.71	0.71	7
Train	0.89	0.74	0.81	43
Tram	0.69	0.82	0.75	11
Walk	0.89	0.85	0.87	291
Accuracy		0.85		1049
Balanced Accuracy		0.70		1049

The algorithm allowed for multiple classifications for bus and car. This was done since it was challenging to distinguish between these two modes, even with the inclusion of OSM data. It also allowed the classification to be unknown. In the results above, it was found that the bus classification failed. It was found that Bus instances were classified as multiple classifications. This is shown in Tables 18 and 19. When the classification was (Car, Bus) it was also usually a Bus or Car.

Table 18: Multiple classifications in training data

Class Combination	Bike	Bus	Car	Metro	Train	Tram	Walk
(car, bus)	9	39	347	1	3	0	17
(unknown)	0	0	2	0	7	0	0

Table 19: Multiple classifications in test data

Class Combination	Bike	Bus	Car	Metro	Train	Tram	Walk
(car, bus)	5	33	167	0	2	0	7
(unknown)	0	0	0	0	0	0	1

The algorithm was also tested on the open geo-dataset described in Section 3.2. The results are shown in Table 20. Here, the label 'Ferry' was excluded. The model achieves 80% accuracy. Balanced accuracy (83%) is higher than in previous results, indicating improved performance across all classes, even those with fewer samples. Bike has perfect recall but very low precision, leading to a weak F1-score. This suggests the model overclassifies instances as Bike, resulting in many false positives. Bus and metro perform well, though metro has lower recall, meaning some metro trips are missed. Train and tram both show strong performance, with high precision and recall. Walk has high precision but lower recall, meaning some walking trips are misclassified.

Table 20: Classification report for open geo-data

Class	Precision	Recall	F1-Score	Support
Bike	0.36	1.00	0.53	9
Bus	1.00	0.80	0.89	5
Metro	1.00	0.60	0.75	5
Train	0.89	0.89	0.89	9
Tram	0.77	0.91	0.83	22
Walk	0.97	0.75	0.85	77
Accuracy		0.80		127
Balanced Accuracy		0.83		127

Table 21 shows the multiple classifications in the open geo-dataset. Here, no multiple classifications were observed because of the absence of car trips in the data. The 'unknown' category, where the algorithm failed to classify a trip confidently, only occurred very few times. Thus, 127 out of the 134 tracks were classified.

Table 21: Multiple classifications in open geo-dataset

Predicted/Observed	Bike	Bus	Metro	Train	Tram	Walk
unknown	1	0	0	0	5	1

```python
def ALG(df):
    def apply_classification(row):
        modes = []
        # Walking
        if (row['speed_p95'] < 13):
            modes.append('walk')
        else:
            # Bike
            if (13 < row['speed_p95'] < 30):
                modes.append('bike')
            else:
                # Tram
                if(row['min_distance_tram']<0.5 and row['
                    std_distance_tram']<250 or row['mean_distance_tram'
                    ]<100):
                    modes.append('tram')
                else:
                    # Metro
                    if(row['min_distance_metro']<1.5 and row['
                        std_distance_metro']<450 or row['
                        mean_distance_metro']<100):
                        modes.append('metro')
                    else:
                        # Train
                        if (row['min_distance_train']<0.05 or row['
                            std_distance_train']<25 or row['
                            mean_distance_train']<100):
                           modes.append('train')
                        else:
                            # Bus
                            if(row['std_distance_bus']<120 or row['
                                mean_distance_bus']<40 or row['
                                min_distance_bus']<0.015):
                                modes.append('bus')
                            # Car
                            if (30 < row['speed_p95'] < 140):
                                modes.append('car')
        return modes if modes else ['unknown']
    df['modes'] = df.apply(apply_classification, axis=1)
    return df
```

Listing 2: Python code for rule-based transport mode classification