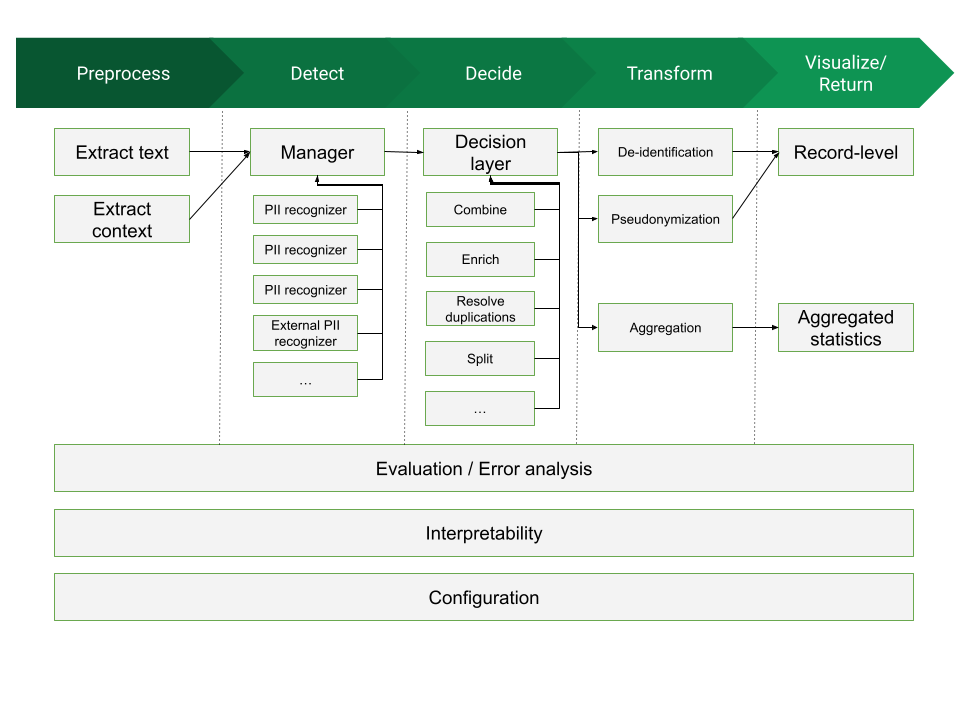
PII data specification  
version 0.3.0

# 1. Overall architecture

The general structure of a framework dealing with PII management could be visualized as the following diagram:



There are up to four processing blocks for such a framework:

1. **Preprocess**: block whose mission is to read a document in an arbitrary format (a Word Document, a Web page, a PDF file, etc) and produce a normalized version, retaining only a simplified version of the high-level structure and all the text data.
2. **Detect**: block in charge of processing input data (usually in text format) and performing detection of candidates to be assigned as PII data. This block uses as input:

* **source document:** we will consider a normalized data format that conveys the raw text contents, together with some structural information. (which can provide useful hints to the PII Detection modules about the relations between text chunks)
* **configuration information**: specification of contextual elements affecting detection (e.g. text language, applicable countries, etc)
* **component information**: the set of available PII Detectors that can be used (assuming we take a modular approach, there might be a database of “pluggable modules” we can use for PII detection).

1. **Decide**: block that takes a number of PII candidates, as produced by the Detection block, and consolidates that information, producing as final result the set of PII elements in the text that need to be addressed. In the process it might combine PII candidates, choose among overlapping PII candidates, reject others, etc. T
2. **Transform**. This is the block that takes the decided PII entities, and acts upon them, depending on the intended purpose.

There can be different Transformation blocks, all of them sharing the same interface but providing different outcomes. Some examples are:

* *Anonymization*: modify the text to eliminate decided PII entities. Depending on options they can be replaced by placeholders, dummy values, generated fake PII data, etc
* *Analytics*: provide the capability to extract and visualize aggregated statistics on decided PII and their associated parameters

# 2. Specification interfaces

The main interfaces to be specified are those that act as boundaries between architecture blocks:

* interface between preprocessing and detection
* interface between detection and decision
* interface between decision and transformation

It might be possible to also define some interfaces internal to one block, so that the block can be decomposed into modular elements (e.g. for pluggable detectors inside the Decision block)

# 3. Specification type

At any given interface, we can envision three types of specification:

1. A **data specification**: syntax & semantics of the data structures that will be sent through one of the interfaces
2. A **program specification**: programmatic interfaces to let components call or be called across the interfaces. This would need to fix an initial default programming language (e.g. Python) to be able to instantiate such programmatic interface; additional languages might be defined later
3. An **API specification**, as a programming language-independent way of interchanging data information across interfaces. This would use a definition such as an [OpenAPI](https://spec.openapis.org/oas/latest.html) specification so that it can be applicable regardless of the programming language; to this aim the *data specification* would be instantiated into a JSON schema or similar

# 5. Data Specification

The data formats for the Detection block will be:

* as *input*, a source document, either generated directly or via the Preprocessing block
* as *output*, a PII collection

## 5.1 Source document

We need to balance two conflicting requirements

1. an easy format to work with: it needs to be machine-processable but also amenable to human editing and reading
2. an expressive format: able to reflect (at least to some level) the document structure, since that structure might be important to connect PII elements

### 5.1.1 Hierarchical source document

In this document format we are trying to preserve two main structural relations between text chunks:

1. an “*is-contained-in*” relation: a text chunk can be considered as semantically contained within another chunk
2. an “*is-next-to*” relation: a text chunk has a relation of being after or before another text chunk

These two relationships can be nested and combined at will. They alone can be enough to describe many of the links that we could need to establish between text chunks (not all of them, but hopefully enough for PII determination).

### 5.1.2 Tabular source document

A Tabular document is one in which the structure has two dimensions, i.e. it is organized (or mostly organized) as rows and columns (so it can be mapped to a table), which contain some type of data, Its semantic premise is that there are some kind of relationships along rows and along colums, relationships that may have implications in terms of PII detection and required processing approaches. Examples of tabular documents are Excel files, or CSV files.

The document header contains some metadata:

* a document id
* context information (optional), which contains three subelements:
  + document context: contains relevant information describing this document, and it has an uspecified form: it might be a metadata dictionary, or just a text string. It is assumed that it
  + a dataset context, similar to the document context but describing the whole dataset this document belongs to

### 5.1.3 Sequential source document

This is a simpler document model in which the document is divided into independent chunks, each one with three elements:

* **id**: an arbitrary string that should be unique per document. Its mission is to make it easier later on to map detected PII instances to the chunk they are part of
* **text**: a text section that contains the textual contents of the chunk. It will be a string containing UTF-8 raw text. It can contain newlines or blank lines, to be considered as part of the text structure, but no formatting or layout contents (it is assumed that exact formatting & layout is lost when creating the source)

The way in which the context is extracted, and therefore its semantics, is not part of the specification; processing modules must only assume that it is related somehow to the current chunk.

## 5.2 PII Collection

A PII collection is the result of running a set of PII detectors on a source document. This result takes the form of a header + a list of detected PII instances.

### 5.2.1 Header

The header contains generic metadata that affects all the PII instances in the collection. Elements of this metadata are:

* date: a timestamp on when the process was run
* format: a string indicating the format the data is in. E.g. pii:pii-collection

### 5.2.2 PII instance

A PII instance describes one recognized PII entity. It can be considered as a dictionary containing three types of information.

### 5.2.3 File format

When formatting a PII Collection for storage or transmission, any format capable of preserving its structure can be used. For the ease of compatibility with most REST-type interfaces, two formats can be proposed:

* **full format**, for storage or local processing: contains the PII Collection as one single JSON object with two subobjects: metadata and piiList
* **streaming format**: it uses NDJSON (aka JSONL) to send the data as separate, newline-delimited chunk

**Full format**

{

"metadata": {

"date": "2022-05-18T15:00:01+00",

"format": "pii:pii-collection",

"format\_version": "0.0.1",

}

}