SN Computer Science (2023) 4:833
https://doi.org/10.1007/s42979-023-02256-4

ORIGINAL RESEARCH

A Distributed Microservice Architecture Pattern for the Automated Generation of Information Extraction Pipelines

Michael Sildatke[1] · Hendrik Karwanni[1] · Bodo Kraft[1] · Albert Zündorf[2]

Received: 1 November 2022 / Accepted: 18 August 2023

Abstract

Companies often build their businesses based on product information and therefore try to automate the process of information extraction (IE). Since the information source is usually heterogeneous and non-standardized, classic extract, transform, load techniques reach their limits. Hence, companies must implement the newest findings from research to tackle the challenges of process automation. They require a flexible and robust system that is extendable and ensures the optimal processing of the different document types. This paper provides a distributed microservice architecture pattern that enables the automated generation of IE pipelines. Since their optimal design is individual for each input document, the system ensures the ad-hoc generation of pipelines depending on specific document characteristics at runtime. Furthermore, it introduces the automated quality determination of each available pipeline and controls the integration of new microservices based on their impact on the business value. The introduced system enables fast prototyping of the newest approaches from research and supports companies in automating their IE processes. Based on the automated quality determination, it ensures that the generated pipelines always meet defined business requirements when they come into productive use.

Keywords Architectural design · Model-driven software engineering · Software and systems modeling · Enterprise information systems · Information extraction · Document classification · Feature detection · Software metrics and measurement

Introduction

Product information often builds the basis for the businesses of modern companies, e.g., comparison portals that offer specific product rankings based on customer preferences. Therefore, one crucial part of the companies' daily business is extracting relevant information. Information Extraction (IE) involves extracting structured information from unstructured sources [1]. To achieve maximum success, the automation of the IE processes is a major business goal.

Product providers often publish information as PDF documents in which tables contain relevant information like prices or contract periods. Since these providers try to address potential customers individually, the documents vary enormously in content and format. Hence, the source of required product information is very heterogeneous, and classic extract, transform, load (ETL) techniques are not sufficiently suitable for process automation. Companies must adopt the newest research approaches to tackle these challenges and gather structured data automatically.

The heterogeneous data basis, the possible upcoming of entirely new formats and the complexity of concrete IE tasks require a vast number of different solution strategies. Furthermore, the continuous improvement of techniques based on new findings from research creates a need for a very flexible and emergent architecture. In table analysis, e.g., several hundred approaches were introduced in 2019 [2]. As a result,

This article is part of the topical collection "Advances on Enterprise Information Systems" guest edited by Michal Smialek, Slimane Hammoudi, Alexander Brodsky and Joaquim Filipe.

✉ Michael Sildatke
michael.sildatke@fh-aachen.de

✉ Hendrik Karwanni
hendrik.karwanni@fh-aachen.de

Bodo Kraft
kraft@fh-aachen.de

Albert Zündorf
zuendorf@uni-kassel.de

[1] FH Aachen, University of Applied Sciences, Aachen, Germany

[2] University of Kassel, Kassel, Germany

Published online: 28 October 2023

companies frequently have to adjust their strategies to react to critical environmental changes.

Identifying the optimal solution strategy and generating the corresponding IE pipeline depends on the specific input document. Any table-based solution strategy will not create appropriate results if there is an input document that does not contain tables. Hence, opportunity costs may arise if the concrete IE process does not consider particular document characteristics.

Such complex and rapidly changing environments require flexible software architectures. Systems that enable evolution by adding code rather than changing existing code ensure the adaption of new situations [3]. Microservice architectures (MSA) are scalable, easy to maintain and extendable [4]. Due to that, they can be used to implement emergent and flexible software systems [5]. Furthermore, agile development approaches like extreme programming or scrum allow fast prototyping and are suitable for creating proofs of concept quickly [6, 7].

In this paper, we extend our previous work presented in Refs. [8, 9] by integrating performance optimizations for the auto-configuration of our system, i.e. caching of intermediate frequently requested component results. Furthermore, we make the system more flexible by replacing the three rigid interfaces `Converter`, `Decomposer` and `Extractor` with a generic one. This reduces the number of components required and thus the effort involved in development. We propose a distributed MSA pattern for the automated generation of IE pipelines and provide a prototype implementation in the German energy industry domain to prove its applicability. The main focus of this architectural pattern is supporting service providers in automating their complex IE processes to avoid opportunity costs and gain business value.

The paper is structured as follows: "Motivation" describes the motivational use case for the introduced system. "Related Work" describes related work. "Concept Definitions" defines the main concepts and "Architectural Pattern" introduces the architectural pattern. "Experimental Evaluation" presents the results of the experimental evaluation while "Conclusion" concludes the paper.

Motivation

The provided pattern is domain-independent and adjustable to different use cases because of its microservice-based architecture and the corresponding generic concepts. However, it is especially suitable in situations lacking document standards. This section introduces a concrete use case in the German energy industry as a motivating example.

Although electricity and gas are commodities, about 3150 different suppliers offer more than 15,000 products in Germany. Usually, energy suppliers customize their products several times per year and publish about 25,000 documents containing relevant information [10]. Since there is no document standard, each supplier uses its own custom format, which may change over time.

Specialized service providers like ene't GmbH[1] or verivox[2] collect information about these suppliers and products to offer services for energy customers, e.g., price comparison portals. For this purpose, employees extract information from heterogeneous and non-machine readable PDF documents by hand. Figure 1 shows the simplified result of a typical IE process.

The following steps are typical for the manual domain-specific IE process and are also shown in Fig. 2:

- **Identifying the related supplier base data record:** Each product is related to a supplier. An extractor will have to create a record containing the supplier's base information if there is no existing one.
- **Identifying the number of products:** Documents can describe several products. Therefore, the extractor has to identify the number of products and relate each piece of information to the correct product.
- **Identifying relevant document parts:** Particular document parts contain relevant information. The extractor must identify these parts before extracting the relevant information.
- **Understanding table semantics:** Product information often is part of tables. The extractor must understand the tables' semantic meaning to extract the information correctly.
- **Understanding text semantics:** Information can be part of natural text. The extractor has to understand the context to capture relevant information.
- **Inferring non-explicit information:** If a specific content is not present, extractors will have to infer non-explicit information. Figure 1 shows an example: if there is no explicit limit B, its value will be 100,000.

The heterogeneous data basis and the complexity of the IE tasks make process automation very difficult. Furthermore, if there are suitable solution strategies for specific document types, missing knowledge about the type of an unknown input document will complicate the identification of the optimal solution strategy. The following document characteristics, also called features, can heavily influence the optimal way of processing:

- **Presence or absence of price tables:** If the document contains tables that store relevant information, pipelines

[1] https://www.enet.eu/.

[2] https://www.verivox.de/.

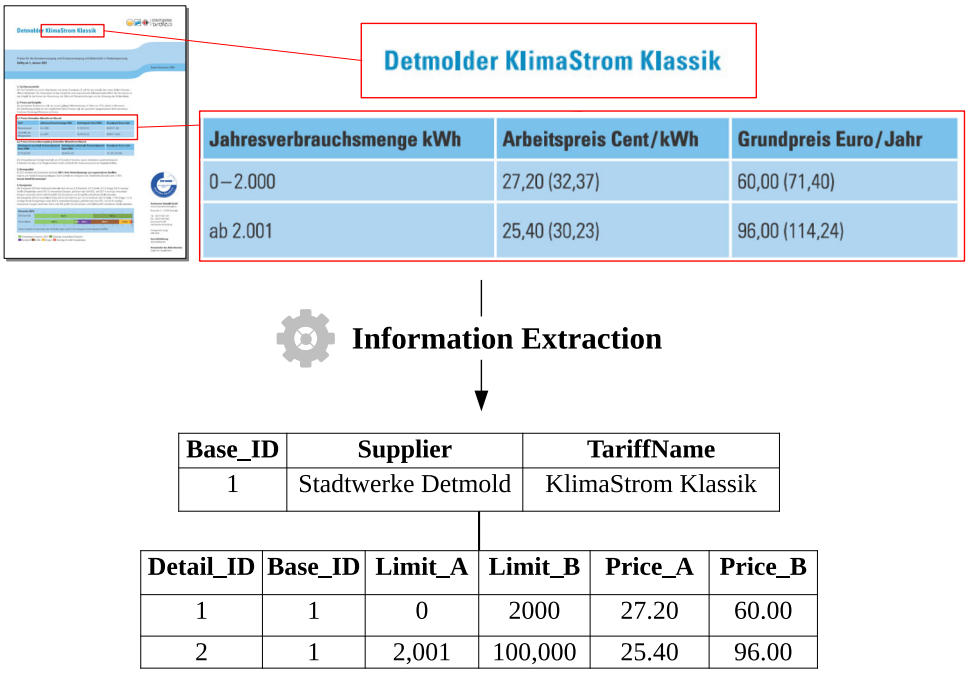

Fig. 1 Structured data as result of an information extraction from a PDF document [8]

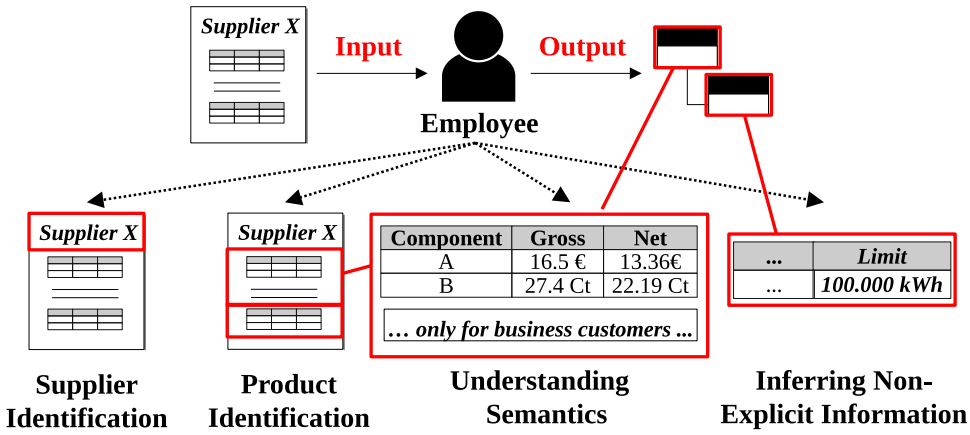

Fig. 2 Steps of the manual information extraction process

will have to be table-based, i.e., integrate specific table decomposers. In case of table absence, pipelines have to be text-based.

- **Scanned or screenshot documents:** Usually, the input of IE processes are scanned or screenshot documents. Pipelines will have to integrate specific pre-processing components, e.g., a text recognition.
- **Page segmentation:** If documents have different columns to present information, e.g., booklets or flyers, pipelines will have to integrate specific segmentation decomposers to keep the correct ordering.
- **Price representations:** Documents can present prices in different representations. While gross and net prices are typical in every domain, there can be more domain-specific representations, e.g., net excluding transportation. The result of the IE process has to contain such prices only once.
- **Consumption-based prices:** Products can have consumption-based prices. If customers consume 2.500 kWh per year, e.g., they will have to pay 25.45 ct/kWh. If they consume more than 2.500 kWh per year, they will have to pay 26.34 ct/kWh. In these cases, the result of an IE process must contain additional information.
- **Time-variable prices:** Products can have time-variable prices. The price for a consumed kWh can be 26.34 cents between 06:00 and 22:00, and 25.54 cents between 22:00 and 6:00. This requires additional extraction steps.
- **Presence or absence of previous prices:** Sometimes, suppliers provide prices from the product history to present a price development. Since these prices are irrelevant, extractors have to ignore them during extraction.
- **Presence or absence of regional limits:** Suppliers can limit their products to specific regions like cities or

streets. Additional extraction steps are required to gather this information.

The automation of the IE processes requires an architecture that, on the one hand, ensures the fast prototyping of new approaches and dynamic variation of strategies. On the other hand, it must enable the automated identification and generation of optimal document-specific pipelines. Hence, an architecture combining these features can support service providers in automating their time-consuming and expensive manual IE processes to gain business value.

Related Work

Despite all benefits of MSA, some challenges exist using them as an architectural basis. Since microservices are autonomous, building dependable systems is challenging. The meaning of specifications needed for the service composition may differ for the underlying technologies [11]. Failing compositions lead to more complexity and unexpected runtime errors [12]. Furthermore, verifying microservice functionalities is challenging [13].

Enterprise integration patterns (EIP) provide approaches for software integration from a theoretical point of view [14]. Frameworks like Apache Camel or Spring Integration bring these approaches into practice providing concrete implementations [15, 16]. Richardson provides microservice patterns to transfer the ideas of EIP into MSAs [17].

Orchestration and choreography are two approaches addressing the challenge of service composition [18]. While the basis for orchestration is a centralized unit that controls the communication between composed microservices, choreography offers a decentralized communication using events. Our provided pattern uses orchestration since one centralized component identifies the optimal solution and generates the corresponding pipeline by composing microservices.

Based on provided functionalities or technical criteria, service discovery enables the automatic detection of services [17]. In contrast to existing approaches, the provided pattern optimizes the service composition based on functional criteria, i.e., the extraction quality.

Fowler points out that software metrics should always link to business goals [19]. Schreiber et al. provide an approach that focuses on business-specific metrics to measure the software quality of research prototypes [20]. Schmidts extended the work of Schreiber by introducing the automated containerization of research prototypes [21]. However, these approaches still require manual management effort and do not solve the problem of automated decision-making for the productive use of prototypes. Hence, they are not suitable for automated pipeline generation in production.

ETL techniques are rule-based and support information gathering from structured formats, e.g. CSV, XML, or JSON files [22]. However, since the required information is not available in a structured way, it is necessary to extract the information with an IE application.

IE applications often try to solve non-deterministic problems. Since, for these problems, boundary conditions may be unstable, Seidler et al. suggest an approach to make these applications more flexible [23]. Since this approach does not focus on the complete process, it leaves out essential steps like conversion. Furthermore, it is limited to extracting information from text and does not provide functionalities to integrate table processing components.

Ontology-based information extraction (OBIE) systems also focus on the IE from unstructured text [24] and are not designed to process PDF documents [25]. Therefore, in our case, the existing systems are not suitable.

In the field of robotic process automation (RPA), robots minimize human efforts by automating interactions with Graphical User Interfaces (GUI) [26]. Robots adapt rule-based behaviors by observing users interacting with affected systems [27]. Since our focused problems deal with an infinite set of possible document types, those rule-based solution strategies are insufficient.

The provided system must be able to classify input documents to identify and generate the optimal document-based pipeline. Jiang and Lilleberg introduce different algorithms to aim at a feature-based text classification [28, 29]. These approaches work on already-known features. In our case, detecting the features themselves is the key, while the classification is a downstream step.

Concept Definitions

In this section, we define the main concepts of our approach as a basis for the architectural pattern.

Artifact object model: We provide a standardized object model with *Artifact* as its abstract root (c.f. Fig. 3). The model allows the definition of different objects that are part of the IE process, e.g. input documents like *PdfDocuments* or specific document elements like *Tables*. Furthermore it enables the definition of *Information* objects. These objects represent pieces of domain-specific information the system should extract, e.g. *CommodityPrice*.

Document feature: A document feature describes a particular document property, e.g., whether the document contains tables or not.

Gold data: A gold data document combines an already processed input document with its extracted information and document features. These documents serve as a basis to test the system's functionalities. The set of all gold data documents is called gold data (c.f. Fig. 4).

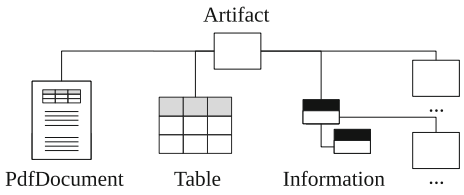

Fig. 3 Object model with artifact as root

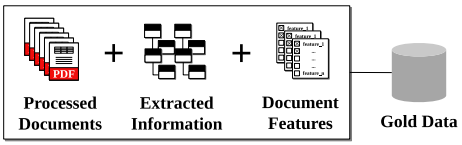

Fig. 4 Contents of gold data [8]

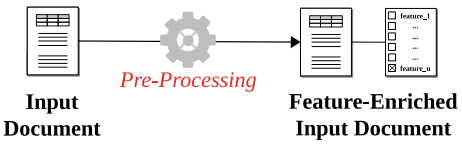

Fig. 5 Input document and feature-enriched input document

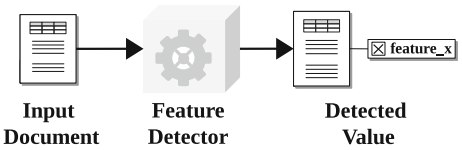

Fig. 6 Input and result of a feature detector

Input document and feature-enriched input document: An input document is unknown and serves as input for the actual IE process. A feature-enriched input document is a pre-processed input document containing a list of detected document features (c.f. Fig. 5).

Feature detector: A feature detector is a software module that detects the value of a specific feature for an unknown input document, e.g. if the document contains a table (c.f. Fig. 6). Combining all detected features from different detectors results in a feature-enriched document.

Component: The IE from heterogeneous documents requires the solution of different tasks, e.g. the conversion from non-machine readable PDF documents into any machine readable representation like plain text, or the detection of tables in PDF documents. A component is a software module that solves exactly one of these specific IE tasks by consuming a specific artifact and producing specific artifacts (c.f. Fig. 7).

Pipeline: A pipeline combines several components. A pipeline is valid if the in- and outputs of these components are consistent, meaning that the output type of the previous component must match the input type of the subsequent component. In our specific case, the input of a pipeline is always a PdfDocument, while the output is always a specific kind of information, e.g. CommodityPrice (c.f. Fig. 8).

Fig. 7 In- and output of a component

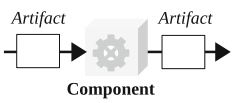

Fig. 8 Construction of a pipeline

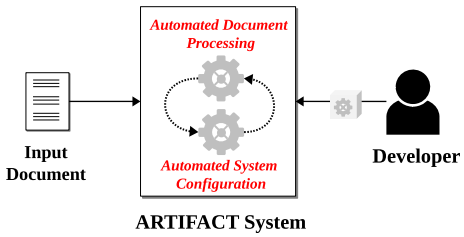

Fig. 9 Core concepts of the architectural pattern

Architectural Pattern

The following section introduces our architectural pattern called ARTIFACT. Figure 9 illustrates its core concepts.

The ARTIFACT system can extract relevant information from unknown input documents through the concept of automated document processing. Developers can push new components to the system to trigger the automated system configuration.

Automated System Configuration

The goal of the automated system configuration is enabling the system to generate and execute the best document-specific pipeline for the IE from an unknown input document. Hence, it is a prerequisite for the automated document processing described in "Automated Document Processing". The process of the automated system configuration includes the gold data-, the feature detection- and the pipeline auto-configuration (c.f. Fig. 10).

Gold Data Auto-Configuration

The set of gold data must always represent real-world boundary conditions. Therefore, the documents of the set have to be balanced. The ratios of document types must correspond to those in reality. Furthermore, conditions may change over time, and the gold data documents must always be up to date to represent all changes. To ensure actuality, we introduce the document manager that frequently updates the set of gold data by deleting old documents and storing new ones

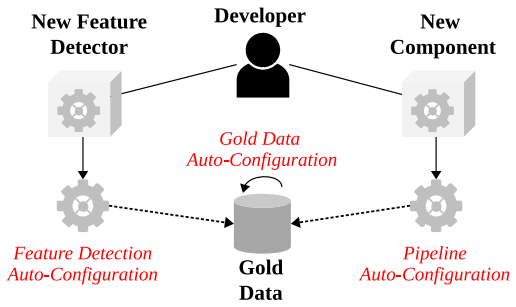

Fig. 10 Overview of the automated system configuration

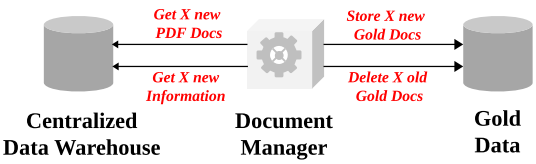

Fig. 11 Document manager keeping gold data up to date [9]

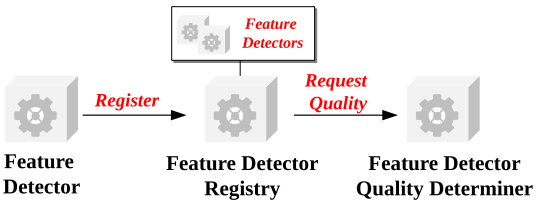

Fig. 12 Registration of a feature detector [9]

from the centralized data warehouse (c.f. Fig. 11). The document manager ensures the automated detection of possible concept drifts.

Feature Detection Auto-Configuration

Developers can deploy updated or new feature detectors to a feature detector registry. This registry manages all available detectors and ensures that a registering detector fulfills the required quality criteria. The registry requests the feature detector quality determiner to achieve this (c.f. Fig. 12).

The feature detector quality determiner tests the specific detector against the set of gold data to determine the quality of a feature detector. As a result, the number of correctly detected values divided by the number of gold data documents determines the quality of the detector. Domain experts or managers can define criteria for each feature that a detector must at least reach to be registered. If the registration is successful, the feature detector registry will inform the mapping generator (c.f. Fig. 13), which is explained in "Pipeline Auto-Configuration".

The introduced concept ensures that the system only takes those features into account that it can reliably detect.

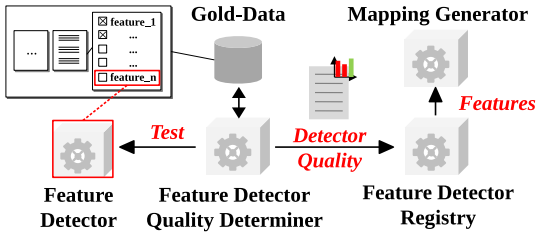

Fig. 13 Quality determination of a feature detector [9]

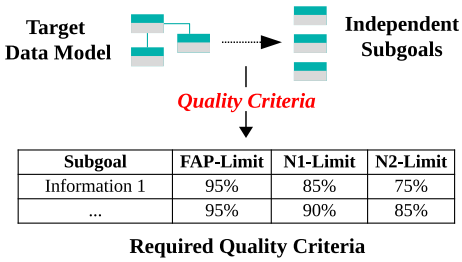

Fig. 14 Defining quality criteria for the goals

Pipeline Auto-Configuration

For business success, the system must generate and execute the optimal pipeline for each input document. To achieve this, we introduce the pipeline auto-configuration concept and provide its building blocks in the following.

Definition of subgoals and quality criteria: Since IE processes are often very complex, our pattern enables the definition of subgoals. A *subgoal* can be an atomic information part of the target data model.

A valid pipeline always results in extracted information defined by a subgoal or the target data model itself. To ensure that the system always meets business requirements in extracting information, domain experts or managers can define different quality criteria (c.f. Fig. 14).

Our system introduces Full Automatic Processing (FAP) of documents. The IE processes often integrate manual approval steps to minimize errors in captured data. In the case of FAP, these manual approval steps are omitted. Accordingly, the *FAP-Limit* defines the percentage of passed tests against the gold data a pipeline has to reach for the FAP of documents. A manual approval step will be required if the system does not reach the FAP-Limit.

The system can use a single pipeline for automated extraction if this pipeline reaches the *N1-Limit* of passed gold data tests. In this case, a subsequent manual approval step is required.

The *N2-Limit* brings pipelines into productive use that perform below the N1-Limit, and which therefore do not work reliable enough. Two different pipelines must at least reach the N2-Limit to confirm each other's results. If those

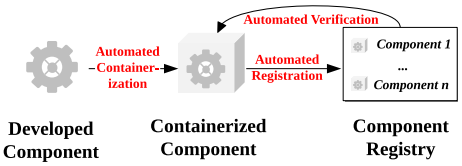

Fig. 15 Component registering at the component registry

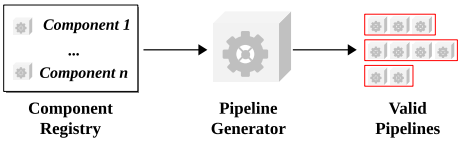

Fig. 16 In- and output of the pipeline generator

results match, the system will accept them for automation with subsequent manual approval.

If the system does not reach any defined limit, an employee will have to extract information by hand before a second employee approves the result. Due to this principle, the system always meets defined business requirements.

Goal-specific pipeline generation: Analogous to the feature detector registry, the system provides a component registry that manages and validates available pipeline components. Developers can implement any component and deploy it to the registry. The registry validates the component by requesting it with dummy data. If the in- and output artifacts match, the component registration was successful (c.f. Fig. 15).

The component registry always informs the pipeline generator microservice about newly registered components. The pipeline generator automatically generates all valid pipelines based on the available components and defined subgoals (c.f. Fig. 16). The pipeline generation is based on backward matching of consumed and produced artifacts.

Pipeline generation algorithm: In the following, we introduce an algorithm that generates all valid pipelines for the given in- and output artifacts. The algorithm is split into a main part and a recursive part. The main part builds the foundation to call the recursive part and returns the overall results. The recursive part generates all pipelines depending on a given intermediate list of ordered components.

Figure 17 shows the main steps of the algorithm that expects three input parameters:

1. A list of allowed input artifact types (*inputArtifacts*)
2. A list of allowed output artifact types, i.e. domain-specific information (*outputArtifacts*)
3. A list of components available for pipeline generation (*components*)

In the main part, the algorithm generates a list of possible starting components that consume any of the defined input artifacts. Additionally, it generates a list of possible ending components that produce any of the defined output artifacts. Afterwards, it iterates over the list of starting components and calls the recursive part for each of them.

Figure 18 shows the recursive steps of the algorithm that expects the following input parameters:

1. A pipeline draft containing already added components (*pipelineDraft*)
2. A list of remaining components not used in the pipeline draft yet (*remainingComponents*)
3. A list of possible ending components (*endingComponents*)

With each call of the recursive part, the algorithm extends an incomplete list of ordered components (*pipelineDraft*) until its last element is a possible and previously determined ending component. The goal of the recursion is to perform a new depth-first search from each last element of a pipeline draft, i.e. a list of ordered components.

In the following, we show the results of our algorithm using an example focusing on three artifacts, namely `PdfDocument`, `TextDocument` and `ProductName`, and four components (c.f. Table 1).

In our example, the input parameters for the main part of the algorithm look as follows:

- *inputArtifacts*: [`PdfDocument`]
- *outputArtifacats*: [`ProductName`]
- *components*: [`PdfToTextC`, `TextPreProc`, `ProductNameEx1`, `ProductNameEx2`]

Following the described steps, the algorithm generates the following pipelines and ensures that each component occurs only once in the list of ordered components:

- `PdfToTextConv` → `TextPreProc` → `ProductNameEx1`
- `PdfToTextConv` → `TextPreProc` → `ProductNameEx2`
- `PdfToTextConv` → `ProductNameEx1`
- `PdfToTextConv` → `ProductNameEx2`

Automated gold data extension: To be able to generate optimal document-specific pipelines, the system has to find appropriate document types. To achieve this, we introduce the automatic gold data extension that forms the basis for finding these document types. The pipeline generator informs the label extender about valid pipelines, which tests each of these pipelines against the set of gold data documents. As a result, it extends each gold data document with a list of suitable pipelines (c.f. Fig. 19).

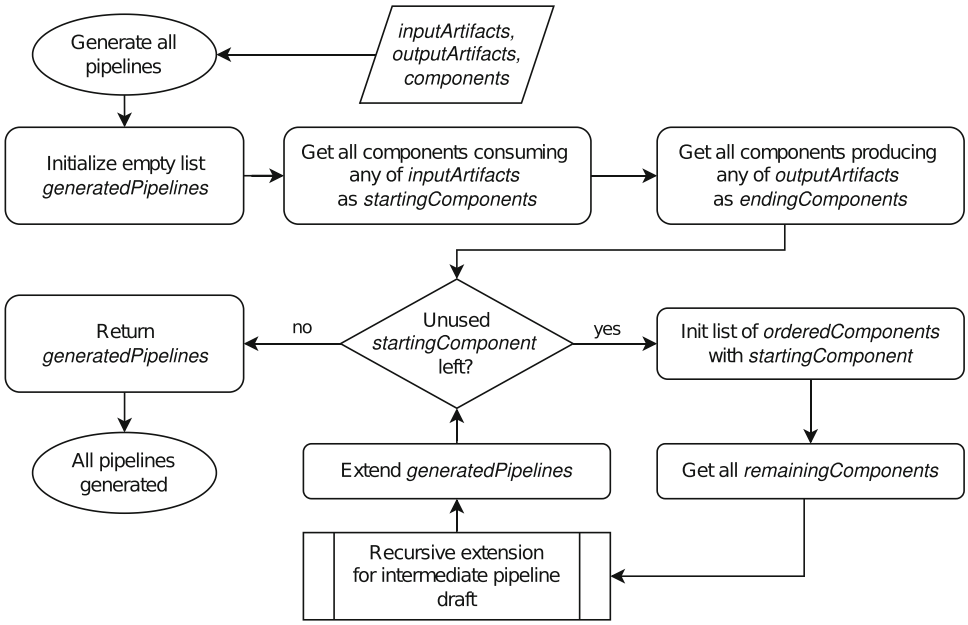

Fig. 17 Main part of the generation algorithm

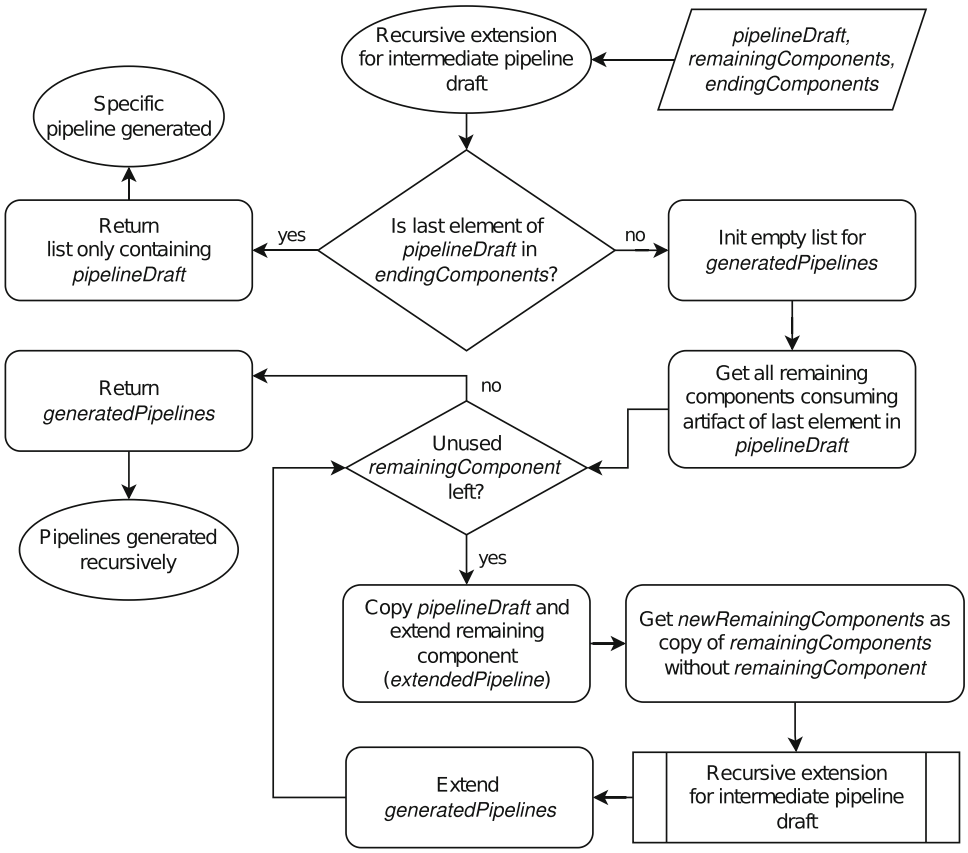

Fig. 18 Recursive part of the generation algorithm

Table 1 Example components

Component	Input	Output
PdfToTextConv	PdfDocument	TextDocument
TextPreProc	TextDocument	TextDocument
ProductNameEx1	TextDocument	ProductName
ProductNameEx2	TextDocument	ProductName

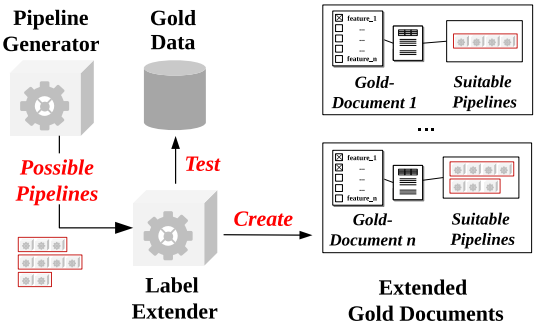

Fig. 19 Automated label extension of gold data documents [9]

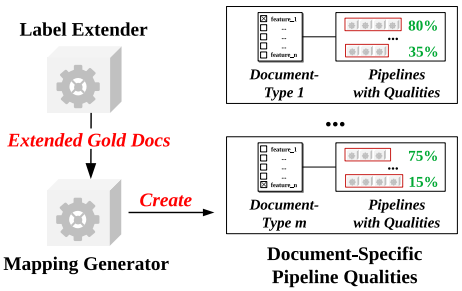

Fig. 20 Generation of document-specific pipeline qualities [9]

Automated mapping generation: The mapping generator supports the system in its decision-making by generating mappings based on the extended gold data documents. A mapping links a specific document type to a quality-ranked list of suitable pipelines. These mappings serve as a basis to decide which pipeline the system should execute if an unknown input document comes into the process (c.f. Fig. 20).

Due to the adaptability of the microservice-based architecture, developers can implement custom mapping strategies to define document types. In our case, the mapping generator creates unique combinations of boolean features to describe specific document types.

Based on the introduced concepts, the system configures itself with any update of existing or addition of new microservices. The system ensures that components come into productive use as soon as they gain business value. Combined with the formalized quality criteria, the architecture minimizes business risks by integrating new prototypes from research.

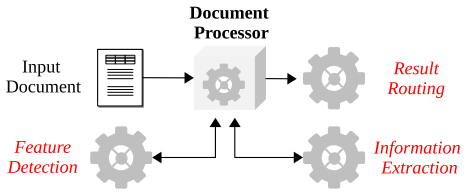

Fig. 21 Steps of the automated document processing

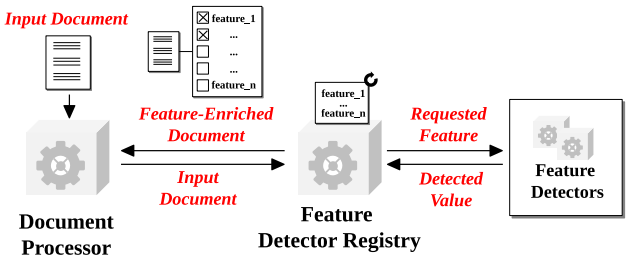

Fig. 22 Enrichment of an input document through feature detection [9]

Automated Document Processing

The overall goal is to extract information from unknown input documents automatically. To achieve this, we introduce three steps during document processing. First, the system pre-processes the input document to detect relevant features. Second, the system performs the actual IE based on the feature-enriched input document. Third, the system routes the results (c.f. Fig. 21). In the following, we explain the steps in detail.

Feature Detection

Figure 22 shows the step of feature detection. The document processor receives an unknown document as input that contains no additional information. The document processor now requests the feature detector registry to enrich the simple input document with relevant features. As a result of the feature detection auto-configuration explained in "Feature Detection Auto-Configuration", the registry knows which features it can detect meeting business requirements. For each of these features, the registry requests the specific detector for detection. After the registry has detected all features, it responds to the document processor with a feature-enriched document. The document processor can now trigger the document-specific IE as the second step explained in "Information Extraction".

Information Extraction

Figure 23 shows the step of information extraction in detail. The document processor requests the component registry to extract information from the previously generated feature-

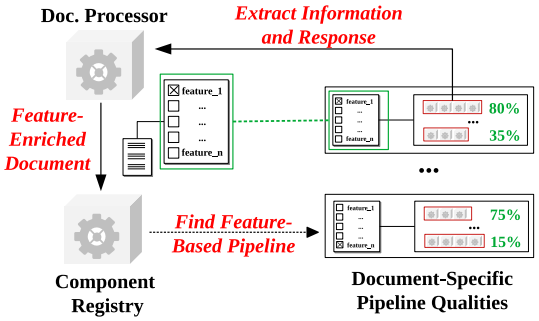

Fig. 23 Information extraction from a feature-enriched document [9]

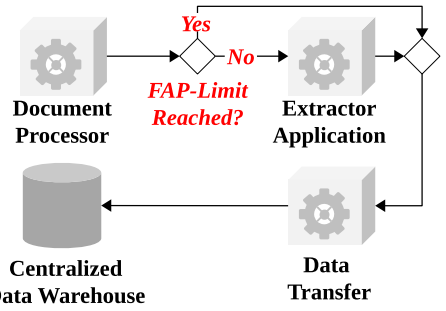

Fig. 24 Document processor routing results

enriched document. The component registry processes the document by matching the list of features with the ones of the document-specific pipeline qualities. If there is an exact match, the component registry will pick the pipelines according to the defined FAP-, N1- and N2-Limits. Otherwise, the registry will use the alternative with the minimal distance. In this case, the affected documents are not qualified for FAP. Developers can implement minimal distance functions that fit best for the focused problem. In our case, the minimal distance function finds the alternative with the highest number of equal features. If there are several ones with the same number of equal features, it will randomly pick one of them. The registry responds with the extracted information and measured pipeline quality.

Result Routing

Figure 24 shows the step of result routing. The document processor checks if the received pipeline quality from the component registry reaches the FAP-Limit. If it reaches the limit, the document processor will directly route the result to the data transfer microservice that persists the information into the centralized data warehouse. If the pipeline quality does not reach the FAP-Limit, the document processor will route the results to the extractor application. Employees use the extractor app for manual IE and result approval. Based on the formalized quality criteria, the concept of result routing enables the FAP of documents to eliminate manual efforts entirely.

Experimental Evaluation

In this section, we demonstrate the practical application of our architectural pattern in the project motivated by "Motivation". As described in "Architectural Pattern", the provided system is extensible and highly adjustable. Developers can implement specific components and detectors for their individual use case. Also, they can customize the mapping strategy and minimal distance function to find appropriate document types.

Table 2 Achieved feature detector qualities

Feature	Detector quality (%)
HAS_TABLES	86
IS_SCREENSHOT	56
IS_COLUMN_SEPARATED	45
HAS_EXACTLY_ONE_PRODUCT	86
HAS_GROSS_PRICES	91
HAS_NET_PRICES	91
HAS_OTHER_PRICE_REPRESENTATIONS	87
HAS_STAGGERED_PRODUCTS	92
HAS_TIME_VARIABLE_PRODUCTS	92
HAS_REGIONAL_CONDITIONS	45

Document Features

The system must detect relevant document features to generate optimal document-specific IE pipelines. Table 2 shows the relevant features for our specific use case and the achieved quality of available feature detectors. Since we define a quality criterion of 80%, the system can take seven out of 10 features into account when finding appropriate document types. The system will also consider the red-marked features as soon as the corresponding detectors reach the desired quality.

Subgoals and Quality Criteria

The IE process aims to extract information regarding electricity products from PDF documents. As a whole, this product information forms a complex target data model. According to "Pipeline Auto-Configuration", we split the target data model into information parts representing subgoals. For these subgoals, we define the quality criteria stated in Table 3.

The automated extraction for each subgoal supports the overall automation and therefore gains business value. However, since the individual subgoals are of different importance

Table 3 Defined subgoals and quality criteria

Subgoal	FAP-Limit (%)	N1-Limit (%)	N2-Limit (%)
DateOfValidity	95	90	75
BasicPrices	95	90	75
CommodityPrices	95	90	75
SupplierName	95	90	75
ProductName	95	80	65
CustomerGroups	95	80	65
MeteringPrices	95	80	65
ProductType	95	70	55
ProductCategory	95	70	55

from a business point of view, we also define different quality criteria.

DateOfValidity, e.g., describes at which point in time a customer can order a specific product. Since it is essential for downstream analysis, we define more sensitive quality criteria. Suppose any pipeline reaches the FAP-Limit of 95% during the automated quality determination, the system will directly route the result of the pipeline to the data transfer microservice as described in "Result Routing". If there is no pipeline reaching the FAP-Limit but the N1-Limit of 90%, the system will use this pipeline solely to route the result to the extractor app for manual approval. Below this limit, two independent pipelines at least reaching the N2-Limit of 75% must confirm each other's results so that the system can route this result for manual approval. In any other case, employees have to extract the *DateOfValidity* in the extractor app by hand. Afterwards, a second employee must approve the manual extraction.

Implemented Artifacts and Components

In addition to the domain-specific information artifacts, the concrete application of our work presented in Refs. [8, 9] led to the implementation of the following artifacts processed during IE:

- **PdfDocument** represents a PDF document and marks the input of IE process.
- **TextDocument** represents machine readable text documents containing a simple string.
- **ImgDocument** represents images containing their binary information.
- **OdtDocument** represents ODT documents containing their binary information.
- **Paragraph** represents a text paragraph as part of a Text- oder OdtDocument.
- **Table** represents a structured and machine readable table.

Furthermore, our previous work required compliance with three firmly prescribed component interfaces, namely `Converter`, `Decomposer` and `Extractor`, leading to the implementation of 24 different components. The extraction strategies reach from more complex ones based on Named Entity Recognition (NER), e.g., *NerSupplierNameEx*, to more simple strategies based on classic Regular Expressions (Regex), e.g., *DictSupplierNameEx*. During the concrete application of our system presented previously in Refs. [8, 9], we realized that the mandatory compliance with the prescribed interfaces limits the flexibility of our system and requires the implementation of unnecessary artifacts and components.

In our specific application, the text-based extractors like *NerSupplierNameEx* work on simple strings. Therefore, first we had to implement an `Element` containing a string, namely `Paragraph`. Second, we had to additionally implement a `Decomposer` that only transforms a `TextDocument` into a `Paragraph` with exactly the same content. However, since the extractors process simple strings, it is technically irrelevant how these strings are encapsulated. Therefore, we replaced the prescribed interfaces with a generic one to improve the flexibility of our system and to reduce the effort involved in development. In our concrete use case, we were able to reduce the number of artifacts and components without affecting the scope of covered functionalities (cf. Table 4).

Achieved Results

We evaluated the system introduced above in three stages:

1. We started without distinguishing document types, i.e. without feature detection (cf. "Extraction Results Without Feature Detection").
2. We integrated the pre-processing step of feature detection to achieve a more document-specific IE process (cf. "Extraction Results With Feature Detection").
3. We enhanced our previous work described in Refs. [8, 9] by reducing the number of prescribed interfaces and extending the system with caching concepts to optimize the performance of the auto-configuration (cf. "Performance Optimization").

In the following, we present the system's results in the different stages.

Extraction Results Without Feature Detection

There are several possible pipelines per information depending on the implemented components. The larger the number of components - the more unmanageable the manual composition of possible pipelines. Table 5 shows the number of

Table 4 Implemented components

ID	Name	Input artifact	Output artifact
C1	PopplerPdfToText	PdfDocument	TextDocument
C2	TesseractPdfToText	PdfDocument	TextDocument
C3	LibrePdfToOdt	PdfDocument	OdtDocument
C4	OdtToText	OdtDocument	TextDocument
C5	PopplerPdfToImg	PdfDocument	ImgDocument
C6	TextPreProcessor	TextDocument	TextDocument
C7	TableBankDec	ImgDocument	Table
C8	CamelotTableDec	PdfDocument	Table
C9	TabulaTableDec	PdfDocument	Table
C10	SimpleRegexDovEx	TextDocument	DateOfValidity
C11	ComplexRegexDovEx	TextDocument	DateOfValidity
C12	RegexBasicPriceEx	TextDocument	BasicPrice
C13	TableBasicPriceEx	Table	BasicPrice
C14	RegexCommodityPriceEx	TextDocument	CommodityPrice
C15	TableCommodityPriceEx	Table	CommodityPrice
C16	NerSupplierNameEx	TextDocument	SupplierName
C17	DictSupplierNameEx	TextDocument	SupplierName
C18	NerProductNameEx	TextDocument	ProductName
C19	NerCustomerGroupEx	TextDocument	CustomerGroup
C20	RegexMeteringPriceEx	TextDocument	MeteringPrice
C21	TableMeteringPriceEx	Table	MeteringPrice
C22	NerProductTypeEx	TextDocument	ProductType
C23	NerProductCategoryEx	TextDocument	ProductCategory

Table 5 Pipeline qualities per information without feature detection [8]

Output artifact	Possible pipelines (%)	Achieved quality	Reached limit
DateOfValidity	10	91	N1
BasicPrice	8	59	–
CommodityPrice	8	54	–
SupplierName	10	77	N2
ProductName	5	51	–
CustomerGroup	5	55	–
MeteringPrice	8	34	–
ProductType	5	55	–
ProductCategory	5	55	–

possible pipelines per information and their qualities without feature detection.

Since there are pipelines for *DateOfValidity* and *SupplierName* that at least reach the N2- but not the FAP-Limit, the system routes the extraction result to the extractor app. Due to the quality assurance step of the manual approval, we could collect data about the extraction correctness shown in Table 6. For the *SupplierName*, the number of *Matches* describes cases in which two independent pipelines confirmed each other's result.

Table 6 Results in production [8]

Information	Documents	Matches	Correct	Quote (%)
DateOfValidity	126	–	117	93
SupplierName	94	65	60	92

The results of the independent pipelines for *SupplierName* matched in 65 of 94 cases, so 29 documents had to be processed manually. In the case of the 65 matched results, the system extracted 60 correctly. Hence, in 92% of the cases for SupplierName, the registry decided correctly to return the automatically extracted result. Focusing on *DateOfValidity*, the system correctly extracted information in 117 of 126 cases using a single pipeline. This result leads to a quote of 93%.

Extraction Results With Feature Detection

In the second stage, we integrated the steps of feature detection to achieve a more document-specific process. As described in "Pipeline Auto-Configuration", we generate all possible feature combinations to find appropriate document types. With seven features, we get $2^7 = 128$ unique combinations. Based on the domain knowledge about exclusive

Table 7 Pipeline qualities per information with feature detection [9]

Output artifact	Achieved quality (%)	Reached limit	Improvement (%-pts.)
DateOfValidity	91	N1	0
BasicPrice	77	–	+ 18
CommodityPrice	74	–	+ 20
SupplierName	85	N2	+ 8
ProductName	69	–	+ 18
CustomerGroup	84	N1	+ 29
MeteringPrice	66	–	+ 32
ProductType	82	N1	+ 27
ProductCategory	86	N1	+ 31

Table 8 Example of a document type reaching FAP

Feature	Value
HAS_TABLES	False
HAS_EXACTLY_ONE_PRODUCT	True
HAS_GROSS_PRICES	True
HAS_NET_PRICES	False
HAS_OTHER_PRICE_REPRESENTATIONS	False
HAS_STAGGERED_PRODUCTS	False
HAS_TIME_VARIABLE_PRODUCTS	False

mutual characteristics of these features, we know that the number of occurring combinations is much smaller in reality. Hence, the system determined 35 unique feature combinations in 1300 gold data documents, each with at least 30 related documents.

Table 7 shows the results of the automated IE with feature detection.

Excluded *DateOfValidity*, the quality for each subgoal increased, and the system reached the N1-Limit for *CustomerGroup*, *ProductType*, and *Product Category*.

Furthermore, the system detected six document types with specific feature combinations for which the pipelines perform extraordinarily well, measured by the defined FAP criteria. Table 8 exemplary shows one of these types.

For each of the six document types, there are pipelines that reach the required FAP-Limits. Hence, no manual extraction or approval effort is necessary. 174 documents of the gold data set belong to one of these document types, so the system identifies a potential for FAP of about 13% in the set of gold data.

The system processed 524 documents with feature detection, and related 76 of them to one of the six mentioned document types. The system processed all of these 76 without any manual effort, which leads to a quote of 14% FAP in practice.

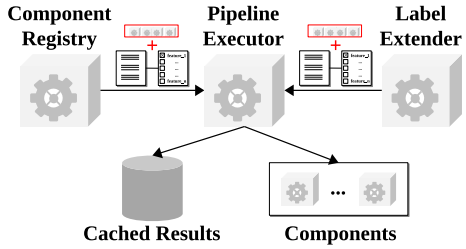

Fig. 25 Pipeline executor reusing cached results

Performance Optimization

In the third stage, we integrated concepts to optimize the performance of the system's auto-configuration. Since each feature or component registry change triggers the auto-configuration, the system runs this process very frequently. The number of possible pipelines and documents increases with the number of components and feature detectors. Therefore, the effort for the auto-configuration may increase exponentially. We extended the system by integrating caches in bottlenecks to avoid performance issues.

The system has to execute pipelines during the processing of unknown input documents and the auto-configuration. Therefore, we introduce the pipeline executor that executes a specific pipeline for a particular input document. As shown in Fig. 25, the component registry and the label extender can request the pipeline executor to receive the extraction result for the combination of a pipeline and a feature-enriched document. While the component registry sends a completely unknown input document, the label extender sends a gold data document.

The pipeline executor stores each component result represented by the specific input document, the component's version and the created output artifacts in the cache. Before requesting the next component, the pipeline executor tries to reuse cached results. Appropriate results must match already generated output artifacts of preceding components as input artifacts for the next component. If there are no matching results in the cache, the pipeline executor will request the corresponding component to generate new results. Afterwards, it stores the new results in the cache.

Since the component registry always requests the execution for an unknown input document, there are no cached results. However, the label extender always requests the execution for the same set of documents. Therefore, the implementation of this caching mechanism strongly optimizes the auto-configuration performance. Table 9 shows an example comparison of the duration without and with caching to determine the qualities for the *DateOfValidity* subgoal. The pipelines contain components introduced with Table 4. The system always determines the pipeline qualities in the same order so that the results are comparable.

Table 9 Duration for the determination of document-specific pipeline qualities

No.	Pipeline	W/o caching (min)	With caching (min)	Diff. (min)
1	C1, C10	5:11	5:06	− 0:05
2	C1, C6,C10	6:02	1:12	− 4:50
3	C2, C10	10:31	10:28	− 0:03
4	C2, C6, C10	11:01	1:28	− 9:33
5	C3, C4, C10	6:44	6:45	+ 0:01
6	C1, C11	5:02	0:49	− 4:13
7	C1, C6, C11	6:36	1:16	− 5:20
8	C2, C11	10:55	0:50	− 10:05
9	C2, C6, C11	11:47	1:13	− 10:34
10	C3, C4, C11	6:52	0:46	− 6:06

It is noticeable that results *1*, *3*, and *5* are almost identical with- and without cached results because the system initially has to create and store results before reusing them. The difference is enormous in the following cases where the system can reuse cached results. Since the conversion steps are time-consuming, integrating the provided caching strategy leads to huge performance optimization when developers push new components into the system.

Implementation

In the following subsection, we present an exemplary implementation of our architectural pattern introduced in "Architectural Pattern".

Due to the nature of microservices, it is not necessary to use one single programming language for all microservices. The communication between these microservices happens programming language-independent in the form of Representational State Transfer (REST) calls over the Hypertext Transfer Protocol (HTTP) protocol.

In our implementation, we used a Python stack for the components, pipeline generator, feature detectors, feature detector quality determiner, label extender, mapping generator, and pipeline executor. Each Python microservice runs a FastAPI[3] web service that provides REST endpoints for the other microservices to call. For the remaining microservices, namely the component registry, feature detector registry, and document processor, we used a Java stack with Maven[4] and Spring Boot.[5]

We made heavy use of OpenAPI[6] to define data transfer objects (DTO) and REST endpoints in a programming language-independent way. The OpenAPI schema enabled us to use the OpenAPI Generator[7] to generate server and client SDKs for the Python and Java microservices.

To ensure that the web services work the same way regardless of the surrounding infrastructure, we used Docker[8] to containerize each microservice. Containerization allows us to deploy these microservices in a container orchestration system such as Kubernetes.[9]

Feature Detectors

Feature detectors are microservices the system uses to detect a specific feature inside a document, e.g., the presence or absence of price tables. To easily add and modify different feature detection strategies, we implemented each feature detector as a separate microservice. Also, this allows developers to choose the most appropriate language and framework for a specific detection problem.

Each feature detector microservice offers two endpoints. The first endpoint performs the detection of the specific feature by receiving a document and returning a boolean flag that indicates the value of the feature. The second endpoint provides information about the detector's name, version and the feature it can detect.

Code Listing 1 shows the implementation of the information endpoint of a feature detector. The returned information tells the feature detector registry that this detector can determine whether a document is an original PDF document or a screenshot.

```
@router.get("/info", response_model=
    ↪ DetectorInfo)
def get_info(self):
  return DetectorInfo(
    name="is-screenshot-detector",
    version="1.0.0",
    detects=DocumentFeatureKeyEnum.
        ↪ IS_SCREENSHOT
  )
```

Code Listing 1 Information Endpoint of a Feature Detector

Feature Detector Quality Determiner

As stated in "Feature Detection Auto-Configuration", we need to evaluate the quality of each feature detector. Therefore, we implemented a web service that tests a detector based on the set of gold documents by detecting the particular feature in each gold document.

Code Listing 2 shows the quality determination of a feature detector. For each gold document, the quality determiner

[3] https://fastapi.tiangolo.com/.

[4] https://maven.apache.org/.

[5] https://spring.io/.

[6] https://swagger.io/specification/.

[7] https://github.com/OpenAPITools/openapi-generator.

[8] https://www.docker.com/.

[9] https://kubernetes.io.

requests the feature detector and compares its result with the expected one of the gold document. The determiner calculates the resulting quality dividing the number of correct results by the number of gold documents.

```
def determine_quality(feature_detector):
  correct_results =0
  for gold_document in self._gold_documents:
    gold_document_feature =self.
        ↪ _get_gold_document_feature(
        ↪ gold_document, feature_detector.
        ↪ detects)
    feature_response =requests.post(
        ↪ feature_detector.endpoint, data=
        ↪ gold_document.json())
    if gold_document_feature ==DocumentFeature
        ↪ .parse_obj(feature_response.json())
        ↪ :
      correct_results +=1
  return correct_results /len(self.
      ↪ _gold_documents)
```

Code Listing 2 Determine Feature Detector Quality

Feature Detector Registry

As also mentioned in "Feature Detection Auto-Configuration", we implemented a web service that manages the feature detectors. Once we defined the required quality criteria for the features shown in Table 2, we can start registering detectors in the feature detector registry.

Code Listing 3 shows the registration of a feature detector. When a feature detector attempts to register, the feature detector registry queries the information endpoint of the detector to determine the detectable feature. The registry then queries the feature detector quality determiner for the detector's quality. If the received quality reaches the defined threshold, the registry will inform the mapping generator by forwarding all relevant features.

```
void addDetector(InetSocketAddress address){
  FeatureDetectorInfo detectorInfo =
      ↪ requestDetectorInfo(address);
  FeatureDetectorQuality quality =
      ↪ requestDetectorQuality(detectorInfo,
      ↪ address);
  if (hasRequiredQuality(quality)) {
    Detector detector = new Detector(address,
        ↪ quality);
    addresses.put(detector.getName(), detector
        ↪ );

    List<String> allFeatures = addresses.
        ↪ values().stream()
      .map(Detector::getDetects)
      .toList();
    notifyMappingGenerator(allFeatures);
  }
}
```

Code Listing 3 Registration of New Detectors [9]

Components

Analogous to feature detectors, we implemented each component as a separate microservice that provides an endpoint to execute its specific task. In addition, each implemented component microservice must provide an information endpoint. This endpoint returns the name and version of the microservice, the endpoint to request processing, as well as the types of the consumed and produced artifacts.

Code Listing 4 shows the implementation of the information endpoint of a component. The returned information signals the component registry that this component can convert a given PDF document into a text document. The component shown in this example internally uses Tesseract[10] to extract text from the PDF document using optical character recognition (OCR).

```
@controller.get("/info", response_model=
    ↪ ComponentEndpointInfo)
def get_info():
  return ComponentEndpointInfo(
    name="TesseractPdfToText",
    consumes="PdfDocument",
    produces="TextDocument",
    version="1.0.0",
    endpoint="/process"
  )
```

Code Listing 4 Information Endpoint of a Component

Component Registry

As stated in "Pipeline Auto-Configuration", we implemented a microservice that manages components. The registration process is mainly analogous to that of the feature detector registry.

Code Listing 5 shows the registration of a new component. When a component tries to register, the component registry queries the component's information endpoint to determine the type of task it implements. The registry will test the component's endpoint with example data and completes the registration if this check is successful. After the registration, the registry forwards all relevant component information to the pipeline generator. This procedure triggers the re-generation of all valid pipelines.

```
@SneakyThrows
public void addComponent(String address, int
    ↪ port) {
  InetAddress inet = InetAddress.getByName(
      ↪ address);
  InetSocketAddress sock = new
      ↪ InetSocketAddress(inet, port);

  ComponentEndpointInfo info =
      ↪ requestComponentInfo(sock);
```

[10] https://github.com/tesseract-ocr/tesseract.

```
  if (verifyEndpoint(info, sock)) {
    Component comp = new Component(sock,info);
    allComps.put(comp.getName(), comp);

    pipelineGeneratorService.notify(allComps);
  }
}
```

Code Listing 5 Registration of New Components [8]

Pipeline Generator

The pipeline generator is a FastAPI web service that realizes the goal-specific pipeline generation stated in "Pipeline Auto-Configuration". It implements the algorithm shown in Figs. 17 and 18.

Code Listing 6 illustrates the main part of the pipeline generation algorithm. We first generate the list of starting components by selecting all components that consume any defined input artifact. Analogously, we generate the list of ending components by selecting all components producing any defined output artifact. Next, we iterate over the list of starting components and call the recursive part of the pipeline generation algorithm. The recursive function returns a list of all valid pipelines with matching in- and outputs. The result of the shown implementation is a complete list of valid pipelines based on all possible component permutations for the given in- and output artifacts.

```
@staticmethod
def build_pipelines(
  input_artifacts: List[str],
  output_artifacts: List[str],
  components: List[Component]
) ->List[Pipeline]:
  generated_pipelines =[]

  starting_components =[c for c in components
      ↪ if c.consumes in input_artifacts]
  ending_components =[c for c in components if
      ↪  c.produces in output_artifacts]

  for starting_component in
      ↪ starting_components:
    ordered_components =[starting_component]
    remaining_components =[c for c in
        ↪ components if c !=
        ↪ starting_component]
    generated_pipelines.extend(
      build_pipeline_recursive(
        ordered_components,
            ↪ remaining_components,
            ↪ ending_components
      )
    )

  return generated_pipelines
```

Code Listing 6 Pipeline Generation

Label Extender

The label extender microservice implements the label extension introduced in "Pipeline Auto-Configuration". After the system has performed the label extension, the label extender triggers the mapping generator service.

Code Listing 7 shows the steps to perform the label extension for all gold documents. The extender runs each pipeline against each document for all subgoals. A pipeline will be suitable for the document if the output matches the expected information of the gold data document.

```
def extend_labels(self):
  extended_docs =[]
  for gold_doc in self.get_gold_documents():
    suitable_pipelines =[]
    for info in DomainModel.information:
      for pipeline in self.
          ↪ get_pipelines_for_information(
          ↪ info):
        result =PipelineExecutor.
            ↪ execute_pipeline(
          pipeline=pipeline,
          gold_document=gold_doc
        )
        if result ==gold_doc.get(info):
          suitable_pipelines.append(pipeline)
    extended_docs.append(
      LabeledGoldDocument(
        gold_document=gold_doc,
        suitable_pipelines=suitable_pipelines
      )
    )
  return extended_docs
```

Code Listing 7 Label Extension of Gold Documents [9]

Mapping Generator

As stated in "Pipeline Auto-Configuration", the mapping generator determines the best document-specific pipelines. To achieve this, we use a rule-based strategy in our current implementation.

In Code Listing 8, we illustrate the first step to generate mappings that link document types to suitable pipelines. For each feature combination, we select all gold documents that match this combination representing a specific document type. We then calculate the quality of each pipeline per document type by counting its occurrences in the lists of all corresponding documents divided by the number of corresponding documents. For each document type, this operation creates a list of suitable pipelines and their qualities. In the second step, the mapping generator ranks the suitable pipelines for each document type based on their quality. The results of this step represent the required document-specific pipelines. After completing both steps, the component registry receives all mappings from the generator. This information allows the registry to handle incoming documents appropriately.

```
def generate_mappings(self, labeled_docs:
    ↪ List[LabeledGoldDocument]):
  document_type_pipeline_mappings =[]
  for feature_combination in self.
      ↪ generate_feature_combinations():
    suitable_pipelines =[]
    matching_gold_docs =self.
        ↪ match_labeled_documents(
        ↪ labeled_docs, feature_combination)
    for labeled_doc in matching_gold_docs:
      suitable_pipelines.extend(labeled_doc.
          ↪ suitable_pipelines)
    for suitable_pipeline in set(
        ↪ suitable_pipelines):
      document_type_pipeline_mappings.append(
        DocumentPipelineMapping(
          features=feature_combination,
          pipeline=suitable_pipeline,
          quality=suitable_pipelines.count(
              ↪ suitable_pipeline) /len(
              ↪ matching_gold_docs)
        )
      )
  return document_type_pipeline_mappings
```

Code Listing 8 Mapping Generation [9]

Pipeline Executor

As mentioned in "Performance Optimization", the pipeline executor is a microservice that executes pipelines during production and auto-configuration. It also implements the result caching.

Code Listing 9 shows the execution of a single pipeline. The method iterates over each component of a pipeline. It determines the input artifacts for the current component. For the first component, the input is the incoming document. The following components receive the matching artifacts of the previous components. Afterwards, the system determines whether the cache already contains the result of the current component and the specific input artifact. If the cache already contains the result, the system will not need to perform further actions on this component. Otherwise, the pipeline executor will query the current component and cache the result. After the method repeats this process for all components, the execution finishes.

```
def execute_pipeline(self) ->
    ↪ PipelineExecution:
  ordered_component_results =[]
  for component in self._pipeline.
      ↪ ordered_components:
    matching_artifacts =self.
        ↪ _get_matching_artifacts(component,
        ↪ ordered_component_results)

    for artifact in matching_artifacts:
      component_result =self._cached_results.
          ↪ get_result(component, artifact)
      if not component_result:
        component_response =requests.post(
            ↪ component.endpoint, data=
            ↪ artifact.json())

        component_output_artifacts =[]
        for output_artifact in
            ↪ component_response.json():
          component_output_artifacts.append(
              ↪ ARTIFACTParser.parse_obj(
              ↪ output_artifact))

        component_result =ComponentResult(
          component=component,
          input_artifact=artifact,
          output_artifacts=
              ↪ component_output_artifacts
        )
        self._cached_results.save_result(
            ↪ component_result)
      ordered_component_results.append(
          ↪ component_result)
  return PipelineExecution(
      ↪ ordered_component_results)
```

Code Listing 9 Pipeline Execution

Document Processor

The document processor is a Java Spring Boot web service and provides a Camunda[11] process engine. As mentioned in "Automated Document Processing", the document processor manages the overall process and runs the FAP of documents, depending on the IE quality.

After the user uploads a document for IE, the document processor calls the feature detector registry to determine relevant features. This step results in a feature-enriched document the processor forwards to the component registry for IE. The document processor receives the extracted information and its estimated quality. The extracted information can be processed automatically if the resulting quality is high enough (see Table 3). Otherwise, employees must check the information manually.

Code Listing 10 shows a Camunda service task that enriches a document from the *DOCUMENT* variable with its detected features and stores it in the variable *ENRICHED_DOCUMENT*. In subsequent service tasks, this variable is used to query the component registry for the actual IE.

```
@Override
void execute(DelegateExecution exec) {
  FileValue docFile = exec.getVariableTyped("
      ↪ DOCUMENT");
  Document document = toDocument(docFile);

  FeatureEnrichedDocument serverResult =
      ↪ sendDocumentToServer(document);
  exec.setVariable("ENRICHED_DOCUMENT",
      ↪ serverResult);
```

[11] https://camunda.com/.

```
}
```

Code Listing 10 Detection Service Task [9]

In Code Listing 11, the following Camunda service task receives the *ENRICHED_DOCUMENT* variable and requests the extraction of all information within the document.

```
@Override
void execute(DelegateExecution exec) {
  ObjectValue docFile = exec.getVariableTyped(
      ↪ "ENRICHED_DOCUMENT");
  FeatureEnrichedDocument document = docFile.
      ↪ getValue(FeatureEnrichedDocument.
      ↪ class);

  List<InformationWithQuality> result =
      ↪ sendDocumentToServer(document);
  if (result.isEmpty()) {
    throw new RuntimeException("No results
        ↪ received!");
  }

  double overallConfidence = result.stream()
    .mapToDouble(InformationWithQuality::
        ↪ confidence)
    .min().orElse(0);
  List<Information> information = result.
      ↪ stream()
    .map(InformationWithQuality::information)
    .toList();

  exec.setVariable("FULLY_AUTOMATED_PROCESSING
      ↪ ", overallConfidence >= 0.95);
  exec.setVariable("RESULT", information);
}
```

Code Listing 11 Extraction Service Task [9]

Conclusion

Product information is the basis for many businesses. Often, there are no document standards, and companies have to extract this information by hand. The automation of affected processes requires many different solution strategies to cover the possible document formats. To gain business value, companies need a system that independently finds optimal pipelines to process documents automatically. To tackle this problem, we introduced a distributed microservice architecture pattern that enables the automated generation of IE pipelines.

The provided architectural pattern is domain-independent and adaptable to use cases focusing on IE from non-standardized documents. On the one hand, the extensible object model allows domain experts to define custom document, element, and domain-specific information types. On the other hand, the introduced registries enable developers to implement microservices that are most suitable for their focused problem.

The automated system configuration ensures that the system independently figures out how to optimally process a specific type of document. Gold data documents build the basis for the system to test valid pipelines and determine their qualities. To achieve this, we introduced several microservices: The document manager always keeps the gold data documents up to date to ensure that the set always represents real-world conditions. The pipeline generator generates valid pipelines based on the in- and output of components. The label extender finds suitable pipelines for each gold data document as preparation for the determination of document types. The mapping generator finds appropriate document types with suitable pipelines ordered by their document-specific quality.

The document processor manages the automated document processing. First, it enriches a simple input document with a list of features by requesting the feature detector registry. Second, it requests the component registry to extract information. Dependent on formalized business requirements as quality criteria, the document processor afterwards routes the generated results. If the quality is sufficient, it will route the results to the data transfer microservice, completely eliminating manual effort. Otherwise, it will route the results to the extractor application, where employees can extract and approve information by hand.

We evaluated the introduced approach by applying it to a concrete problem in the German energy industry. The evaluation shows that the implemented system successfully helps our industry partner to automate its IE process and integrate the newest research findings as early as possible.

Furthermore, the evaluation shows that the integration of the pipeline executor and result caching improves the performance of the auto-configuration enormously. Also, replacing the three rigid interfaces `Converter`, `Decomposer` and `Extractor` with one generic interface reduced the number of required components and thus the development effort. In future research, we will try to further optimize the performance of the system by automating the scaling of components with higher loads, such as conversion components.

Funding Open Access funding enabled and organized by Projekt DEAL.

Data Availability The data is not available publicly.

Declarations

Conflicts of interest All authors declare no conflicts of interest.

Ethical standards This article does not contain any studies with human participants or animals performed by any of the authors.

References

1. Cardie C. Empirical Methods in Information Extraction. AI Mag. 1997;18(4):5. https://doi.org/10.1609/aimag.v18i4.1322.
2. Hashmi KA, et al. Current status and performance analysis of table recognition in document images with deep neural networks. http://arxiv.org/abs/2104.14272. arXiv:2104.14272 (2021).
3. Hanson C, Sussman GJ. Software design for flexibility—how to avoid programming yourself into a corner. Cambridge: MIT Press; 2021.
4. Jamshidi P, Pahl C, Mendonça NC, Lewis J, Tilkov S. Microservices: the journey so far and challenges ahead. IEEE Softw. 2018;35(3):24–35.
5. Newman S. Building microservices: designing fine-grained systems. 1st ed. Sebastopol: O'Reilly Media; 2015.
6. Beck K. Extreme programming—die revolutionäre Methode für Softwareentwicklung in kleinen Teams; [das Manifest]. München: Pearson Deutschland GmbH; 2003.
7. Rubin KS. Essential scrum—a practical guide to the most popular agile process. 01 ed. Boston: Addison-Wesley Professional; 2012.
8. Sildatke M, Karwanni H, Kraft B, Zündorf A. ARTIFACT: Architecture for Automated Generation of Distributed Information Extraction Pipelines. In Proceedings of the 24th International Conference on Enterprise Information Systems - Volume 2: ICEIS; ISBN 978-989-758-569-2; ISSN 2184-4992, SciTePress, pp. 17–28. https://doi.org/10.5220/0010987000003179.
9. Sildatke M, Karwanni H, Kraft B, Zündorf A. FUSION: Feature-based Processing of Heterogeneous Documents for Automated Information Extraction. In Proceedings of the 17th International Conference on Software Technologies - ICSOFT; ISBN 978-989-758-588-3; ISSN 2184-2833, SciTePress, pp. 250–260. https://doi.org/10.5220/0011351100003266.
10. ene't GmbH. Markdaten Endkundentarife Strom and Gas 2022. Retrieved from https://download.enet.eu/uebersicht/datenbanken. Accessed 30 Sept 2023.
11. Dragoni N, et al. Microservices: yesterday, today, and tomorrow 2017. eprint http://arxiv.org/abs/1606.04036 [cs].
12. Lewis J, Fowler M. Microservices. https://martinfowler.com/articles/microservices.html (2014). Accessed 30 Sept 2023.
13. Chowdhury SR, Salahuddin MA, Limam N, Boutaba R. Re-architecting NFV ecosystem with microservices: state of the art and research challenges 2019;33(3):168–176. https://ieeexplore.ieee.org/document/8688711/. https://doi.org/10.1109/MNET.2019.1800082. Accessed 30 Sept 2023.
14. Hohpe G. Enterprise integration patterns—designing, building and deploying messaging solutions. Boston: Addison-Wesley Professional; 2003.
15. Camposo G. Cloud native integration with apache camel—building agile and scalable integrations for Kubernetes platforms. New York: Apress; 2021.
16. Fuld I, Partner J, Fisher M, Bogoevici M. Spring integration in action. New York: Simon and Schuster; 2012.
17. Richardson, C. (2019) Microservices patterns: With examples in Java. Shelter Island: Manning Publications.
18. Peltz C. Web services orchestration and choreography 2003;36(10):46–52. http://ieeexplore.ieee.org/document/1236471/. https://doi.org/10.1109/MC.2003.1236471. Accessed 30 Sept 2023.
19. Fowler M. An appropriate use of metrics (2013). https://martinfowler.com/articles/useOfMetrics.html#WhatsWrongWithHowWeUseMetrics. Accessed 30 Sept 2023.
20. Schreiber M, Kraft B, Zündorf A. Metrics driven research collaboration: focusing on common project goals continuously 41–47 (2017). https://doi.org/10.1109/SER-IP.2017..6.
21. Oliver Schmidts, Bodo Kraft, Marc Schreiber, and Albert Zündorf. 2018. Continuously evaluated research projects in collaborative decoupled environments. In Proceedings of the 5th International Workshop on Software Engineering Research and Industrial Practice (SER&IP '18). Association for Computing Machinery, New York, NY, USA, 2–9. https://doi.org/10.1145/3195546.3195549.
22. Kimball R, Caserta J. The data warehouse ETL toolkit: practical techniques for extracting, cleaning, conforming, and delivering data. Hoboken: Wiley; 2011.
23. Seidler K, Schil A. Service-oriented information extraction 25–31 (2011). http://portal.acm.org/citation.cfm?doid=1966874.1966879. https://doi.org/10.1145/1966874.1966879. Accessed 30 Sept 2023.
24. Saggion H, Funk A, Maynard D, Bontcheva K. Ontology-based information extraction for business intelligence 843–856 (2007).
25. Wimalasuriya DC, Dou D. Ontology-based information extraction: an introduction and a survey of current approaches. J Inf Sci. 2010;36(3):306–23.
26. Ivančić, Lucija & Suša Vugec, Dalia & Vuksic, Vesna. (2019). Robotic Process Automation: Systematic Literature Review. https://doi.org/10.1007/978-3-030-30429-4_19.
27. Aguirre, Santiago & Rodriguez, Alejandro. (2017). Automation of a Business Process Using Robotic Process Automation (RPA): A Case Study. 65–71. https://doi.org/10.1007/978-3-319-66963-2_7.
28. Jiang S, Pang G, Wu M, Kuang L. An improved k-nearest-neighbor algorithm for text categorization. J Expert Syst Appl. 2012;39(1):1503–9.
29. Lilleberg, Joseph, Yun Zhu and Yanqing Zhang. "Support vector machines and Word2vec for text classification with semantic features." 2015 IEEE 14th International Conference on Cognitive Informatics & Cognitive Computing (ICCI*CC) (2015): 136–140.