ARTIFACT: Architecture for Automated Generation of Distributed Information Extraction Pipelines

Michael Sildatke[1], Hendrik Karwanni[1], Bodo Kraft[1] and Albert Zündorf[2]
[1]*FH Aachen, University of Applied Sciences, Germany*
[2]*University of Kassel, Germany*

Keywords: Modelling of Distributed Systems, Model Driven Architectures and Engineering, Software Metrics and Measurement, Agile Methodologies and Applications, Domain Specific and Multi-aspect IS Engineering.

Abstract: Companies often have to extract information from PDF documents by hand since these documents only are human-readable. To gain business value, companies attempt to automate these processes by using the newest technologies from research. In the field of table analysis, e.g., several hundred approaches were introduced in 2019. The formats of those PDF documents vary enormously and may change over time. Due to that, different and high adjustable extraction strategies are necessary to process the documents automatically, while specific steps are recurring. Thus, we provide an architectural pattern that ensures the modularization of strategies through microservices composed into pipelines. Crucial factors for success are identifying the most suitable pipeline and the reliability of their result. Therefore, the automated quality determination of pipelines creates two fundamental benefits. First, the provided system automatically identifies the best strategy for each input document at runtime. Second, the provided system automatically integrates new microservices into pipelines as soon as they increase overall quality. Hence, the pattern enables fast prototyping of the newest approaches from research while ensuring that they achieve the required quality to gain business value.

1 INTRODUCTION

Many businesses build their services based on product information. Amazon, e.g., collects information about over 350 million products to sell them on their marketplace platform[1]. Other examples are comparison portals that use product information to offer their customers a ranking of the most suitable alternatives.

Often, providers publish product information in PDF documents in which tables contain important price information. Since these PDF documents are only human-readable and vary enormously in content and format, employees must extract the relevant information by hand. To gain business value, companies attempt to automate the process of Information Extraction (IE). Because classic ETL technologies reach their limits, businesses use the newest technologies and approaches from research. In the field of table analysis, e.g., several hundred approaches were introduced in 2019 (Hashmi et al., 2021).

The underlying IE problems are often very complex, so it takes a long time and much effort to develop suitable strategies. Moreover, rapidly changing environmental requirements result in adjustments to the software. Manual effort is needed to bring frequently emerging solutions into productive use.

Due to the great variety of document formats and the strengths of specific technologies, it is necessary to develop different extraction strategies. Since completely unknown formats can occur, the developed strategies also have to be highly adjustable. This situation leads to an extensive set of possible strategies. Thus, identifying the most suitable strategies is challenging. Furthermore, the reliability of extracted information is a very critical factor for business success.

These challenges prevent companies from automating their IE processes.

This paper introduces an architectural pattern that tackles the challenges mentioned above based on distributed microservices. The pattern ensures fast prototyping of the newest approaches and the automated composition of the most suitable strategies. Based on formalized quality criteria, it guarantees that automatically extracted information meet business requirements.

[1]https://www.bigcommerce.com/blog/amazon-statistics/#amazon-everything-to-everybody

Sildatke, M., Karwanni, H., Kraft, B. and Zündorf, A.
ARTIFACT: Architecture for Automated Generation of Distributed Information Extraction Pipelines.
DOI: 10.5220/0010987000003179
In *Proceedings of the 24th International Conference on Enterprise Information Systems (ICEIS 2022) - Volume 2*, pages 17-28
ISBN: 978-989-758-569-2; ISSN: 2184-4992
Copyright © 2022 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

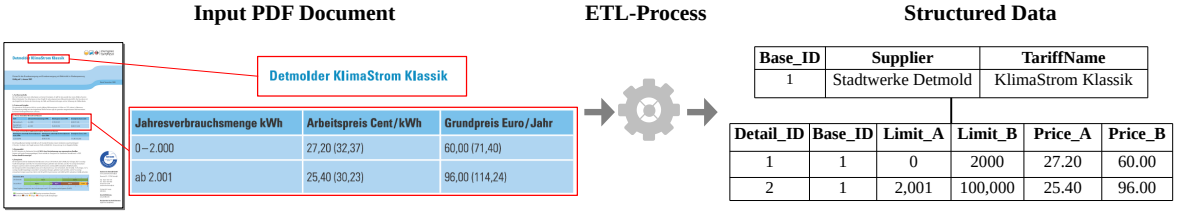


Figure 1: Simplified information extraction workflow example from an input PDF document into structured data.

The paper is structured as follows: Section 2 describes the real-world project which motivates our approach. Section 3 describes related works. Section 4 introduces **Ar**chi**t**ecture for Automated Generati**o**n of Distributed In**f**orma**t**ion Extra**ct**ion Pipelines (ARTIFACT). Section 5 describes the experimental evaluation of ARTIFACT in the real-world project, while Section 6 summarizes the paper. Section 7 ends the paper with an overview of future developments.

2 MOTIVATION

The following section provides an example from the energy industry that motivates our ARTIFACT pattern and the importance of (semi-)automatic information extraction.

In Germany, about 3,150 energy suppliers offer more than 15,000 different electricity or gas products[2]. Other service providers use this product information as the basis for specific services, e.g., comparison of prices. Figure 1 illustrates a typical part of an information extraction workflow in a simplified way.

Usually, suppliers adjust their products 1-2 times a year, so service providers have to process about 25,000 documents annually. Suppliers use custom formats because there is no standard. These custom formats may change over time and upcoming suppliers cause completely new ones.

Non-machine readable PDF documents are the source of relevant information, so employees have to extract the information by hand. A common manual IE process typically includes the following steps:

- **Matching the Supplier with the Base Data.** The extractor has to match the providing supplier with the base data. If there is no base data record yet, the extractor has to create one.
- **Identifying the Number of Products.** Documents can describe several products. Therefore, the extractor has to determine how many different products they have to consider.
- **Identifying Relevant Document Parts.** Not all parts of the document contain relevant data. The extractor identifies only the parts which contain relevant data.
- **Understanding Table Semantics.** If the document contains several products, maybe one table will contain all price information. The extractor has to separate the content of the table according to every single product.
- **Understanding Text Semantics.** Some information is part of natural text. The extractor has to examine the relevant text parts and their contexts to get the relevant information.
- **Resolving Different Information Representations.** Usually, there are various ways of price representation in a document, i.e., gross or net. The extractor has to consider that they ought to extract the information only once.
- **Inferring Non-explicit Information.** Some information is non-explicit and results from the absence of specific content. The extractor has to take this from the context. Figure 1 shows an example: If there is no explicit limit B, its value will be 100,000.

The correctness of the extracted data is fundamental because it forms the basis for downstream services. Incorrect data causes poor quality and therefore lowers the business value.

Since information extraction is complex and sensitive at the same time, automation is challenging to achieve. Manual extraction is very time-consuming and expensive. Reducing its effort becomes economically relevant.

Automation of these processes requires an architecture that ensures the fast prototyping of the newest approaches, including a dynamic variation of strategies. Combined with the automated composition of the most suitable strategies, such an architecture can help companies to gain business value.

[2] *ene't Markdaten Endkundentarife Strom & Gas* https://download.enet.eu/uebersicht/datenbanken

3 RELATED WORK

The challenges mentioned above are especially related to the fields of flexible software, fast prototyping, software integration and composition, as well as software quality metrics.

Rapidly changing environments require flexible software architectures. Systems that enable evolution by adding code rather than changing existing code ensure the adaption for new situations (Hanson and Sussman, 2021). Furthermore, established concepts like separation of concerns, bounded context or Domain-driven Design (DDD) emphasize the need for software modularization (Tarr et al., 1999; Evans and Evans, 2004). An Microservice Architecture (MSA) is scalable, easy to maintain and extendable since each microservice is an independent unit (Jamshidi et al., 2018). Due to that, microservices and MSA can be used to implement these concepts and build flexible software (Newman, 2015).

Ongoing research on new technologies leads to a vast number of upcoming IE approaches. Agile development approaches like Extreme Programming or Scrum allow fast prototyping and are suitable to create proofs of concept easily (Beck, 2003; Rubin, 2012).

Several frameworks support fast prototyping with microservices, e.g., Spring Boot for Java or FastAPI for Python (Walls, 2015; Voron, 2021).

Despite all benefits of MSA, there are also some challenges. Building dependable systems is challenging because microservices are autonomous (Dragoni et al., 2017). Different underlying technologies may have various means of specifications needed for the composition of services (Dragoni et al., 2017). Due to that, the verification of microservice functionalities is challenging (Chowdhury et al., 2019). Possible failing compositions of microservices lead to more complexity in the connections between those services and unexpected runtime errors (Lewis and Fowler, 2014).

These challenges are addressed in the field of software integration and service composition. Enterprise Integration Pattern (EIP) provide theoretical approaches for software integration (Hohpe and Woolf, 2003). Frameworks like Apache Camel or Spring Integration implement EIP and can be used to bring theoretical integration approaches into practice (Camposo, 2021; Fuld et al., 2012). Microservice Patterns describe approaches to transfer the ideas of EIP into MSAs (Chris Richardson, 2018).

Primarily, there are two approaches handling service composition: orchestration and choreography (Peltz, 2003). The basis for orchestration is a centralized unit that controls the communication between microservices. Choreography uses events to realize a decentralized communication between microservices. We use orchestration as the basis. Thus, the centralized unit can handle the composition of microservices according to identify the best alternative.

Service discovery allows automatic detection of services based on provided functionalities and technical criteria, e.g., response times (Marin-Perianu et al., 2005). To solve the challenge of service composition, the *Service Registry Pattern* combines the concepts of self-registering services and service discovery (Chris Richardson, 2018). The patterns mentioned do not focus on optimizing service composition based on functional criteria, e.g., extraction quality.

Appropriate software metrics should be explicitly linked to goals (Fowler, 2013). Metrics Driven Research Collaboration (MEDIATION) focuses on the development of research prototypes and uses business-specific metrics to measure software quality (Schreiber et al., 2017). (Schmidts et al., 2018) provide an approach that combines MEDIATION with containerization of research prototypes. However, these approaches do not focus on highly distributed architectures and still require manual management to decide whether a prototype is used in production.

The known approaches are not suitable for fast prototyping combined with automated service composition based on functional quality criteria.

The motivated problems lay in the field of IE. IE describes the field of extracting structured information from unstructured text (Cardie, 1997).

Since IE applications often deal with non-deterministic problems, in which boundary conditions may change, e.g., through changes in data formats, (Seidler and Schil, 2011) suggest an approach making these applications more flexible. This approach is limited to the extraction from natural text and does not focus on table analysis. Furthermore, it does not focus on the complete extraction process and leaves out essential steps, e.g., PDF conversion. This approach is not evaluated in practice yet and does not completely address our needs.

4 ARTIFACT PATTERN

The following section introduces the ARTIFACT pattern and its core concepts.

4.1 Artifacts, Components & Pipelines

In the following subsection, we define the basic terms of the ARTIFACT pattern. The base models of our

pattern are *Artifacts*. As shown in Figure 2, we distinguish three different types of artifacts.

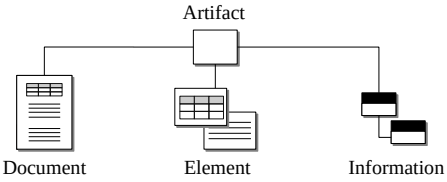


Figure 2: Artifacts.

Documents form the basis for an information extraction process, e.g., PDF documents or text documents. Document parts which have a specific structure are *Elements*, e.g., paragraphs or tables. *Information* is the result of information extraction and is part of elements, e.g., product name.

Components are software modules that solve tasks in an information extraction process. They consume a specific type of artifact and produce another one (c.f. Figure 3).

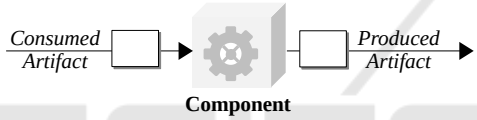


Figure 3: Component.

Converters consume a document of a specific type and produce another document. In other words, they convert a document into another format, e.g., from PDF to text (c.f. Figure 4).

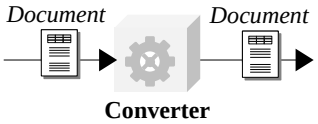


Figure 4: Converter.

Decomposers split a document into its specific documents parts, e.g., paragraphs or tables (c.f. Figure 5). For that, they consume a document of a specific type and produce a list of elements.

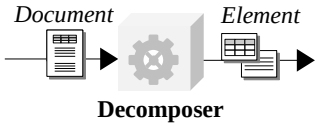


Figure 5: Decomposer.

Extractors consume one or more elements of a specific type and produce an artifact of type information. They perform the actual information extraction (c.f. Figure 6).

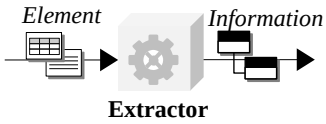


Figure 6: Extractor.

An ordered combination of specific components is called a *Pipeline* and implements a concrete information extraction strategy (c.f. Figure 7).

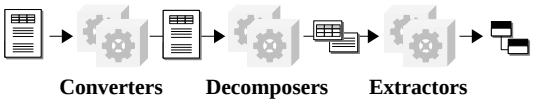


Figure 7: Pipeline.

The definition of consuming and producing artifacts realizes a strict typing. The typing ensures the reusability of components and the goal-specific pipeline generation described in Subsection 4.6.

4.2 Gold-Standard & Document Manager

Automatic information extraction is a non-deterministic problem because the external requirements frequently change, e.g., through new upcoming or changing document formats. Therefore, developers can only make assumptions about the underlying problem.

So-called gold-standard documents form the basis for testing the developed components.

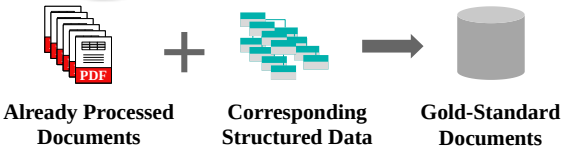


Figure 8: Gold-standard documents.

As shown in Figure 8, a gold-standard document combines an already processed document and its corresponding structured data, i.e., the manually extracted information.

Developers store these documents in a database and use them to test the components. For this, developers compare the expected results with the automatically extracted ones. Testing a component against all gold-standard documents can produce credible quality metrics.

The set of gold-standard documents must be balanced. That means that the ratios of documents corresponding to a specific format must correlate to those of processed documents in reality.

Furthermore, the set of gold-standard documents has to be up-to-date because there can be fundamental changes of document formats in reality over time. To ensure the actuality of the gold-standard document set, ARTIFACT uses the document manager shown in Figure 9.

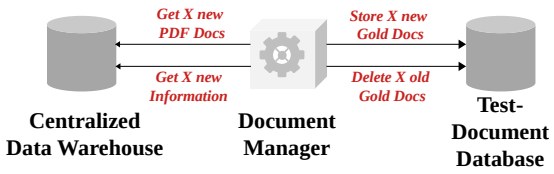


Figure 9: Document manager.

A centralized data warehouse stores the processed documents and their corresponding extracted information. The document manager frequently checks the centralized data warehouse for new documents and corresponding information and automatically updates the set of gold-standard documents. Therefore, it inserts new documents and deletes old ones to keep the set of gold-standard documents at a manageable size and always up-to-date.

4.3 Definition of Subgoals

IE problems are often very complex. Therefore, it is essential to separate the target information model into independent parts. Each information part represents a subgoal that can help to increase the degree of automation successively.

Figure 10: Defining subgoals.

As shown in Figure 10, process experts can split the target data model into several independent pieces of information.

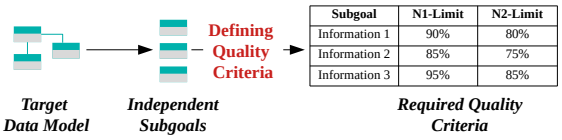
Different strategies are suitable for the extraction of certain information. Developers can, e.g., use rule-based strategies to extract prices from tables or Natural Language Processing (NLP) models to extract product names from text.

As soon as suitable pipelines meet the required quality criteria for a specific subgoal, the system automates affected parts of the IE process.

Due to this, the degree of automation increases successively.

4.4 Formal Definition of Quality Criteria

Formal and measurable quality criteria are needed to determine whether the extraction of independent information can be automated.



Subgoal	N1-Limit	N2-Limit
Information 1	90%	80%
Information 2	85%	75%
Information 3	95%	85%

Figure 11: Definition of quality criteria.

As shown in Figure 11, process experts can define several limits for each information, respectively, each subgoal. The limits represent business requirements.

With the *N1-Limit*, process experts define which percentage of passed tests against the gold standard a pipeline has to reach before the system can use it for automation.

The *N2-Limit* supports to bring pipelines into productive use that do not reach the N1-Limit and therefore would perform not well enough solely. This limit controls which quality two independent pipelines each have to achieve for combined extraction. If the results of the independent pipelines match, the required confidence is given and the result can be used for automation. Due to that, the system will process documents reliably even if single strategies are not yet entirely suitable.

Nevertheless, employees will have to extract the affected information by hand if the conditions mentioned above are not fulfilled.

The formal and measurable quality criteria ensure that business requirements will always be met for specific process parts if they are automated.

4.5 Component Registry

A base concept of the ARTIFACT pattern is the component registry shown in Figure 12.

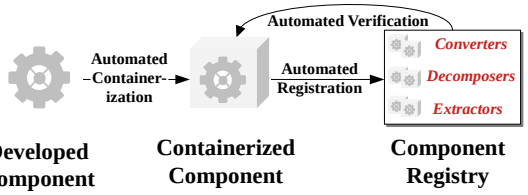


Figure 12: Component registry.

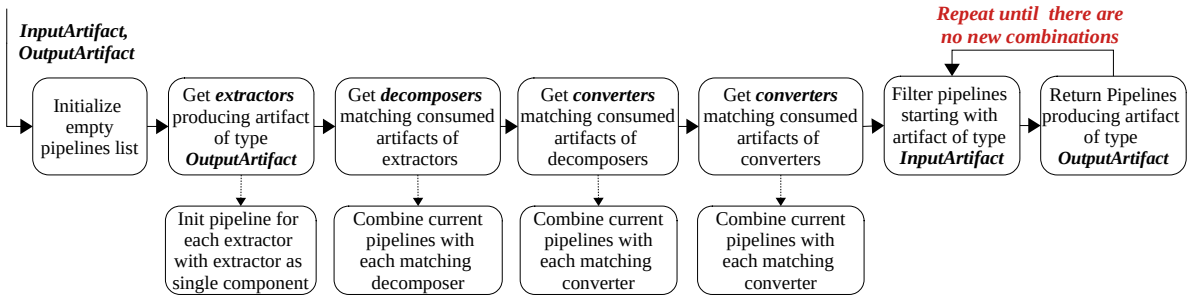


Figure 13: Concept of the pipeline generation algorithm based on backward matching of consumed and produced artifacts.

Developers implement components to solve specific tasks, e.g., converting a PDF document into a text document. Different frameworks and programming languages are better suited than others to solve particular tasks. Therefore, developers implement components as platform-independent microservices.

They are automatically containerized via CI/CD[3] and registered to a central component registry. Each microservice provides an information endpoint that returns information about the task type, the consumed and the produced artifact types.

The component registry sends an example request to a registering microservice and verifies the response. Due to that, all registered microservices are valid.

The component registry handles the communication with the specific microservices and serves as an intermediary for pipeline generation.

4.6 Goal-specific Pipeline Generation

The complexity of automated information extraction leads to a vast number of components solving specific tasks. Different combinations of components can extract the same information, e.g., the product name.

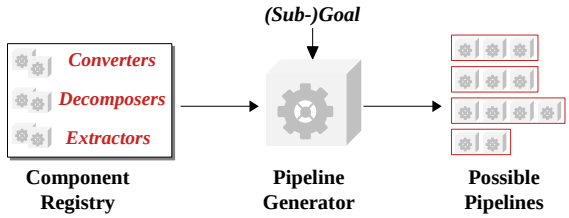


Figure 14: Pipeline generation.

Figure 14 shows pipeline generation as one key concept of ARTIFACT. The pipeline generator performs the automatic generation of possible pipelines depending on a specific (sub-)goal.

The pipeline generator can build possible pipelines through the backward matching of consumed and produced artifacts. Figure 13 shows the concept of the algorithm for the automated pipeline generation.

In the following, we explain the steps of the algorithm using an example. We assume that there are four artifacts (c.f. Table 1) and five components (c.f. Table 2). As mentioned in Section 1, there may be much more components according to the required strategies.

Table 1: Example artifacts.

Artifact	Type
PdfDocument	Document
TextDocument	Document
Paragraph	Element
ProductName	Information

Table 2: Example components.

Component	Type	Input	Output
PdfToTextC	Converter	PdfDocument	TextDocument
TextPreProc	Converter	TextDocument	TextDocument
ParagraphD	Decomposer	TextDocument	Paragraph
ProductNameE1	Extractor	Paragraph	ProductName
ProductNameE2	Extractor	Paragraph	ProductName

Suppose we want to generate all possible pipelines for *ProductName*, the steps of the algorithm look as follows. Extractors producing ProductName are *ProductNameE1* and *ProductNameE2*.

Currently, the generated pipelines only contain the extractors and look as follows:

[ProductNameE1], [ProductNameE2]

All extractors consume a Paragraph as input. Therefore the only matching decomposer is *ParagraphD*. The generated pipelines look as follows:

[ParagraphD, ProductNameE1],
[ParagraphD, ProductNameE2]

All decomposers consume a TextDocument as input. Therefore matching converters are *PdfToTextC* and *TextPreprocessor*. The intermediate results are:

[3]https://docs.gitlab.com/ee/ci/

[PdfToTextC, ParagraphD, ProductNameE1],
[TextPreProc, ParagraphD, ProductNameE1],
[PdfToTextC, ParagraphD, ProductNameE2],
[TextPreProc, ParagraphD, ProductNameE2]

PdfToTextC consumes a PdfDocument, but there are no converters that produce a PdfDocument. *TextPreProc* consumes a TextDocument. Therefore the only matching converter is *PdfToTextC*.
The generated pipelines look as follows:

[PdfToTextC, ParagraphD, ProductNameE1],
[TextPreprocessor, ParagraphD, ProductNameE1],
[PdfToTextC, TextPreProc, ParagraphD, ProductNameE1],
[PdfToTextC, ParagraphD, ProductNameE2],
[TextPreProc, ParagraphD, ProductNameE2],
[PdfToTextC, TextPreProc, ParagraphD, ProductNameE2]

There are no more new combinations, so all possible pipelines are generated. Now filtering only those pipelines whose first converters consume a PdfDocument lead to following result:

[PdfToTextC, ParagraphD, ProductNameE1],
[PdfToTextC, TextPreProc, ParagraphD, ProductNameE1],
[PdfToTextC, ParagraphD, ProductNameE2],
[PdfToTextC, TextPreProc, ParagraphD, ProductNameE2]

4.7 Automated Quality Determination

The qualities of possible pipeline variants may differ widely. According to defined metrics, the system automatically chooses the best pipelines for each (sub-) goal to achieve the highest business value.

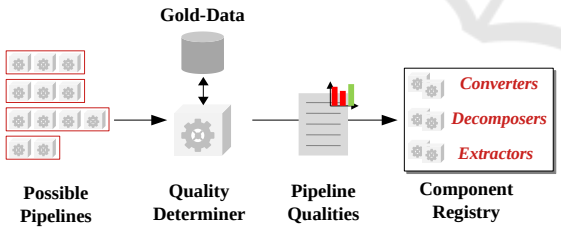


Figure 15: Quality determiner.

Figure 15 shows automated quality determination as a key concept of ARTIFACT. The quality determiner tests each possible pipeline against the set of gold-standard documents. Due to that, it finds the best available pipelines for a specific goal.

The automated quality determination ranks pipelines for each (sub-)goal ordered by the percentage of passed tests. Any change to the component registry or the set of gold-standard documents triggers the determination.

The information about each pipeline quality is sent to the component registry and serves as a basis for the ad-hoc automation at runtime described in Subsection 4.8.

4.8 Ad-hoc Automation at Runtime

It is desirable to maximize the degree of automation and therefore gain business value. At the same time, the system must meet defined quality criteria. Thus, ARTIFACT introduces the ad-hoc automation at runtime shown in Figure 16.

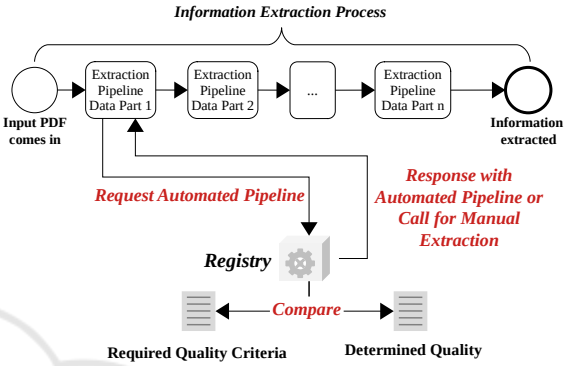


Figure 16: Ad-hoc automation at runtime.

A process engine controls the overall IE process, e.g., BPMN-based. A task in the process model represents the extraction of specific information.

Each step has a decision to determine whether one or two independent pipelines can handle the extraction task according to the defined limits. If there are no suitable pipelines, the process engine will trigger the manual extraction.

Because every change to the system triggers the automatic quality determination, the ARTIFACT pattern ensures ad-hoc automation at runtime.

5 EXPERIMENTAL EVALUATION

In this section, we demonstrate the practical application of our ARTIFACT pattern introduced in Section 4. In the project, we applied our pattern that was motivated in Section 2.

5.1 Defined Subgoals & Quality Criteria

The project's overall goal is automated information extraction of all relevant information from PDF documents. Since the holistic consideration of all information is very complex, we define subgoals following Subsection 4.3.

We split the target data model into independent data parts representing our subgoals. The successful

automation of each subgoal increases the degree of overall automation and therefore gains business value.

From the business point of view, these data parts are of different importance. While price information is more critical for downstream processes, the product's name itself is less important. Therefore, we define different quality criteria for each data part shown in Table 3.

Table 3: Defined subgoals & quality criteria.

Data Part	N1-Limit	N2-Limit
DateOfValidity	90%	75%
BasicPrices	90%	75%
CommodityPrices	90%	75%
SupplierName	90%	75%
ProductName	80%	65%
CustomerGroups	80%	65%
MeteringPrices	80%	65%
ProductType	70%	55%
ProductCategory	70%	55%

We, e.g., define *DateOfValidity* as one independent data part. It describes at which point in time a customer can order a specific product. It is essential for downstream analysis, e.g., time-based price comparisons.

Due to its importance, we define an N1-Limit of 90%. Thus, the system only chooses pipelines for automation that aim at least 90% correctly extracted results in the automated gold-standard test.

If there is no pipeline reaching the limit of 90% in the automated gold-standard test, the system will use the N2-Limit to find alternatives. There must be at least two pipelines that aim 75% each in the test. If two independent pipelines reach this value, the system will pick them for automation. If their results do not match, DateOfValidity will have to be extracted by hand.

5.2 Implemented Components

We are in an early stage of the project so that the number of implemented components steadily increases.

Currently there are five converters (c.f. Table 4), five decomposers (c.f. Table 5) and 14 Extractors (c.f. Table 6)

Table 4: Implemented converters.

Name	Input Artifact	Output Artifact
PopplerPdfToText	PdfDocument	TextDocument
TesseractPdfToText	PdfDocument	TextDocument
LibrePdfToOdt	PdfDocument	OdtDocument
PopplerPdfToImg	PdfDocument	ImgDocument
TextPreProcessor	TextDocument	TextDocument

Table 5: Implemented decomposers.

Name	Input Artifact	Output Artifact
TableBankDec	ImgDocument	Table
CamelotTableDec	PdfDocument	Table
TabulaTableDec	PdfDocument	Table
TextParagraphDec	TextDocument	Paragraph
OdtParagraphDec	OdtDocument	Paragraph

Table 6: Implemented extractors.

Name	Input Artifact	Output Artifact
SimpleRegexDovEx	Paragraph	DateOfValidity
ComplexRegexDovEx	Paragraph	DateOfValidity
RegexBasicPriceEx	Paragraph	BasicPrice
TableBasicPriceEx	Table	BasicPrice
RegexCommodityPriceEx	Paragraph	CommodityPrice
TableCommodityPriceEx	Table	CommodityPrice
NerSupplierNameEx	Paragraph	SupplierName
DictSupplierNameEx	Paragraph	SupplierName
NerProductNameEx	Paragraph	ProductName
NerCustomerGroupEx	Paragraph	CustomerGroup
RegexMeteringPriceEx	Paragraph	MeteringPrice
TableMeteringPriceEx	Table	MeteringPrice
NerProductTypeEx	Paragraph	ProductType
NerProductCategoryEx	Paragraph	ProductCategory

There are several extractors that are based on modern Named Entity Recognition (NER) technologies from the field of NLP, i.e., *NerSupplierNameEx,NerProductNameEx, NerCustomerGroupEx, NerProductTypeEx, NerProductCategoryEx*.

DictSupplierNameEx, e.g., provides an alternative for the extraction of supplier names and is based on classical Regular Expressions (Regex).

5.3 Goal-specific Pipelines & Qualities

Based on the implemented components, there are several possible pipelines per information. The larger the number of components, the more unmanageable is the manual detection of possible combinations.

Table 7: Pipelines per information (initial gold-standard set).

Output Artifact	Possible Pipeline	Best Pipelines	Reached Limit
DateOfValidity	10	92%	N1
BasicPrice	8	60%	-
CommodityPrice	8	55%	-
SupplierName	10	77%	N2
ProductName	5	50%	-
CustomerGroup	5	55%	-
MeteringPrice	8	35%	-
ProductType	5	55%	-
ProductCategory	5	55%	-

Table 7 shows the number of possible pipelines per information and the quality of the best one tested against the initial set of gold-standard documents.

Table 8 shows the quality of the pipelines after the first update of the gold-standard set. The qualities were determined and updated automatically, while there were small quality changes.

Table 8: Pipelines per information (updated gold-standard set).

Output Artifact	Possible Pipeline	Best Pipelines	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%	-

Since there is a pipeline for DateOfValidity reaching 91% (c.f. Figure 17), the N1-Limit of 90% is exceeded. Therefore, the system can automate the extraction step for DateOfValidity.

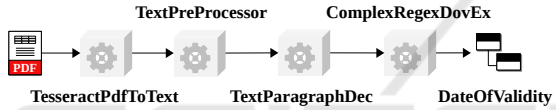


Figure 17: DateOfValidity pipeline.

For the SupplierName, there is no pipeline reaching the required N1-Limit of 90%, but several pipelines reaching the N2-Limit of 75% (c.f. Figure 18 and Figure 19). Hence, the system will automate the extraction step for the SupplierName if two different pipelines return the same result. Otherwise, it triggers manual extraction.

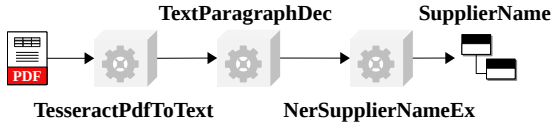


Figure 18: SupplierName pipeline 1.

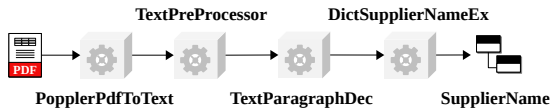


Figure 19: SupplierName pipeline 2.

Due to quality assurance, two employees double-check the manually extracted information. We have collected the data shown in Table 9 through double-checking since the partial automation has been active.

Table 9: Results in production.

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 that 29 documents had to be processed manually. In the case of the 65 matched results, the system extracted 60 correctly.

The extraction step for DateOfValidity is fully automated and reaches the required quality criteria. In 92% of the cases for SupplierName, the registry decided correctly to return the automatically extracted result.

5.4 Implementation

In the following subsection, we present an exemplary implementation of our ARTIFACT pattern.

Due to the nature of microservices, there is no need to use a unified programming language for all microservices. We used a Python stack for the pipeline generator and the conversion, decomposition and extraction components in our implementation. For the component registry and the process orchestrator we also used a Java 17 stack with Maven and Spring Boot[4]. We realize the communication between microservices via Representational State Transfer (REST) calls.

To minimize the manual effort for data models, REST endpoints and client code implementation, we use OpenAPI[5] to define a programming language-agnostic definition of the information above. Via Swagger Codegen[6], we generate server stubs and client SDKs for the specified API.

We containerize every microservice with Docker[7]. Thus, we ensure that the applications run the same way, regardless of the surrounding infrastructure. The containerization also enables us to deploy those microservices on a container-orchestration system like Kubernetes[8].

Figure 20 shows the currently developed microservice architecture. In the following, we describe the most important microservices.

5.4.1 Components

As stated in Subsection 4.1, we divide the extraction of single information into three different types

[4]https://spring.io/
[5]https://swagger.io/specification/
[6]https://swagger.io/tools/swagger-codegen/
[7]https://www.docker.com/
[8]https://kubernetes.io

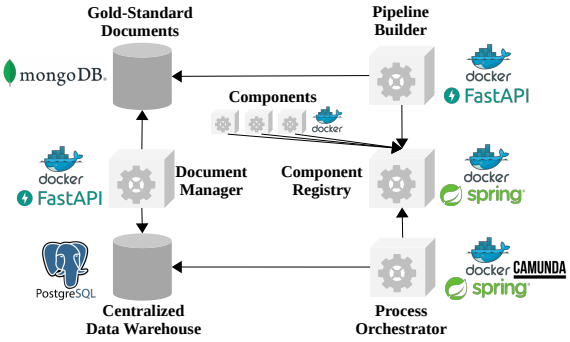


Figure 20: Implemented Microservice Architecture.

of components: converters, decomposers and extractors. We implemented each component as a separate microservice that provides an endpoint for the operation mentioned above. Additionally, every component microservice provides an information endpoint. This endpoint returns the name and version of the microservice. It also returns which type of task it implements and which artifacts it consumes and produces.

Code Listing 1 shows the implementation of the information endpoint of a component. The presented information signals the component registry that this component is a converter that consumes a PDF and returns a text document. Internally the component uses Tesseract[9] to extract text from the PDF document using OCR.

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

Code Listing 1: Information endpoint of a component.

5.4.2 Component Registry

As described in Subsection 4.5, we implement a microservice that manages all components mentioned above. At first, we need to provide the quality criteria mentioned in Subsection 5.1. After that, we can start registering components at the component registry.

Code Listing 2 illustrates the registration of a new component. When a component tries to register itself at the component registry, the registry queries the information endpoint of the component in order to determine its task type. After that, the registry tests the component's endpoint with example data. If this succeeds, the component will be registered. Afterwards, the registry informs the pipeline builder about the new component by forwarding all relevant component information. Since building and evaluating all possible pipelines takes some time, the method does not wait for the pipeline builder's result. Instead, the pipeline builder performs a POST after building and evaluating all pipelines. As a result, the registry is ready for use.

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

  ComponentEndpointInfo info =
      ↪ requestComponentInfo(sock);
  if (verifyEndpoint(info, sock)) {
    Component com = new Component(sock,info);
    allComps.put(com.getName(), com);
    pipelineBuilderService.notify(allComps);
  }
}
```

Code Listing 2: Registration of new components.

5.4.3 Pipeline Builder

In our implementation, we combine the goal-specific pipeline generation from Subsection 4.6 and the automated quality determination from Subsection 4.7 into a single microservice called Pipeline Builder.

Pipeline Builder is a FastAPI[10] web service. It provides endpoints for pipeline generation and quality determination. Code Listing 3 illustrates the determination.

```
@controller.post("/determine")
def post_determine():
  determined_qualities =
      ↪ PipelineBuilderService.
      ↪ determine_qualities()
  return determined_qualities
```

Code Listing 3: Pipeline determination endpoint.

Code Listing 4 shows the steps taken to determine the quality of each pipeline. The endpoint returns the result.

```
def determine_qualities():
  determined_qualities =[]
  gold_documents_request =requests.get('/gold
      ↪ -documents')
```

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

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

```
gold_documents =[GoldDocument.parse_obj(
    ↪ json_data) for json_data in
    ↪ gold_document_request.json()]
for info in Config.get_information():
  for pipeline in self.generate_pipelines(
      ↪ Document.PdfDocument, info):
    passed_tests =0
    for gold_document in gold_documents:
      expected_info =gold_document.
          ↪ get_information(info)
      extracted_info =pipeline.process()
      if expected_info ==extracted_info:
        passed_tests +=1
    quality =passed_tests /len(
        ↪ gold_documents)
    determined_qualities.append(
       PipelineQuality(
         pipeline=pipeline,
         quality=quality
       )
    )
return determined_qualities
```

Code Listing 4: Pipeline quality determination.

5.4.4 Process Orchestrator

The process orchestrator is a Java Spring Boot web service that provides a Camunda[11] process engine. As mentioned in Subsection 4.8, the process orchestrator calls the component registry to extract information from a given PDF document.

In our current state of implementation, the process orchestrator acts as the primary user interface. The user uploads a PDF document via a Camunda form. For each defined extraction task, the orchestrator uploads the document to the component registry with the request to extract a given type of information. If any extraction task requires manual actions, the process orchestrator will prompt the user to enter the missing data. If all information is available, the process orchestrator will store the information in the centralized data warehouse.

Code Listing 5 shows the implementation of a Camunda Service Task that receives a PDF document from the *DOCUMENT* variable and returns the extraction result for a single information type in the variable *RESULT*. If there is no result, the service task will set the value of the *RESULT* variable to *null*. A *null* value marks the document for manual extraction.

[11]https://camunda.com/

```
@Override
public void execute(DelegateExecution exec)
    ↪ throws Exception {
  String resultType = getResultType(exec);

  FileValue documentFile = exec.
      ↪ getVariableTyped("DOCUMENT");
  PdfDocument document = getPdfDocument(
      ↪ documentFile);
  List<Object> result = sendDocumentToServer(
      ↪ document, resultType);
  if (result == null || result.isEmpty()) {
    exec.setVariable("RESULT", null);
  } else {
    exec.setVariable("RESULT", result);
  }
}
```

Code Listing 5: Extraction service task.

6 CONCLUSION

With ARTIFACT, we provide an architectural pattern that ensures the automated service composition into pipelines based on functional business criteria. The provided system automatically chooses the best pipeline by determining all possible alternatives. For that, we adapted the service registry pattern and the service orchestration.

Due to the use of the quality determiner (c.f. Figure 15) and the document manager (c.f. Figure 9) we completely automate end-to-end testing. Thus, we ensure the minimization of testing costs and effort. We have shown that no manual effort is needed to test new components because the system triggers testing automatically when required. Furthermore, we ensure that the set of gold-standard documents always represents current environmental conditions.

The introduced concepts of automated pipeline generation (c.f. Subsection 4.6) and automated quality determination (c.f. Subsection 4.7) guarantee that possible side effects of the systems are automatically detected, e.g., overall quality loss. Hence, we enable fast prototyping and risk-free integration of the newest approaches from research.

Beyond that, ARTIFACT makes manual management reactions to quality changes obsolete. The system decides whether it can use a pipeline according to the required quality criteria or not.

Moreover, through the concept of defining N2-Limits for quality control, we can bring components into productive use that would perform not well enough solely.

ARTIFACT is not limited to information extraction sourcing from natural text. Furthermore, we pro-

vide an approach to implement information extraction for arbitrary documents or data formats, e.g., tables.

Additionally, ARTIFACT addresses more complex IE problems because it includes tasks like converting non-machine-readable into machine-readable documents, e.g., PDF to text.

Due to the application in a real-world project, we have shown that our pattern supports companies to automate their information extraction process successively and gains business value.

7 OUTLINE

In the course of future development, we would like to add a classification mechanism to the information extraction processes. We assume that there are several document classes with different characteristics. Possible pipelines could perform differently to single document classes.

Additionally, we would like to add caching mechanisms to the different pipeline runs. As shown in Section 5, some pipeline parts are recurring when processing a specific document. Due to performance reasons, the system could cache intermediate results of specific steps.

Beyond that, we would like to optimize the choice of possible pipelines if the results were nearly equal. The system should be able to take other metrics like the expected pipeline performance into account when choosing.

REFERENCES

Beck, K. (2003). *Extreme Programming - die revolutionäre Methode für Softwareentwicklung in kleinen Teams ; [das Manifest]*. Pearson Deutschland GmbH, München.

Camposo, G. (2021). *Cloud Native Integration with Apache Camel - Building Agile and Scalable Integrations for Kubernetes Platforms*. Apress, New York.

Cardie, C. (1997). Empirical Methods in Information Extraction. page 15.

Chowdhury, S. R., Salahuddin, M. A., Limam, N., and Boutaba, R. (2019). Re-Architecting NFV Ecosystem with Microservices: State of the Art and Research Challenges. 33(3):168–176.

Chris Richardson (2018). *Microservices Patterns*.

Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R., and Safina, L. (2017). Microservices: Yesterday, today, and tomorrow.

Evans, E. and Evans, E. J. (2004). *Domain-driven Design - Tackling Complexity in the Heart of Software*. Addison-Wesley Professional, Boston.

Fowler, M. (2013). An appropriate use of metrics.

Fuld, I., Partner, J., Fisher, M., and Bogoevici, M. (2012). *Spring Integration in Action -*. Simon and Schuster, New York.

Hanson, C. and Sussman, G. J. (2021). *Software Design for Flexibility - How to Avoid Programming Yourself into a Corner*. MIT Press, Cambridge.

Hashmi, K. A., Liwicki, M., Stricker, D., Afzal, M. A., Afzal, M. A., and Afzal, M. Z. (2021). Current Status and Performance Analysis of Table Recognition in Document Images with Deep Neural Networks.

Hohpe, G. and Woolf, B. (2003). *Enterprise Integration Patterns - Designing, Building And Deploying Messaging Solutions*. Addison-Wesley Professional.

Jamshidi, P., Pahl, C., Mendonca, N. C., Lewis, J., and Tilkov, S. (2018). Microservices: The Journey So Far and Challenges Ahead. 35(3):24–35.

Lewis, J. and Fowler, M. (2014). Microservices.

Marin-Perianu, R., Hartel, P., and Scholten, H. (2005). A Classification of Service Discovery Protocols. page 23.

Newman, S. (2015). *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, first edition edition.

Peltz, C. (2003). Web services orchestration and choreography. 36(10):46–52.

Rubin, K. S. (2012). *Essential Scrum - A Practical Guide to the Most Popular Agile Process*. Addison-Wesley Professional, Boston, 01. edition.

Schmidts, O., Kraft, B., Schreiber, M., and Zündorf, A. (2018). Continuously evaluated research projects in collaborative decoupled environments. In *2018 IEEE/ACM 5th International Workshop on Software Engineering Research and Industrial Practice (SER IP)*, pages 2–9.

Schreiber, M., Kraft, B., and Zündorf, A. (2017). Metrics Driven Research Collaboration: Focusing on Common Project Goals Continuously. In *2017 IEEE/ACM 4th International Workshop on Software Engineering Research and Industrial Practice (SER IP)*, pages 41–47.

Seidler, K. and Schil, A. (2011). Service-oriented information extraction. In *Proceedings of the 2011 Joint EDBT/ICDT Ph.D. Workshop on - PhD '11*, pages 25–31. ACM Press.

Tarr, P., Ossher, H., Harrison, W., and Sutton, S. (1999). N degrees of separation: multi-dimensional separation of concerns. In *Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No.99CB37002)*, pages 107–119.

Voron, F. (2021). *Building Data Science Applications with FastAPI - Develop, manage, and deploy efficient machine learning applications with Python*. Packt Publishing Ltd, Birmingham.

Walls, C. (2015). *Spring Boot in Action -*. Simon and Schuster, New York.