CZECH TECHNICAL
UNIVERSITY
IN PRAGUE

F3

Faculty of Electrical Engineering
Department of Computer Science

Bachelor's Thesis

Data warehouse extension DAFOS

Jevhen Olehovyč Ponomarenko
Software Engineering and Technology

Prague, June 2020
Supervisor: Ing. Jiří Šebek

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Ponomarenko** Jméno: **Jevhen Olehovyč** Osobní číslo: **465825**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Softwarové inženýrství a technologie**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Rozšíření datového skladu DAFOS

Název bakalářské práce anglicky:

Data warehouse extension DAFOS

Pokyny pro vypracování:

Starfos je fulltextový vyhledávač nad datovým skladem Dafos, agregátorem veřejných datových zdrojů z prostředí veřejné podpory, výzkumu a vývoje. Datový sklad má být v blízké budoucnosti rozšířen o prezentaci dat z nového datového zdroje: European Patent Office. S narůstajícími nároky na automatizaci stahování a úpravu dat se nabízí možnost aplikace nových nástrojů pro optimalizaci těchto procesů.
Zadání:
1. Popište datový sklad DAFOS
2. Popište technologie a postupy používané v datových skladech a ETL procesech
3. Realizujte požadavky:
-Zrusení závislostí importerů.
-Staging area s možnosti dotazování nad daty.
-Refactoring datového modelu: větší granularita tabulek.
-Centralizované místo pro ukládání logů s možnosti dotazování se nad daty.
-Zrusení závislostí importovacích skriptů, v ideálním prípadě musí být vsechny
-kroky idempotentní.
-Verzování dat
-Historie změn dat uživatelem
-Automaticky generované reporty o nově importovaných datech
4. Analyzujte technické požadavky na vylepšení systému z hlediska rozšířitelnosti a udržitelnosti
5. Přidejte nový datový zdroj European Patent Office - databáze patentů
6. Otestujte implementovanou funkcionalitu na úrovní jednotkových a integračních testů

Seznam doporučené literatury:

1. VAISMAN, Alejandro a Esteban ZIMÁNYI. Data Warehouse Systems [online]. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014 [cit. 2019-10-09]. DOI: 10.1007/978-3-642-54655-6. ISBN 978-3-642-54654-9.
2. KIMBALL, Ralph a Margy ROSS. The data warehouse toolkit: the definitive guide to dimensional modeling. Third edition. Indianapolis, IN: John Wiley & Sons, [2013]. ISBN 978-1118530801. KIMBALL,
3. Ralph a Joe CASERTA. The data warehouse ETL toolkit: practical techniques for extracting, cleaning, conforming, and delivering data. Indianapolis, IN: Wiley, c2004. ISBN 978-0764567575.

Jméno a pracoviště vedoucí(ho) bakalářské práce:

Ing. Jiří Šebek, kabinet výuky informatiky FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **14.02.2020** Termín odevzdání bakalářské práce: **22.05.2020**

Platnost zadání bakalářské práce: **30.09.2021**

Ing. Jiří Šebek
podpis vedoucí(ho) práce

podpis vedoucí(ho) ústavu/katedry

prof. Mgr. Petr Páta, Ph.D.
podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

.
Datum převzetí zadání

Podpis studenta

Acknowledgement / Declaration

I want to thank my supervisor Jiří Šebek and colleagues Robert Shönfeld and Radovan Lupták, for valuable advice while writing this thesis. Unique appreciation goes to my family and girlfriend for patience and support during my studies.

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis. I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on June 22, 2020

. . . .

Abstrakt / Abstract

Tato bakářská práce se zabývá automatizaci ETL procesů v datovém skladu Technologické agentury České republiky - DAFOS. Po nastavení požadavků na nový ETL framework, analyzuje možná průmyslově využívaní řešení pro orchestraci ETL procesů a implementuje strategie pro řešení problému, které vznikly ztrátou kontroly nad během ETL procesu jako jsou například: dohled nad změnami dat nebo místo pro ukládání dočasných dat z možnosti dotazování nad nimi. Na základě definovaných postupů integruje nový datový zdroj do datového skladu: kolekci patentů z OPS. Nov technologická řešení byla zařazená do datového skladu: MongoDB jako místo pro ukládání dočasných dat a Apache Airflow pro orchestraci ETL procesů.

Klíčová slova: Apache Airflow; Django; DAFOS; Starfos; datové sklady; ETL; bakalářská práce; TA ČR; MongoDB; European Patent Office;

Překlad titulu: Rozšíření datového skladu DAFOS

This bachelor thesis deals with the automation of ETL in the DAFOS data warehouse. After laying out the requirements for the new ETL framework, it analyzes possible approaches for orchestration of the ETL processes and implements strategies for sub-problems that arose from the full automation. The new patent data set was added into the warehouse using the defined approaches. The solution provides answers to many sub-problems that resulted from shifting control of the ETL from a developer: governance of the newly modified data or data staging area with query capabilities. New technologies were introduced into the technological stack of the warehouse: MongoDB as a staging area solution and Apache Airflow for allowing a unified approach to defining and scheduling the ETL processes.

Keywords: Apache Airflow; Django; DAFOS; Starfos; data warehouse; ETL; bachelor thesis; data governance; MongoDB; European Patent Office; TA CR;

Contents /

Tables / Figures

Chapter 1
Introduction

ETL processes are one of the most vital parts of the data warehouse. As the data warehouse grows in data, manual scheduling of these tasks becomes insufficient: dependencies of these tasks have to be well documented, and the whole process requires attention from a developer. A couple of solutions answering the questions on how to design and schedule ETLs effectively in Python were developed in recent years and serve as an industry-standard nowadays. ETL orchestration is not the only requirement needed for better maintainability and extensibility of the DAFOS data warehouse. Subproblems covered in the thesis also include data staging area and solutions for better control of incoming data.

1.1 Structure of the document

This document is composed of four main chapters. In **Chapter 2** I define key concepts later discussed in the document. I analyze available solutions for orchestration of ETLs in **Chapter 5** and then automate current scripts handling integration of IS VaVaI data set in **Chapter 6** and integrate a new data set of patents in **Chapter 7**.

Chapter 2
Data warehousing

In the late 1970s, most organizations used relational database systems for storing vital information and supporting day-to-day operations [1]. These systems proved to be insufficient as increasing market requirements led to the need for accessing the right information at the right time. As a result, a new set of tools was formed to better support new requirements for improving the operations of businesses. In the 1990s, data warehousing and online analytical processing (OLAP) were developed to help the decision-making process. This involves a set of tools, algorithms, and architectures to allow the accumulation of information from various data sources over a period of time to enable the analysis of its evolution and discovery of strategic information. The universally accepted definition of a data warehouse was developed by Bill Inmon in the 1980s and is "a subject-oriented, integrated, time-variant and non-volatile collection of data used in strategic decision making.". [2]

- subject-oriented - data is divided into units (e.g tables) by the type, not by the source of data,
- integrated - warehouse integrates data coming from various sources and in different formats,
- non-volatile - data warehouses are designed to be "read-only", so after data is saved it should not be manually edited [3].

2.1 ETL processes

ETL processes are essentially the most important component of DWH [2]. ETL stands for *Extract*, *Transform*, *Load* as these are the stages of data during the process. A properly designed ETL system extracts data from various sources, transforms the data using quality and consistency standards, so that separate sources can be combined. It then saves them in the destination system in a format that can be later easily used by other applications.[4]

The design and implementation of these processes can be a very demanding task, and even though it is a functionality hidden from the users, it easily consumes 70 percent of the resources needed for implementation and maintenance of a typical data warehouse.[4] These components play a key role in defining a ETL framework[5]:

- Triggers - event listeners, perform logic when some criteria is met,
- Database tables - destination formations used to store the information,
- Software libraries - libraries produced by third parties that can be used for more straightforward data transformation,
- Scripts - abstractions that perform business logic,
- Notifications - real-time analytics over the system,
- Schedulers - modules that orchestrate the whole framework.

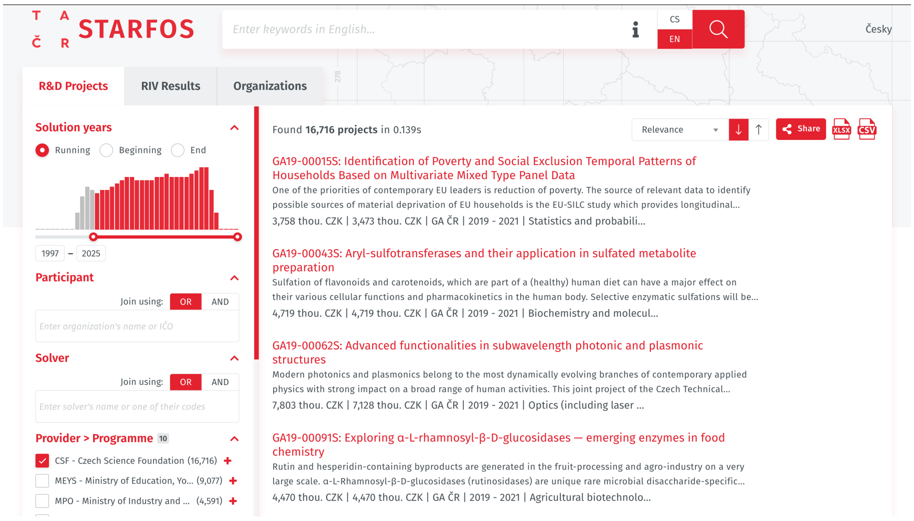
Chapter 3
Technology Agency of the Czech Republic

The Technology Agency of the Czech Republic, *an organizational unit of the state*, is a state institution that oversees the process of public contests in which funding is allocated to applied research projects. Main objectives of TA CR include:

- analysis and realization of applied research programs,
- evaluation of completed R&D projects,
- provision of funds R&D projects,
- communication between business sector and research groups.

3.1 DAFOS data warehouse

DAFOS is one the key results of the Proeval[6] project. DAFOS data warehouse serves as a source of data used for purposes of Starfos full-text search engine, depicted in **Figure 3.1** (for projects and results in the field of research, experimental development, and innovations that have been supported by public funds of the Czech Republic). DAFOS is not a platform for collecting data related to the research and application process from users, DAFOS only aggregates the data from various sources to later present them in a more readable way for the public.


T A Č R
STARFOS
Enter keywords in English...
CS
EN
Česky
R&D Projects
RIV Results
Organizations
Solution years
● Running
○ Beginning
○ End
1997 – 2025
Participant
Join using: OR AND
Enter organization's name or IČO
Solver
Join using: OR AND
Enter solver's name or one of their codes
Provider > Programme 10
☑ CSF - Czech Science Foundation (16,716)
☐ MEYS - Ministry of Education, Yo... (9,077)
☐ MPO - Ministry of Industry and ... (4,591)
Relevance
Share
XLSX
CSV

Found **16,716 projects** in 0.139s

GA19-00015S: Identification of Poverty and Social Exclusion Temporal Patterns of Households Based on Multivariate Mixed Type Panel Data
One of the priorities of contemporary EU leaders is reduction of poverty. The source of relevant data to identify possible sources of material deprivation of EU households is the EU-SILC study which provides longitudinal...
3,758 thou. CZK | 3,473 thou. CZK | GA ČR | 2019 - 2021 | Statistics and probabili...

GA19-00043S: Aryl-sulfotransferases and their application in sulfated metabolite preparation
Sulfation of flavonoids and carotenoids, which are part of a (healthy) human diet can have a major effect on their various cellular functions and pharmacokinetics in the human body. Selective enzymatic sulfations will be...
4,719 thou. CZK | 4,719 thou. CZK | GA ČR | 2019 - 2021 | Biochemistry and molecul...

GA19-00062S: Advanced functionalities in subwavelength photonic and plasmonic structures
Modern photonics and plasmonics belong to the most dynamically evolving branches of contemporary applied physics with strong impact on a broad range of human activities. This joint project of the Czech Technical...
7,803 thou. CZK | 7,128 thou. CZK | GA ČR | 2019 - 2021 | Optics (including laser ...

GA19-00091S: Exploring α-L-rhamnosyl-β-D-glucosidases — emerging enzymes in food chemistry
Rutin and hesperidin-containing byproducts are generated in the fruit-processing and agro-industry on a very large scale. α-L-Rhamnosyl-β-D-glucosidases (rutinosidases) are unique rare microbial disaccharide-specific...
4,470 thou. CZK | 4,470 thou. CZK | GA ČR | 2019 - 2021 | Agricultural biotechnolo...

Figure 3.1. A view of the Starfos search engine

3.2 Domain

The domain of data stored in DAFOS is projects and results in the field of research, experimental development and innovations that have been supported by public funds of the Czech Republic.

The domain model of key entities is presented in **Figure 3.2**.

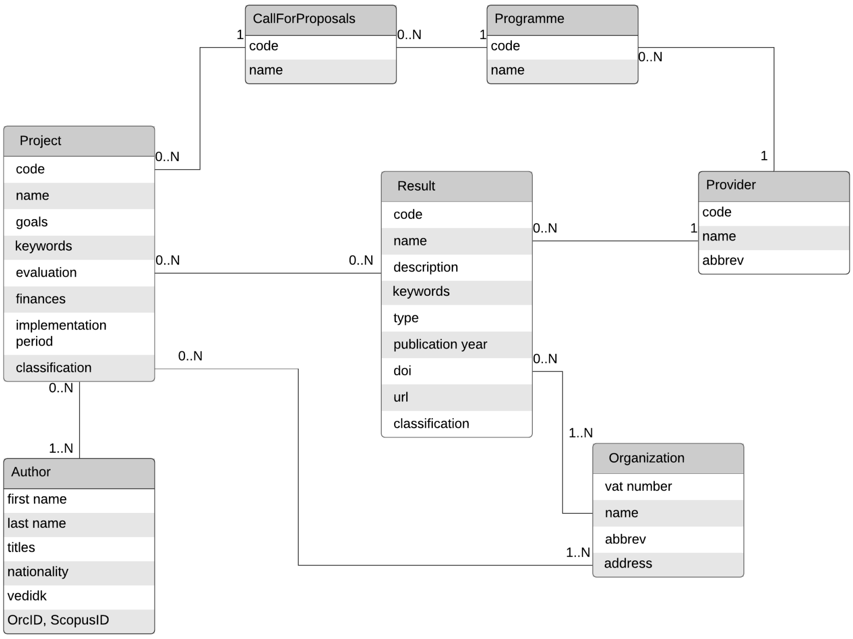


Figure 3.2. The domain model of critical entities in the warehouse

Project

The table contains information about projects, a particular R&D activity with a defined goal; there are over 50 thousand records currently present in the warehouse.

Results

The table contains metadata about scientific publications, patents, trademarks, and other forms of applied research results; there are currently over 1 million records present in the database.

Organization

The table contains information on real-world entities of various legal statuses i.e., business entities or universities. There are over 7 thousand records present.

3.3 Architecture

DAFOS warehouse has a centralized architecture [7] with the addition of the search engine part. The architecture of the warehouse was influenced by the need to support predefined queries over the data. The relational database is not the only destination of the data; data gets serialized and indexed by a Solr engine before it is presented to the end-user. This kind of architecture proved to be one the easiest architectures to implement and perfectly solves its purpose. No single architecture is dominant in terms of information and system quality [7] and DAFOS is not an exception; the decision to use centralized architecture came solely from the requirements: start small and deliver short-term benefits while having a long term plan. The high-level view of the architecture is presented in **Figure 3.3**.

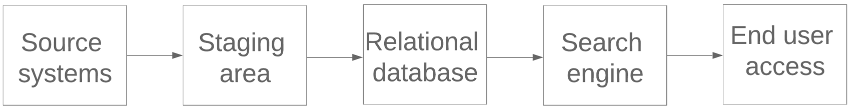


Figure 3.3. The high-level view of the architecture

Chapter 4
Software requirements specification

This chapter outlines software requirements for the new ETL framework that will serve as a platform for integrating new data sources into the DAFOS data warehouse. The main aim of this document is to define new requirements based on experience with the current system.

4.1 Surrounding the requirements

Requirements for the new ETL framework are divided into three categories: business needs, data integration, and software requirements. The software requirements logically arise from the requirements of this thesis, which are:

1. decoupling importing scripts,
2. staging area with query capabilities,
3. refactoring data model,
4. centralized log storage with query capabilities,
5. idempotency of individual importing steps,
6. data versioning,
7. tracking manual changes of data,
8. reports about newly changed data.

4.1.1 Business needs

The biggest shortcoming of the DAFOS warehouse is how frequently data gets updated. As mentioned earlier, the process of importing data needs to be completely redone on every major update of data, and requires a lot of manual steps. Therefore, data gets updated approximately once every three months. A new ETL system will need to address this issue. The fully automated process comes with high risks; the developer team is no longer in charge of the process. Thus, there is a need to report changes in data that take place. One other point is that the final costs of the solution should be taken into account.

A Summary of the business requirements:

- Automation - ETL processes need to be fully automated and scheduled with little to no effort from the developer side,
- Data lineage - the need for tracking changes of data,
- Cost - cost to maintain, develop and run the solution should be minimal.

These requirements are closely related to requirement number **1, 8, 6**.

4.1.2 Data integration requirements

There are many data sources with a variety of output formats producing data that is later stored in multiple places. The new ETL system needs to integrate these sources and provide a common interface to interact with them. As the number of source systems grows, need for unified storage of credentials for access becomes apparent. In order to have a better overview of incoming data, the data staging area needs to provide an easy-to-use interface for querying the data and sharing analytical information about them.

A summary of the data integration requirements:

- Common interface for loading data - interface for importing data into a data warehouse should be as general as possible to make integrating a new source easier,
- A set of tools for data extraction - set of utility functions/modules for standardized interaction with data staging area,
- Data staging area - data should be accessible in the source format by the analytical and developer team before it is loaded into the destination database. Querying of the data needs to be supported.

These requirements are closely related to requirement number **2, 5**.

4.1.3 Software requirements

For a smoother transition, legacy processes that handled importing logic need to be fully supported into the new system, and their integration should be straight-forward. The new ETL system needs to address the issue of the speed of ETL processes. Parallelization of some steps in ETL processes is required and, therefore, a new system will act transparently when these requirements arise. Even though the chosen method will be tightly coupled with the current stack used in the project, it will still allow integrating new technologies into it, e.g. databases, frameworks for processing data.

Summary of the software requirements:

- Support for current modules - the absence of the need to completely rewrite the current system will help in the adoption of it,
- Parallelization - each component of the ETL process will be easily parallelized if it is allowed to use multiprocessing
- Out of the box integration of other services - services like Google Cloud Platform can be easily integrated into the ETL process.

Chapter 5
An analysis of available workflow orchestration solutions

It was decided that to allow automating of ETL scripts, a new framework or service would need to be integrated into the warehouse. Another solution could leverage the use of cron, but this tool proved to be insufficient because of the lack of triggering capabilities, i.e., run task A if task B had finished successfully.

There is a wide variety of systems used for the management of ETL workflows. Some of them are robust Saas solutions, while others are just simple libraries. The primary limits of Saas systems are the technologies surrounding them: they are deployed in the cloud and sometimes require the use of specific technologies designed for the particular cloud solution, i.e., AWS S3 or GCP . There are great automation frameworks available for Hadoop and Spark; however, these are not suitable within our scenario, since neither of these systems are currently utilized in DAFOS nor do they pose any benefits in the current state of the warehouse.

5.1 Scope and purpose

There is only a handful of open-source workflow managers that allow full control over the ETL life cycle, i.e. execution and scheduling of scripts, their dependency management and overview over the state of the scripts. Tools chosen for the analysis are expected to perfectly fit into the technology stack used in DAFOS. This means they are able to run python code and easily communicate with other services such as the database server and Solr server. The final candidate will support the biggest subset of the requirements defined in **Chapter 4**.

5.1.1 Luigi

Luigi[8] is an open-source tool for managing ETL pipelines. It was developed in Spotify. Luigi is implemented as a service: a web server that serves as a front-end for tracking the state of pipelines. Luigi was designed to run on a single computer, but it, nevertheless, allows the concurrent execution of tasks. Unfortunately, Luigi does not provide a native method of handling the scheduling of jobs and, thus, an alternative method would have to be developed.

Primitives

The data pipeline, represented as a Python class, consists of *Task* classes. The task interface provides these methods:

- requires() - definition of dependencies for the task,
- output() - method defining outputs of the task, the method has to return persisted entity (value written to a file system, populated database table etc.) or another Task to declare dependency,
- run() - a method that handles main logic of the step.

Details

In order to provide a comfortable interface, all outputs of each task have to be stored on disk. The dependencies are represented as dependencies on targets, not tasks by themselves. This allows a secure mechanism for deciding whether a task is finished. If the target is saved on disk, the task execution was done.

5.2 Apache Airflow

Apache Airflow[9] is an open-source project developed at Airbnb in 2014 that was merged into Apache Software Foundation in 2016. Airflow is a tool that provides the ETL team with everything associated with the heavy plumbing. Air-flow is implemented as a service running a server that serves as an adminis-trative and monitoring interface to the underlying processes. It also takes care of scheduling tasks and notifying the ETL team if any errors occurred. The package is available to download through PyPI.

Primitives

The data pipeline, DAG, consists of **Operator or Sensor** classes. Task interface provides these methods:

- DAG - directed acyclic graph: data-pipeline, and its dependencies representation,
- Operator - a node of the DAG, a wrapper around a callable that can be used to define a task. This abstraction allows for a better definition of the atomic steps of the pipeline.

Details

Airflow offers everything needed to successfully automate workflows that communicate with external services, i.e. managing connections, configurable parallelization, and much more.

The only downside of this tool is resource consumption. Airflow also introduces the need to maintain a new database for storing metadata about individual tasks. Airflow definitely seems to be the most massive tool discussed in this chapter.

5.3 Dagster

Dagster[10] is the youngest framework discussed. It was built with different principles in mind to provide some extra functionality compared to the other tools. Its architecture is very similar to the ones discussed before (a back-end server used for managing and monitoring of pipelines).

Primitives

- solid - unit of computation with inputs and outputs defined
- pipeline - composition of solids

Details

Dagster introduces a new feature not available in other tools: the static testing of solids. Every time the pipeline is to be executed, a set of tests is run against the schema of solid outputs and inputs. It also allows for the customization of a scheduler and provides a default cron scheduler.

Unfortunately, Dagster does not allow the dynamic creation of solids from code nor the parallelization of tasks out of the box. Another service would have to be put in place (e.g., Celery) in order to allow horizontal scaling of workers.

5.4 Final Conclusion

The tools discussed in this chapter are very similar to each other; all of them were built with different principles in mind, however. The question of finding the best candidate is somewhat subjective, one might prefer the external configuration of tasks using XML files over database tables, and technical at the same time.

Feature	Airflow	Luigi	Dagster
Pipeline visualization	yes	yes	yes
Pipeline configuration decoupled from code	yes	no	yes
Utilities for testing	partially	no	yes
Paralelization out of the box	yes	yes	no
Scheduling	yes	yes	yes
Triggering of tasks based on state of the previous tasks	yes	no	yes
Centralized log storage	yes	no	yes
Support for workload distribution over more computing	yes	no	yes
Static pipeline testing	no	no	yes

Table 5.1. A comparison of features available by each tool

Looking at table 5.1, Dagster covers nearly all elements needed for a complete ETL solution, and Luigi comes out as the most minimalistic solution. The question definitely boils down to choosing between Airflow and Dagster, because Luigi by itself does not cover as many of the requirements discussed in **Chapter 4** as the other tools. Before choosing the final candidate, both tools had to be manually tested.

The testing was accomplished by writing a simple data pipeline for loading a patent classification from and focusing on how easy it is to write and test the pipeline, parallelize the pipeline, and find documentation.

Airflow came out as the winning solution because it provides an easy to use interface for parallelization, even for individual parts of the pipeline. This is a much-needed requirement that was not satisfied by Dagster by default. The strict definition and validation of outputs and inputs of solids enforced by Dagster felt just too limiting to work with. The parameters of Airflow DAGs are stored in the code, but the tool provides a standard interface for storing all other metadata needed for pipeline (i.e., connections to source systems and services). This seems like a better approach as opposed to storing configurations for individual pipelines in XML files stored on disk. Another big difference between Dagster and Airflow is that Dagster does not allow the dynamic creation of pipelines in code. This limits the way the pipeline can be composed, and creating a number of tasks based on the responses from external systems becomes hard to achieve.

Airflow came out as the most mature and robust solution with a great user base. Even though it has some disadvantages in the provided utilities for testing pipelines, it still provides the ETL team with everything needed for running pipelines.

Chapter 6
The automation of the current ETL processes

The current implementation of the ETL framework in DAFOS is solely based on custom python scripts that are divided into only two stages: download (Extract) and import (Transform-Load). These scripts are represented as Django management commands for the purposes of using the Django context, primarily Django ORM inside the script. The scripts serve as an interface between developer and classes that perform the actual logic. Classes are divided based on the file format that is being consumed (XML, JSON, CSV) and the source of the data. Downloaded data is stored on the disk as raw files (untransformed XML files, transformed and aggregated zipped JSON files or responses from APIs cached in a sqlite database). This approach is highly inconsistent and requires heavy documentation.

The automation of the whole process is non-trivial mainly because source systems do not provide good support for incremental updates. Therefore, every time the data needs to be updated, the whole database is built from the ground up. This is done because there is no way to decide what records have been changed. Sometimes these drawbacks require the process to be performed on the testing server first, and after it is fully completed, the data is dumped from the destination database into a file and later loaded into the production database. During this phase, the production app is unavailable, so the process usually takes place at night, when there is a minimum of users accessing the site.

Since the process of importing R&D results is sub-optimal, this process takes around 30 hours to finish, as there are many other entities present in response (such as organizations, solver, etc.). The current ETL process requires the developer's attention because the current state of the script can only be examined by checking the logs. This is because a cumbersome full update of the data happens around twice a year, and takes a big bite of the commercial potential of the whole solution.

6.1 Analysis

6.1.1 As-is state

The key source of data is IS VaVaI[11], other sources include ARES[9] and Google Places API. A high level view of components taking part in the data manipulation process is presented in **Figure 6.1**

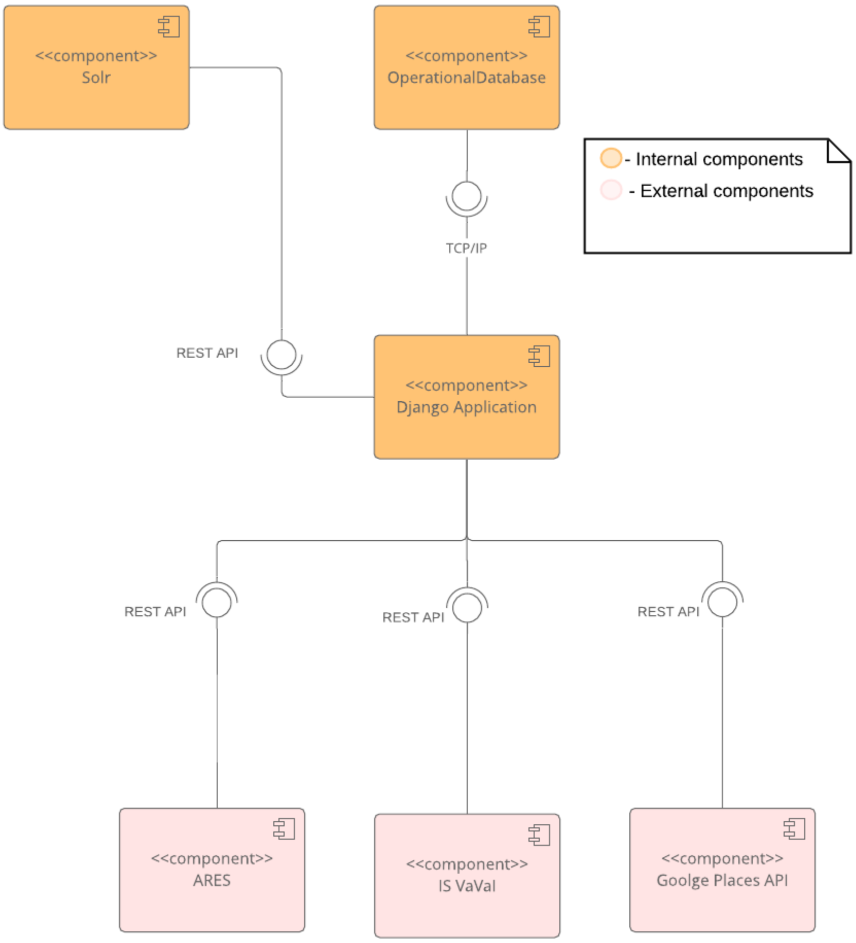


Figure 6.1. The structure of components responsible for data retrieval and manipulation

- IS VaVaI - information system of R&D operated by the Office of the Government of the Czech Republic
- ARES - registers of Economic Subjects / Entities
- Google Maps Platform - geo data
- static files - enumerations downloaded from various sources
- data patches - changes of data performed by a human, these changes overwrite data downloaded from source systems.

Data downloaded from these sources go through a complex data pipeline to later be stored and indexed in Solr. Records downloaded from source systems are stored on disk in the form of JSON and XML files; this allows us to minimize the load on the source systems because data do not have to be queried several times. The contents of these files are read by importing scripts, normalized, and saved into a relational database using Django ORM. When all the data from the staging area are saved, Solr imported scripts are run to denormailize the data and populate the Solr document store using HTTP API. This process is outlined in **Figure 6.2**.

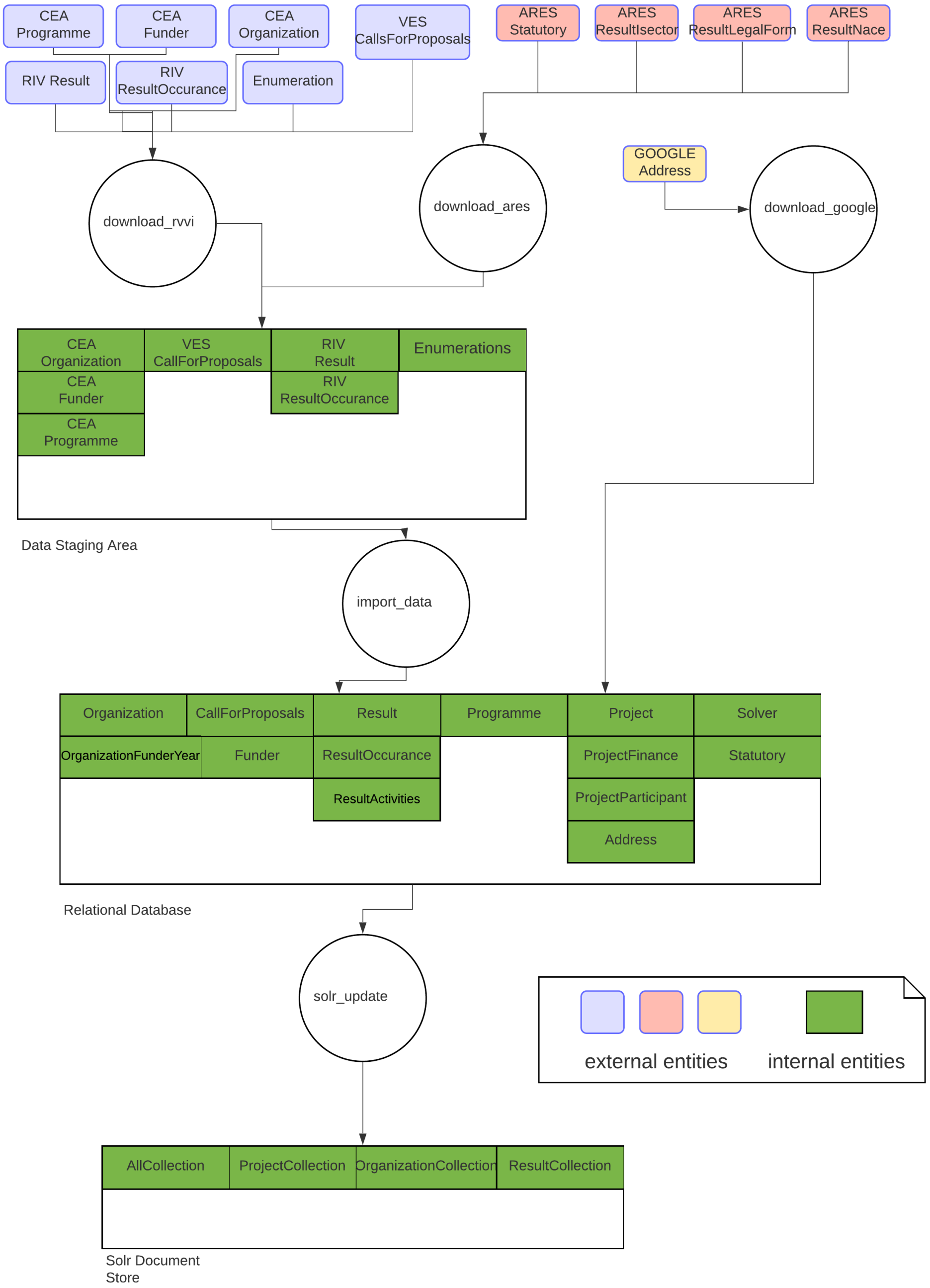


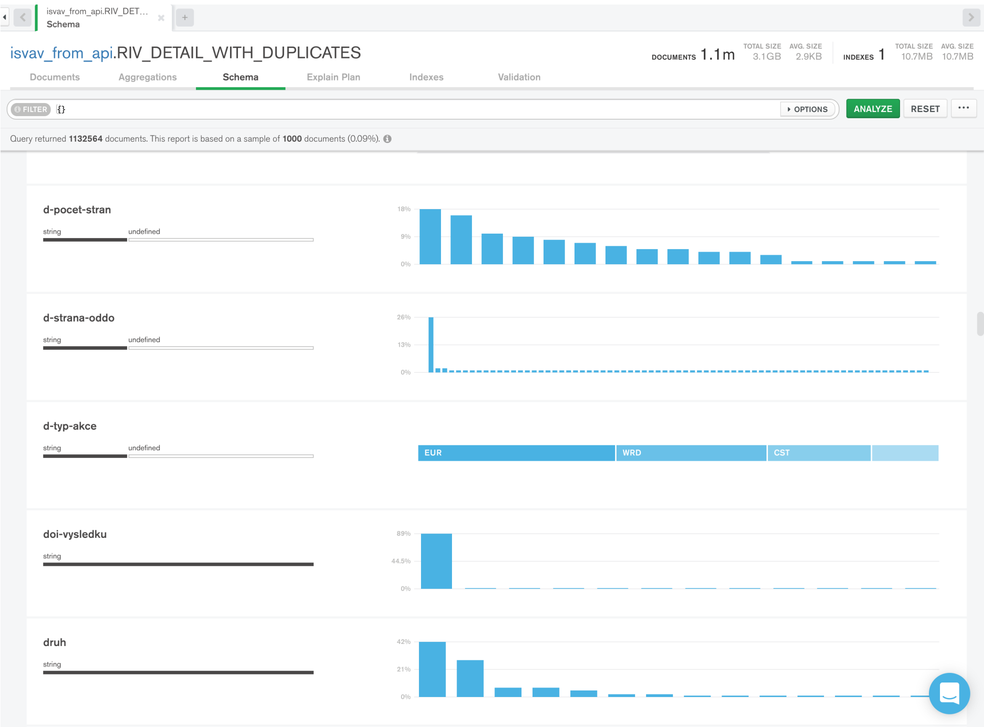
Figure 6.2. The flow of data throughout the warehouse

6.1.2 To-be state

To ensure the least amount of write operations to the staging area and relational database, a new extra field was appended to every record stored: data hash. This field is a hash computed from the contents of the JSON files converted to Python dictionaries, so every time we want to download a new record, we have to check the data hash field first, if it matches the value stored in the staging area, this means that the record was not changed and we can skip it. The old way of importing data did not only go through business logic changes, changes in the technologies used were also present.

For better manipulation of the data in the staging environment, the file system had to be replaced by something more robust that would allow us to better query and analyze the data in the staging environment. MongoDB was chosen as the best candidate for the task. Here are the key benefits of using MongoDB instead of the file system to store JSON files:

- Schemaless documents - schema of the stored data does not need to be known in advance, nevertheless it is possible to define it and require all documents to adhere to it.
- Querying of the data - MongoDB allows us to filter the data in the collection. We can easily find records that match our query in a matter of seconds using a standard query language.
- Parallelization - write and read operations can be quickly done in bulk [12]. This allows us to speed up the process without using any programming constructs like spinning up threads and processes.
- Analysis - MongoDB Compass (GUI tool for a view of the database) has a dominant feature of analyzing field mappings of the data illustrated in **Figure 6.3**. This serves as an excellent tool for analyzing of the data types and statistics of field values.
- Sharing of data - the analytical team can easily access the data through GUI tools and share insights using standard approaches as opposed to getting access to the actual machine, using ssh to view and download the files and later process them.



Query returned **1132564** documents. This report is based on a sample of **1000** documents (0.09%).

Figure 6.3. A view of an analysis of MongoDB collection

These changes represent a significant shift in the architecture of the whole solution reflected in **Figure 6.4** . The final solution was heavily influenced by the need to keep as much of the original code base as possible.

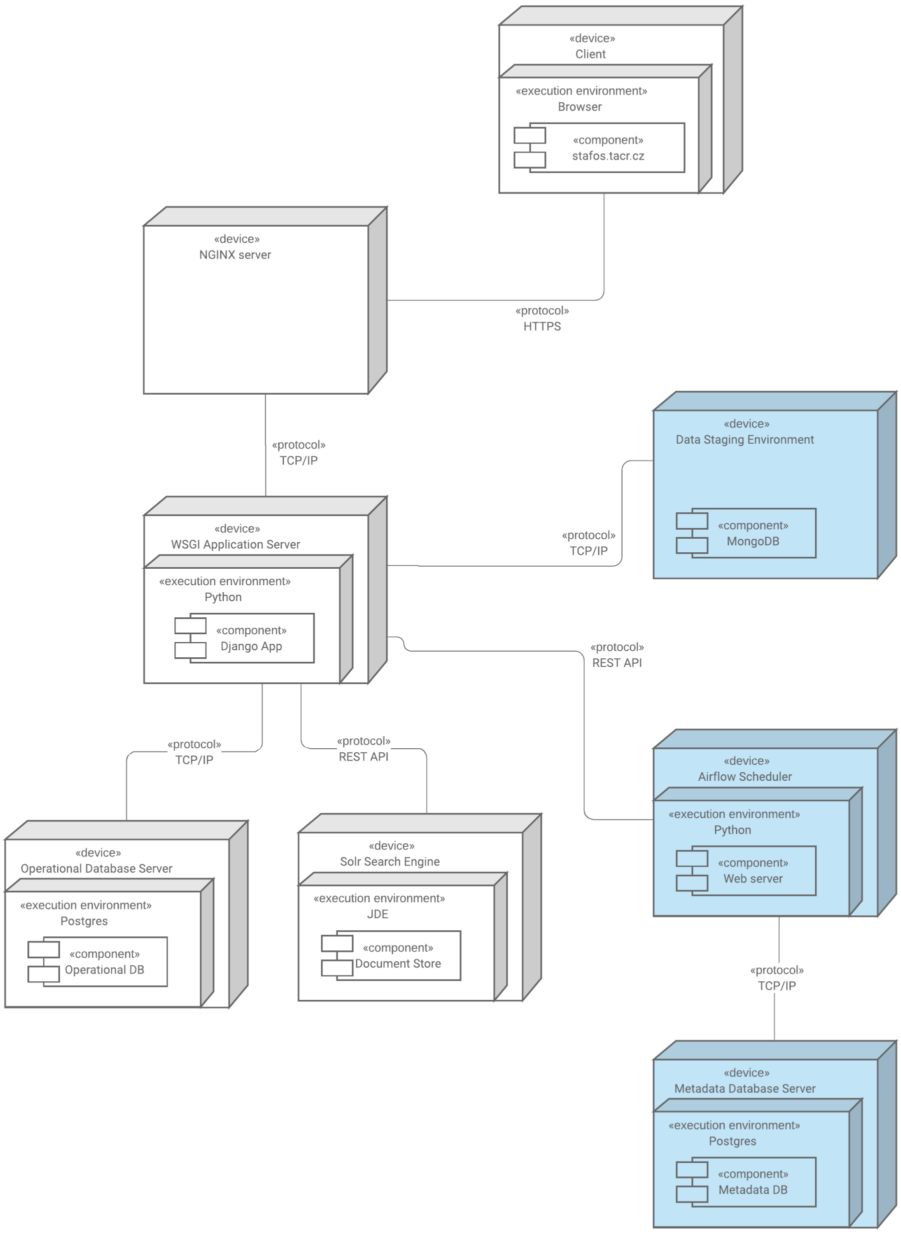


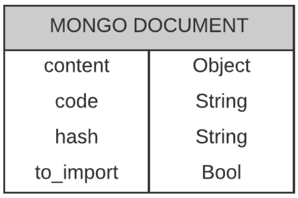
Figure 6.4. Changes in the architecture of the system, added systems are depicted in blue

6.2 Implementation

Apache Airflow was chosen as the system to handle the automation of the scripts. The way Airflow handles the automation was perfect for our use case: a class-based approach for defining logic that needs to be automated. It was straightforward to integrate old importing classes into Airflow because only one new Airflow Operator was created and later applied to all of them.

6.2.1 Data staging area

Shift to using MongoDB as a data staging area was not the only change needed to support the requirement of incremental import of data. For the ability to decide which records were changed during the extraction part of the process, a few extra metadata fields were added to every record saved in the staging area.The schema of the record with metadata fields is reflected in **Figure 6.5**. For a clear way of adhering to this schema, an ODM library, like mongongine[13], could be used. Unfortunately, validation of each record slows down the saving and updating process significantly[14]. Therefore the presence of metadata fields and their comparing was only enforced on the level of app logic, not on the database level.



MONGO DOCUMENT	
content	Object
code	String
hash	String
to_import	Bool

Figure 6.5. A model of the mongo record with metadata fields

Every time a record is being saved in the MongoDB, first the hash of the JSON transformed into a Python dictionary is computed and compared to the hash of stored record sharing the same code. This logic implies that every record can be uniquely identified by either combining several fields, and creating a key or by using natural key of the record. If hashes do not match, we have a record with changed fields and want to replace the original with the *to_import* flag set to **True**. This process is depicted in **Figure 6.6**. This approach allowed us to identify the changed records needed to be imported. In order to speed up the process, a unique index on the *code* field is created when a collection is created.

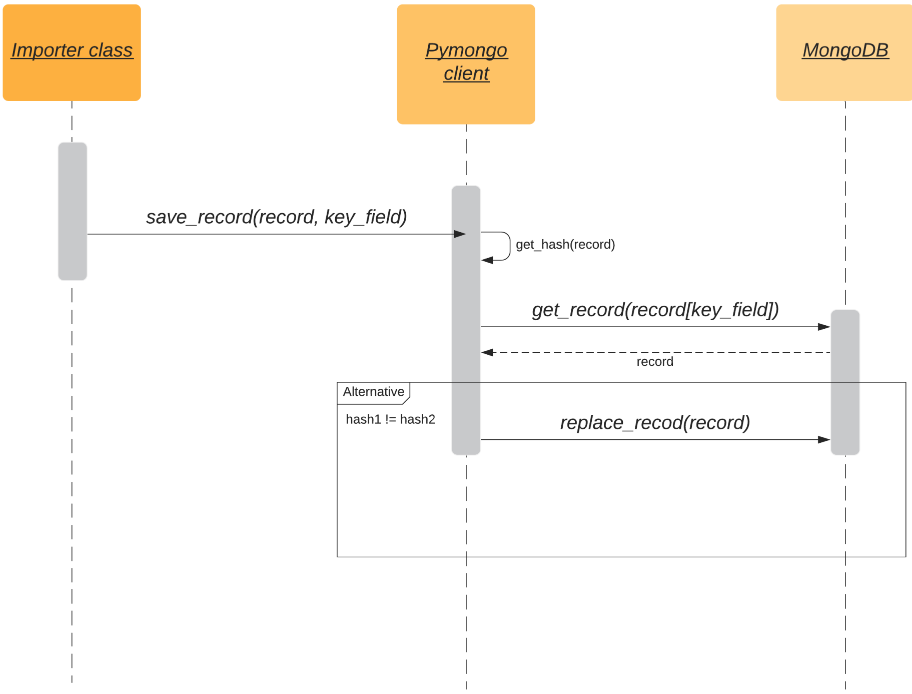


Figure 6.6. The process of saving a record to the staging area

6.2.2 Data governance

The full automation of ETL processes requires the developer team to find a new way of tracking changes in the data over time. Since the ETL process becomes something that will run on a recurring basis, rather than a one time process, logging the changes of data becomes an insufficient solution for keeping record of changed data. Three ways of handling this requirement were identified:

- Database table - every time a record is updated or created, a new row containing information about the change is created.
- Logging - information about the change is logged into a file and later aggregated using custom logic into a CSV report.
- Data lineage framework integration - services like Apache Atlas are specifically designed to allow complex data governance. Every task would define inputs and outputs and propagate information about the changes to the service using REST API.

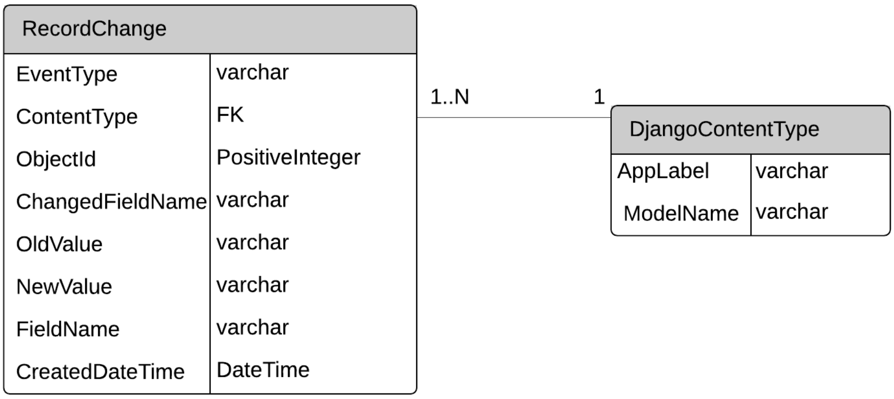
It was decided to go through with the custom database table solution because it allows us to:

- integrate the view over the data into the Django admin interface, shown **Figure 6.8** to allow the filtering of the records,
- comfortably share the insights about the changes using web interface as opposed to sharing custom CSV reports stored on disk,

- track changes outside of the Airflow environment using the standard approach: the Django ORM instead of integrating a full-blown framework that would require the developer team to learn new technology.

Record changes

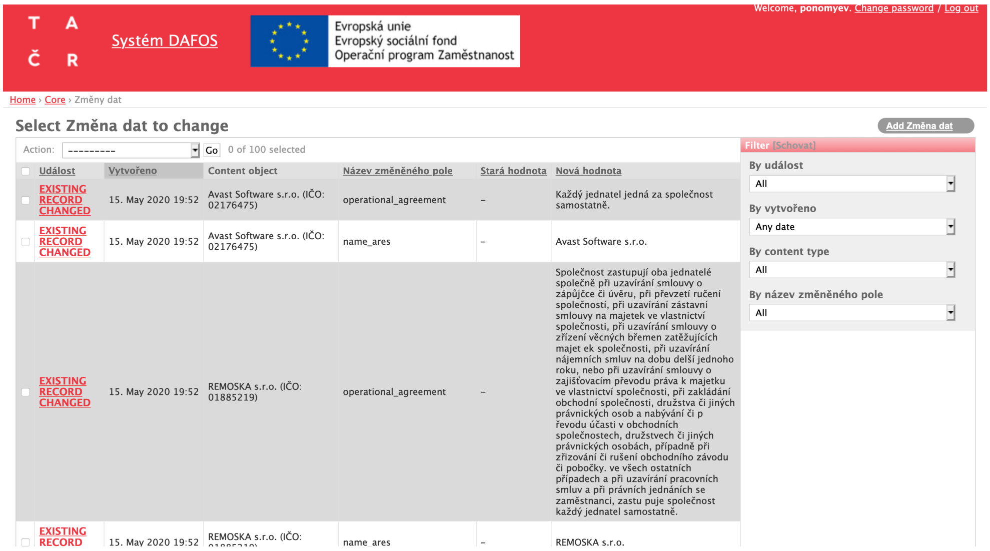
For purposes of tracking changes of the records over time, a custom *RecordChange* model was created, its structure is described in **Figure 6.7**. The main purpose of this model is to differentiate between created and updated records. In order to be able to track changes to any records, the Django content types framework [15] was utilized. It represents and stores information about the models installed in the Django project. This framework, along with the generic foreign keys, allowed to create records referring to any model present in the project.



RecordChange	
EventType	varchar
ContentType	FK
ObjectId	PositiveInteger
ChangedFieldName	varchar
OldValue	varchar
NewValue	varchar
FieldName	varchar
CreatedDateTime	DateTime

DjangoContentType	
AppLabel	varchar
ModelName	varchar

Figure 6.7. The record change relation diagram



Welcome, **ponomyev**. Change password / Log out

Home › Core › Změny dat

Select Změna dat to change

Add Změna dat

Action: ---------- Go 0 of 100 selected

□	Událost	Vytvořeno	Content object	Název změněného pole	Stará hodnota	Nová hodnota
□	EXISTING RECORD CHANGED	15. May 2020 19:52	Avast Software s.r.o. (IČO: 02176475)	operational_agreement	–	Každý jednatel jedná za společnost samostatně.
□	EXISTING RECORD CHANGED	15. May 2020 19:52	Avast Software s.r.o. (IČO: 02176475)	name_ares	–	Avast Software s.r.o.
□	EXISTING RECORD CHANGED	15. May 2020 19:52	REMOSKA s.r.o. (IČO: 01885219)	operational_agreement	–	Společnost zastupují oba jednatelé společně při uzavírání smlouvy o zápůjčce či úvěru, při převzetí ručení společností, při uzavírání zástavní smlouvy na majetek ve vlastnictví společnosti, při uzavírání smlouvy o zřízení věcných břemen zatěžujících majet ek společnosti, při uzavírání nájemních smluv na dobu delší jednoho roku, nebo při uzavírání smlouvy o zajišťovacím převodu práva k majetku ve vlastnictví společnosti, při zakládání obchodní společnosti, družstva či jiných právnických osob a nabývání či p řevodu účasti v obchodních společnostech, družstvech či jiných právnických osobách, případně při zřizování či rušení obchodního závodu či pobočky. ve všech ostatních případech a při uzavírání pracovních smluv a při právních jednáních se zaměstnanci, zastu puje společnost každý jednatel samostatně.
□	EXISTING RECORD	15. May 2020 19:52	REMOSKA s.r.o. (IČO:	name_ares	–	REMOSKA s.r.o.

Filter [Schovat]

By událost
All

By vytvořeno
Any date

By content type
All

By název změněného pole
All

Figure 6.8. View of RecordChange admin interface

6.2.3 Common loading interface

A fundamental abstraction was introduced to allow a unified approach to the loading phase of ETL: ImportedModel. It is an abstract Django model, shown in **Figure 6.9** that is that is expected to be subclassed by every model that will require a view of changes of it over time. The only method present in the model is responsible for two crucial tasks: performing a check on “data_hash” field and ultimately saving precious time by not performing a write operation in the database if hashes of records are equal and the creation of *RecordChange* records, discussed in the previous section, if the specific record has never been saved in the relational database or it has been modified.

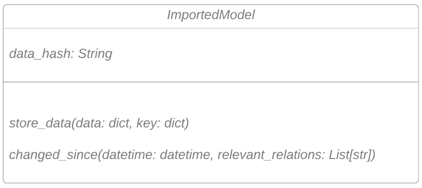

ImportedModel
data_hash: String
store_data(data: dict, key: dict)
changed_since(datetime: datetime, relevant_relations: List[str])

Figure 6.9. The ImportedModel interface

6.2.4 Automation

Apache Airflow is equipped with everything needed for implementing the ETL process. Therefore, great emphasis was placed on using as many Airflow features as possible to make the project more maintainable and present the ETL process with all of its dependencies to the developer as clearly as possible.

Airflow

To follow the requirement of integrating legacy scripts handling the process into a new system, one key operator, with a simple interface described in **Code snipped 6.10**, was developed:

```
class ImporterOperator(PythonOperator):

    @apply_defaults
    def __init__(
            self,
            importer_cls,
            *args,
            **kwargs,
        ):
        self.importer = importer_cls(very_verbose=True,)
        super().__init__(self.importer.run,)

    def execute(self, context):
        super().execute(context,)
```

Code snippet 6.10. Importer operator interface

This allows us to pass a python class to a native Python Operator and execute the *run* method. Note that this approach does not entirely follow the principle that every operator should be atomic, because we are executing, transforming and loading logic in one step but it was necessary for easier adoption.

Secrets

Historically, the data needed to perform the downloading process, namely tokens for access to the RVVI API, were stored in the specific location on the disk, thus requiring the developer to know the location and document this part of the process. Airflow provides standard storage for secrets that can be shared across many processes using Airflow variables. Secrets are persisted in the Airflow metadata database and stored encrypted, so no one can see the contents without having the key for decryption.

Instead of hardcoded connections in the code, Airflow connections were used to keep track of different kinds of source systems such as REST services defined by URL, or database connections. Airflow connections provide a general interface for storing all metadata associated with connections, e.g. schemes, authentication credentials. This data can be queried anywhere in the code, and most importantly, it is decoupled from the code itself. Airflow also provides a way to limit the number of sessions using the specified connection: *Airflow pools*. Pools limit the number of task instances that are using the connection. This can be helpful when tasks are run in parallel, and source systems define some kind of throttling.

Django

Apache Airflow is a standalone app, and the Django context, namely app and model registry, is not provided by default. In order to be able to import modules from the Django app itself, the manual way of initialization of the Django context had to be developed (described in **Code snippet 6.11, 6.12 and 6.13**).

Challenges:

- sys.path - in order to import modules from the Django app, top-level root package needs to be present in the sys.path global variable: a list of strings that specifies the search path for modules [16].
- Django context - Django settings and app registry need to be initialized to access Django ORM and values from the *settings* module.

A simple script was created to fulfill the second requirement: initializing the Django context.

```
import os.environ as env
import sys
# module path to settings
env["DJANGO_SETTINGS_MODULE"] = "dafos.settings"

import django

django.setup()
```

Code snippet 6.11. Setting up the Django context

This script is not sufficient on its own because to import it successfully, its path has to be known by the python interpreter. It would be possible to simply use logic defined in **Code snippet 6.12** every time this module is to be imported, though a different approach, described in **Code snippet 6.13** was taken to follow the DRY principle. An environmental variable *PYTHONPATH* along with an installation dependent paths get propagated to sys.path on interpreter startup [16]. This variable was used to allow imports of modules from the Django app in standalone scripts by simply adding a path to the top-level package to *PYTHONPATH* in a shell profile file (.bash_profile, .zsh_profile).

```
import sys
sys.path.append("/path/to/dafos-module")
```

Code snippet 6.12. Expanding Python path using sys module

```
export PYTHONPATH="${HOME}/dafos"
```

Code snippet 6.13. Expanding Python path using environmental variable

Solr incremental updates

In the past, the serialization of data into Solr was handled by iterating over every record in the database and serializing it one by one. This is no longer acceptable because this process would have to be run frequently and take a lot of time. For the reason of performing incremental updates of the data on a regular basis, a new way of discovering records for serialization needed to be found.

This task is not trivial because schemas of Solr documents are highly nested, and changes to the related records and their fields need to be tracked. Two ways of handling this requirement are possible:

- Individual selection - an individual selection of the records to be updated in the Solr collection based on the date of the last import,
- Continuous integration - the serialization of record on every *save* or *update* event, for example using **Django signals**.

Manual selection using a predefined SQL query was identified as a better fit for the task because it eliminates redundant write operations in the Solr that would arise from the solution involving pub/sub pattern (even though it brings some inconsistency and harder maintainability of the code associated with the need to alter the queries with changes of the database schema). This approach requires a SQL query to be defined for every Solr collection. There are only three collections present in Solr, and their number is not expected to grow drastically.

To identify newly changed records, RecordChange table, discussed in **Chapter 6.2.2**, has to be queried for newly changed records and later synced to Solr. Since Solr documents are denormalized from the relational database schema and composed of many entities that belong together, like organization and result, it is not sufficient to only query one entity; we need to identify changes in related records as well.

A new model, shown in **Figure 6.14** was created to decouple changes in records in the relational database and status of serialization of these records in Solr. SolrSyncStatus is a class providing an interface allowing to store information on what records have been stored in Solr. The content type framework, combined with generic keys, discussed before, was utilized to create relations between arbitrary models and SolrSyncStatus. It was possible to provide fields “is_synced” and “last_synced” as attributes of specific models, but it was decided to use a 1:1 relation to allow idempotency of Solr serialization step in the ETL.

The serialization process is composed of these steps: query newly changed records using the “RecordChange” model along with changed related entities, and sync them to the Solr while creating SolrSyncRecords to keep track of which records have been already indexed.

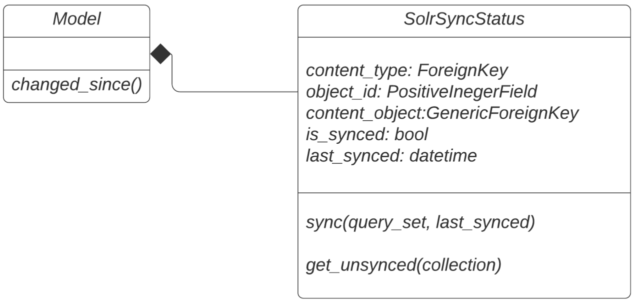


Figure 6.14. The SolrSyncStatus class diagram

6.2.5 Summary

The use of Airflow proved to be very beneficial for defining ETL. It was possible to clearly present the developer with a rather complicated process of ISVAV data set integration. The resulting DAG is presented in **Figure 6.15**.

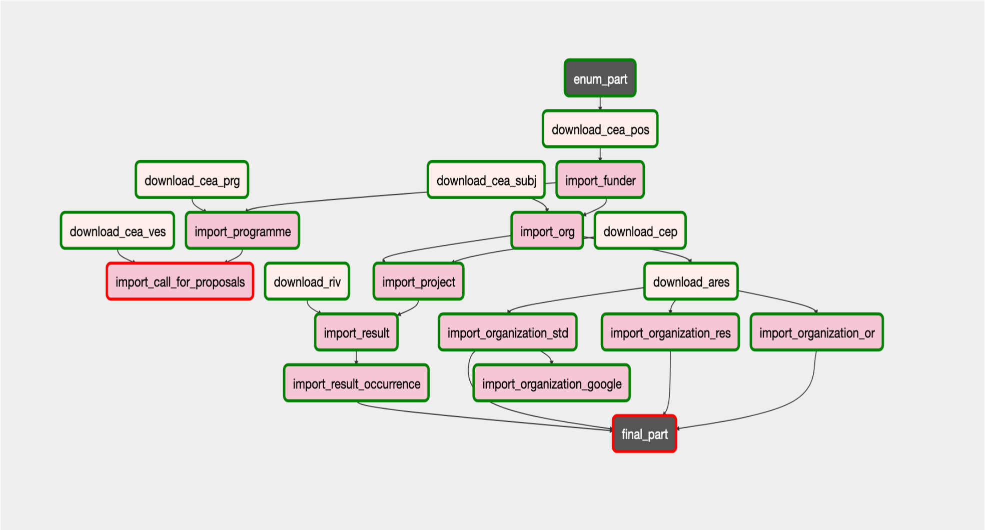


Figure 6.15. A view of dependencies and state of individual tasks in resulting DAG

6.3 Testing

The implemented functionality was tested using the unit and integration tests. Only a few changes were made to the parsing logic, therefore, only a few unit tests covered these changes.

Integration tests played a crucial role in asserting error-free interactions with the new service MongoDB.

6.3.1 Integration tests

Integration tests test whether the system as a whole operates as expected [18]. Several types of interactions could be tested using this approach: communication between classes, modules, or services. For this reason, the definition of an integration test in this project is as follows:

- it verifies how the system works in integration with external dependencies - database, data staging area, or services providing data,

- it tests a particular data pipeline and its parts,
- it tests the presence of metadata needed for the DAG to function properly.

Approach

- Do not mock dependencies over which you have full control (internal services). Using mocks to mimic services can get very tedious: rewriting substantial logic that is not in control of a developer.
- Use "production", original database version, and always be as close to the production environment as possible. This requirement is sometimes hard to follow if no virtualization techniques are used e.g. Docker and Kubernetes.
- Mock dependencies over which we don't have control. No control dependency means you can't remove side effects after interaction with this dependency (external API).

It was decided not to mock or stub the internal dependencies, so a way to provide them in tests needed to be developed. There are no CI or CD principles put in place in the project and the project itself is not containerized.

One possible approach could include integrating a virtualization service such as Kubernetes, which could be used to simulate the production environment with all services available. This environment could be created on the test session creation and deleted on the session finish. Even though this approach could pave the way for containerization of the whole project for purposes of development, it was decided not to go through with it because it would introduce a new technology to learn and is far out of the scope of this thesis.

The final solution performs a similar technique of automated initialization of internal services but without the use of virtualization services. The Pytest framework provides a perfect method for initialization and destruction of dependencies for tests, i.e. fixtures.

MongoDB

The MongoDB dependency in tests was handled by spinning up a new system process running MongoDB on a different port and different database directory than the real production database. The same method as described in the previous chapter was used to provide the fixture in tests. A fixture with *session* scope was created for initializing MongoDB and another fixture with a "function" scope, inherited from it, was created to provide a database client and drop databases between each test run to ensure that MongoDB testing instance is initialized only once when required.

6.3.2 Test suite

The functionality related to interacting with MongoDB was tested in *test_mongo* test suite. This suite is centered around testing of the interface for saving documents into the staging area: saving duplicates in one batch of saved records or asserting that specific flags are changed when contents of document change.

The *test_record_change* test suite is composed of integration tests that examine the functionality related to the creation of *RecordChange* records when a new entity is saved or modified and performing queries on newly revised records and related records.

The *test_store_data* test suite examined the logic of saving of raw data into the relational database and the creation of the *RecordChange* records.

Test suite	Test case
test_mongo	test_fields_added_empty_db
	test_data_change test_duplicate_in_batch test_extra_fields
test_record_change	test_basic_usage
	test_data_types test_changed_since_basic test_changed_since_relevant_relations_basic test_changed_since_relevant_relations_advanced
test_store_data	test_basic
	test_idempotency

Table 6.16. The test suite for newly implemented functionality

Chapter 7
Integrating the PATSTAT data source

Patents represent one of the most valuable types of a result of applied research. There are several organizations all over the world, notably WIPO , EPO and USPTO are overseeing the process of application and validation of patents. The process is somewhat standardized across the main continents; however there is no such thing as a worldwide patent or a central entity providing the data on patents.

7.1 Analysis

The RVVI API interface provides data on patents that were sponsored by funds from the Czech Republic, thus, they were already part of the results collection. This set of patents will be expanded by patents that were developed by Czech scientists or the ones that are active in the Czech Republic.

7.1.1 Business requirements

The main point of interest are patent publications and applications from 2007 onward, with the designated or contracting state being the Czech Republic. For better decision making, these attributes of publication are the most relevant:

- Publication code
- Title
- Classification
- Citations
- Application date
- Publication date
- Validity of patent
- Designated contracting state
- Applicant / proprietor
- Applicant / proprietor (country)
- Inventors
- Inventor (country)

Common attributes should be discovered among the current entities present in the warehouse.

7.1.2 Data sources

Potential sources of data on patents are "Open Data" service from Czech Industrial Office [1] and services provided by the European Patent Office. European Patent Office provides several methods of retrieving patents data in bulk suitable for our use case:

Product name	Price	Format
Open Patent Services	free	REST, JSON
DOCDB Bulk data set	9100 €	XML
PATSTAT	975 €/year	SQL
PATSTAT online	975 €/year	CSV

Table 7.1. Data formats and price comparison

The biggest challenge was to choose the right data source for the task with minimum costs. It was also possible to utilize more than one source. The open data service of the Czech Industrial Office was not sufficient, because it does not explicitly declare whether the patent is valid nor does not provide information on patents applied through other patent offices. It does, on the other hand, provide weekly incremental updates of data. This could be very beneficial for the process of incremental updates of data. Looking at the price of the services offered by , Open Patent Services looks like a promising solution. The fact that it is the only an entirely free solution provided combined with the standard HTTP interface to query the data and the ability to transform it in any desirable way resulted in choosing OPS as the primary source of data. It would be easier to use the SQL database provided by EPO and maintain it apart from the central relational database used in the warehouse, but the final solution provides us with many benefits, not limited to:

- the ability to have access to most up to date data provided by OPS,
- the ability to transform and clean the raw data into desirable structures,
- the ability to combine data already present in the warehouse with the newly integrated data set easily, notably RIV results and business subjects.

Even though the final solution leveraging the use of HTTP interface over the use of static SQL tables is more complicated and time consuming, it offers many benefits that will be crucial in the long run. Unfortunately, OPS does not provide a reliable way to explicitly select a particular subset of patents, only the patents with contracting or designated state being the Czech Republic. Despite this, it is possible to query individual patents based on their codes. The new challenge was: how to get only codes for patents supporting our use case.

To be able to obtain only the codes of patents in an acceptable format and manner, one manual step was needed in the data pipeline. It is possible to obtain a free subscription for PATSTAT online database for three months. The final subset of patents available for querying is limited to only the ones that were published through EPO, but this fact does not restrict us in any way. The only restricting outcome of this approach is not being able to automate the process of producing a report containing said patents because the interface of PATSTAT online is specifically designed to allow only human interaction using a limited subset of SQL directives.

[1] https://isdv.upv.cz/webapp/webapp.opendata.tm

7.2 Implementation

7.2.1 Domain model

The patent data set is covered by the content of the business needs described in **Chapter 7.1.1**. The resulting domain model is shown in **Figure 7.2**.

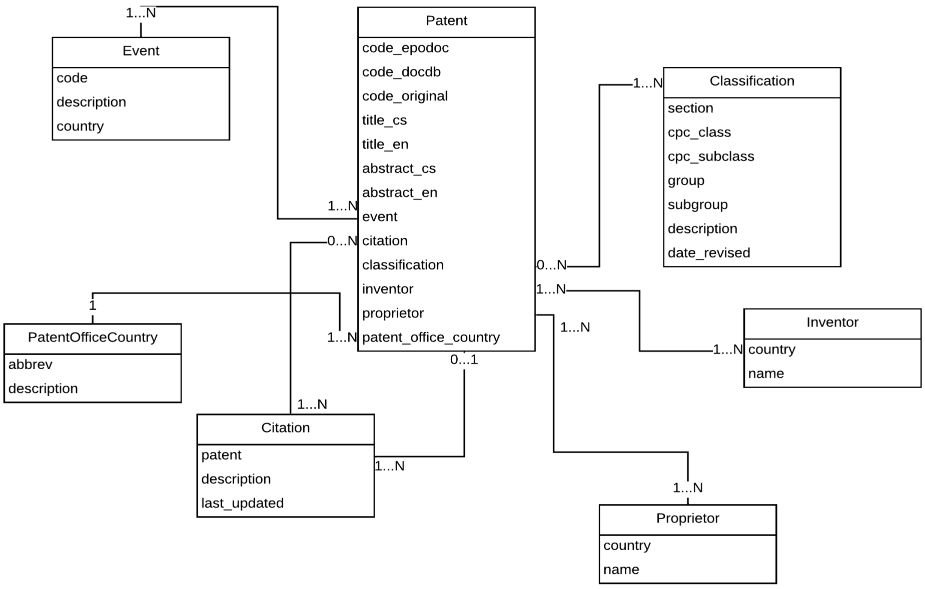


Figure 7.2. Patent data set domain model

7.2.2 Data pipeline

The Open Data Service from EPO was chosen as a primary source of up-to-date data. Unfortunately, the API interface does not support complex querying of the data, which makes the task of finding only a subset of patents that are valid in the Czech Republic unachievable using only this interface. In order to obtain codes of publications adhering to the requirement, other sources of data are needed.

The PATSTAT data set represents a relational database accessible from the web interface or through buying the data set on a hard disk containing SQL scripts that can build the database. These services provide a more significant scope of the data than is needed and cost more than an acceptable price for such an overblown solution.

A semi-automatic solution was chosen for the patents' data pipeline. Only one manual step is required for the process of downloading the patents data set in order for the whole solution to be free of charge. PATSTAT online interface provides a way to create a free account that is valid for three months. This allows us to create SQL reports that can only contain patents that were processed by the EPO. The step of exporting a report of patent applications with fees paid for the Czech Republic will only happen once every 4 months since data on new patents get added at the same rate to the PATSTAT.

The final flow of data in the pipeline is shown in **Figure 7.3**. The data flow stages are very similar to the ones discussed in **Chapter 6** in order to keep a unified approach for ETL processes across the whole warehouse.

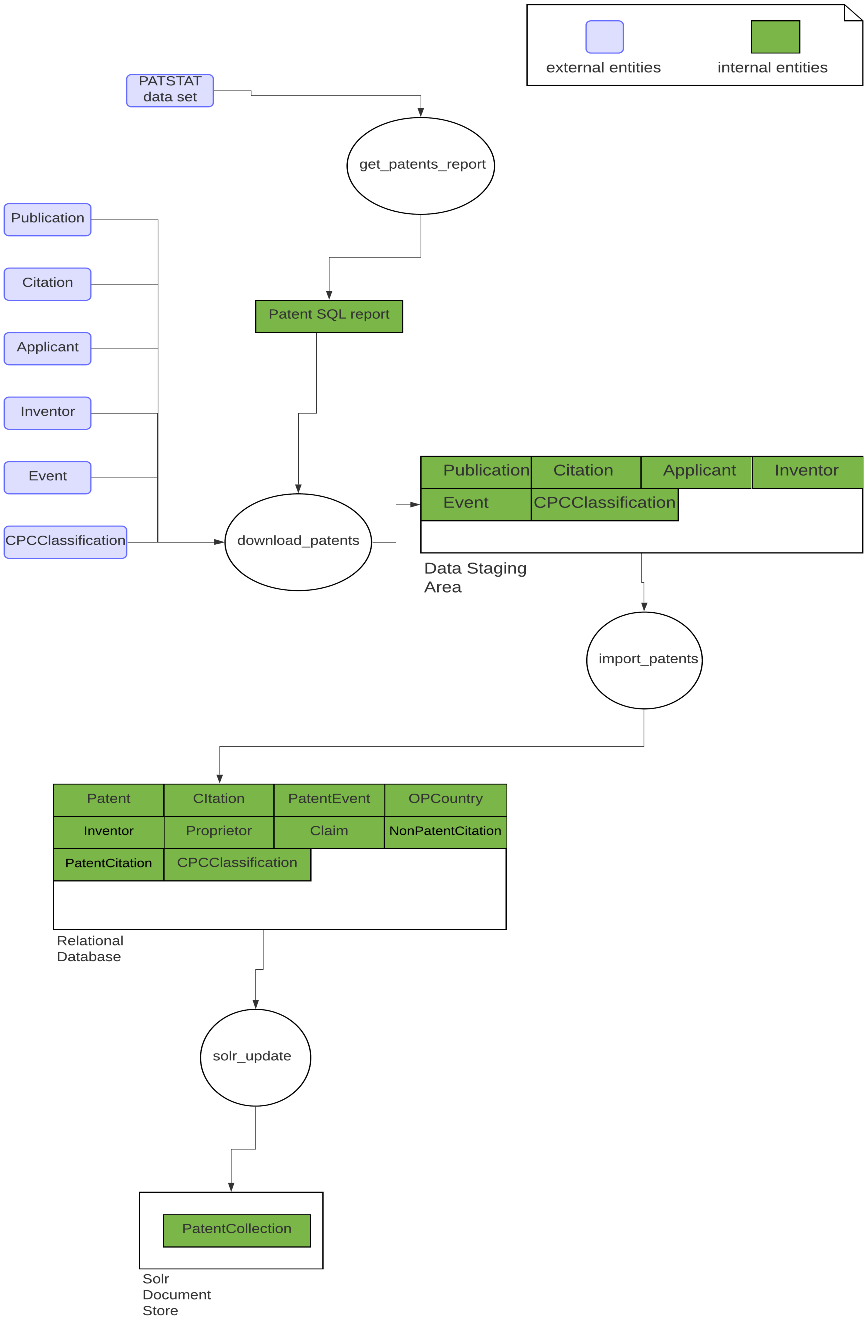


Figure 7.3. Patents data pipeline

7.2.3 Extract

In order to provide a unified approach to the extract stage of the ETL process, the data staging area was used in the same way as in **Chapter 6**. Data has to be loaded into MongoDB first before it was be transformed and loaded into the operational database. The schema of the documents stored in the staging area is identical to **Figure 6.5** to provide functionality for finding recently changed entities.

OPS provides many endpoints for querying information on specific subjects: classification, events, etc. Multiple endpoints were utilized for finding complete information on patents. Each set of records was assigned an individual MongoDB collection for a better decomposition of the loading phase. Each collection shared one common attribute: code of the patent in epodoc format to allow further aggregation.

Unfortunately, the OPS limits the number of concurrent requests to the service, so parallelization of the extraction phase was out of the question.

7.2.4 Transform

The structure of data provided by source systems is not under the control of a developer, thus, establishing a common interface for this stage of the pipeline is not simple, if not impossible.

There were attempts to do so historically. A hierarchy of importing classes was developed for pipelines that use ISVAV as a source system. These classes were distinguished by formats of the data (JSON, XML), types of source and destination storage (database tables, files), and the types of entities they are handling. This hierarchy poses a typical problem of taking a highly object-oriented approach of writing code: we want a banana but must build the jungle and gorillas first. This approach takes a heavy toll on the ability to unit test these classes. Mocking and stubbing unrelated functionality takes a lot of time and is hard to maintain, which results in most of the importing classes not being tested by unit tests. These drawbacks were taken into account when implementing the transforming interface for the patent data set.

Since the schema of source documents provided by OPS is well defined and shared across different patent offices in Europe (and different services serving the data by OPS itself), we can supply a set of classes that transform standard python dictionaries without the need to differentiate between different types of source storage.

This was achieved by defining a set of DTO classes,defined in **Figrue 7.5** handling only the transformation of data. To ease the problem of developing these classes, a new way of defining classes added in Python 3.7 was used: `@dataclass`[17]. This allows for easier development of DTO classes by removing the need to define `__init__` and `__repr__` methods as they are generated automatically (thus, creating code which is easier to read). A simple example of dataclass used for defining DTO is described in **Code snippet 7.4**:

```
@dataclass
class PatentDto(AbstractDto):
    title_cs: str = None
    title_en: str = None
    abstract_cs: str = None
    abstract_en: str = None

    codes: PatentCodesDto = field(
        init=False)
    inventors: List[InventorDto] = field(
        default_factory=list)
    applicants: List[InventorDto] = field(
        default_factory=list)
    citations: List[CitationDto] = field(
        default_factory=list)
    classification: List[ClassificationDto] = field(
        default_factory=list)

    @staticmethod
    def parse(data: dict) -> PatentDto:
        ...

    def save(self) -> Patent:
        ....
```

Code snippet 7.4. Example of dataclass

Unfortunately, the dataclass interface limits the use of *@property* descriptors, therefore, in order to provide a standard methods of defining getters and setters, classic classes were used.

This approach allows us to easily unit-test business logic as opposed to previous interface, where classes had to be instantiated with all of their dependencies met.This requires a lot of mocking and extra effort to test simple logic.

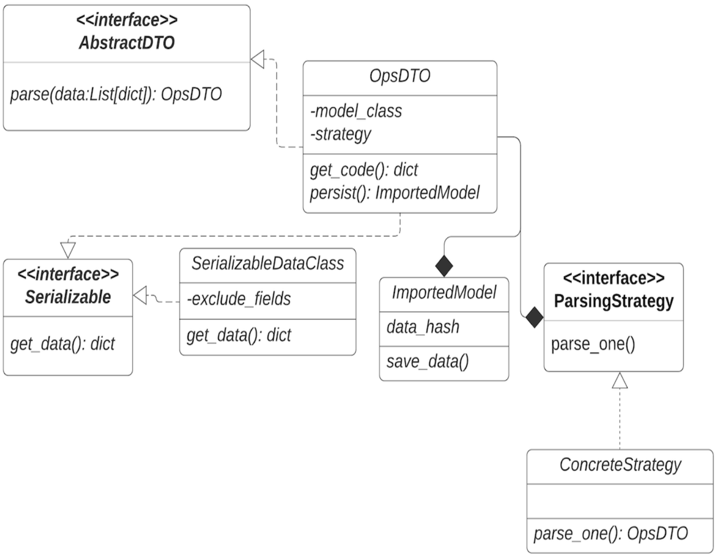


Figure 7.5. The class diagram of the extraction interface

7.2.5 Load

The loading stage of the pipeline is identical to the interface used in **Chapter 6** in order to guarantee the creation of *RecordChange* records later used in monitoring the progress of pipelines and providing a way to find changed records that have to be indexed in Solr.

The loading phase consisted of two parts: loading of records into the relational database and later loading them into Solr. In order to speed up the process, loading was run in parallel with the ability to regulate the number of workers handling the loading phase. Even though Airflow provides a way to create tasks dynamically in a DAG, this functionality was insufficient. The execution plan of DAG has to be known at compile-time, i.e., the time when the DAG code gets parsed by the Airflow scheduler. This limits the parallelization because there is no way to determine the number of new documents to be loaded as this variable will be known later in the process when records have been extracted from source systems. In order to allow the parallelization of loading tasks, there is a need to have information on the total number of jobs running and current iteration. Then it is possible to query MongoDB limiting the number of retrieved records and skipping through them by batch size.

7.2.6 Automation

Significant focus was placed on unifying the individual stages of the ETL process across all pipelines, but some trade-offs had to be made. The automation of the patent data set ETL processes was achieved through defining a set of Airflow operators handling communication across the source and destination services. In order to provide a unified approach for ETL, considerable focus was placed on using the Airflow interface as much as possible. This means, namely moving logic out of Django commands into Airflow operators and storing ETL metadata in the database instead of on disk to allow easy integration with Airflow. The following section will give more details on how different parts of Airflow framework were put to use:

Operators

A set of Airflow Operators, pictured in **Figure 7.6**, was created. These operators are where the logic is stored. They are meant to be reusable pieces of code but some serving a very specific use case

- EPOSimpleHttpOperator - ensures that an access token needed for requests to OPS is present and up to date, if not it will be refreshed and populated to the headers of the request and saved to Airflow variable to allow its reuse by other operators.
- EPOToMongoOperator - ensures that records downloaded from OPS to staging area conform to the schema defined in **Figure 6.5**.
- MultipleEPOToMongoOperator - allows to query OPS enpoint for multiple entity codes one by one
- MongoToPostgresOpertor - operator for saving Mongo documents to destination tables in relational database using OpsDTO sub-classes.
- DictToMongo - base operator for storing Python dictionaries in MongoDB while adhering to schema defined in **Figure 6.5**.
- MongoPatentToPostgresOperator - specific operator for storing patents from MongoDB in relational database with additional information located in various MongoDB collections.

These operators are meant to be used across all DAGs that download data from OPS.

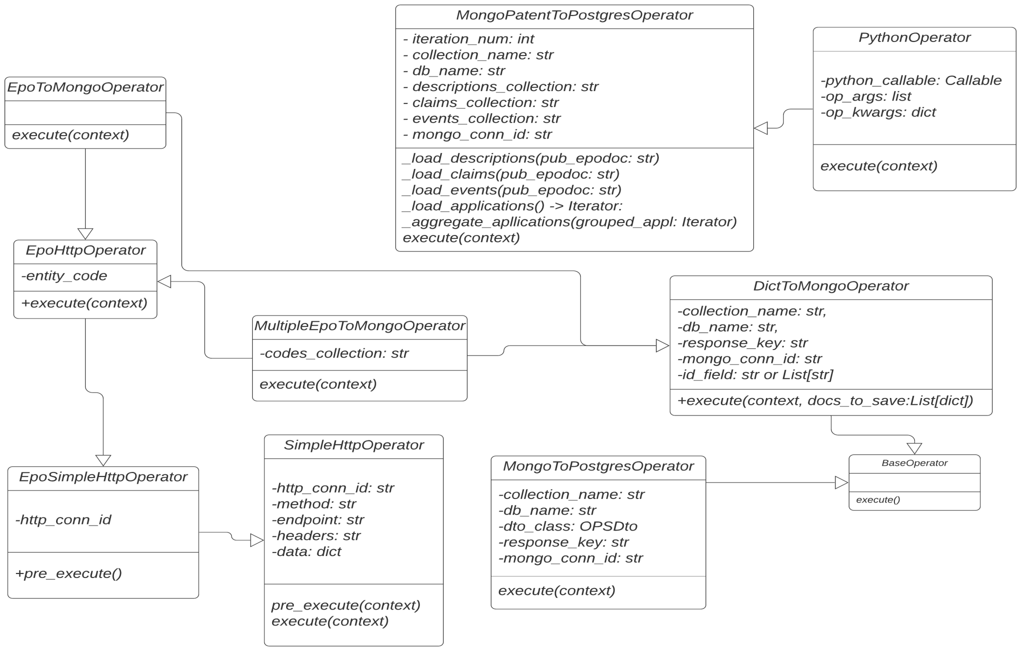
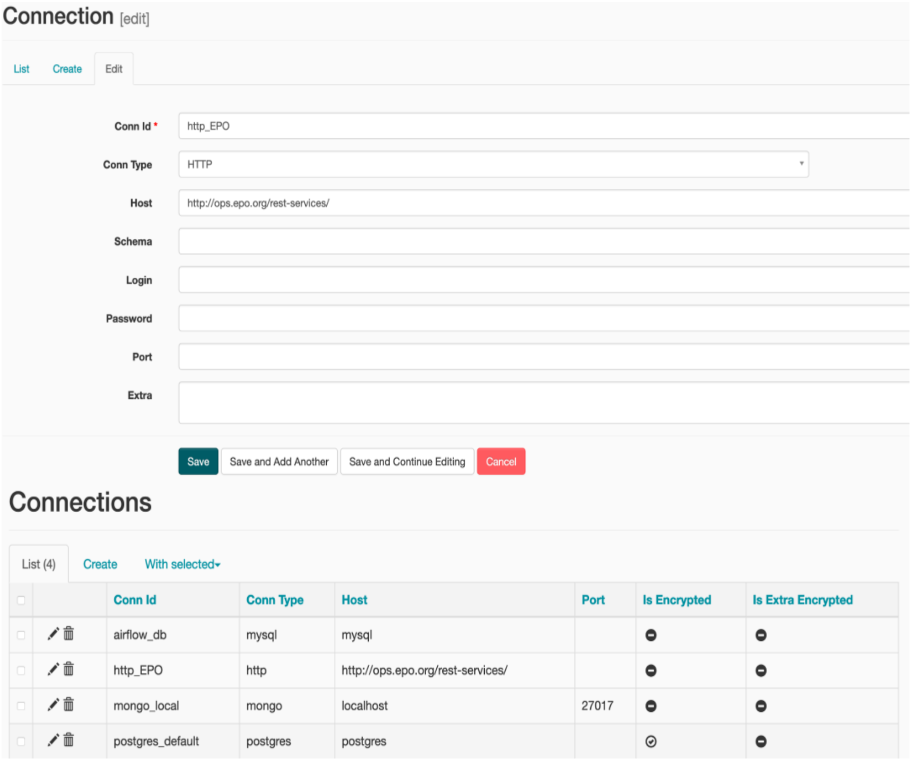


Figure 7.6. A class diagram of extraction interface

Connections

The connections to external systems were defined using the Airflow connection interface. Connections provide a common interface for defining details of the connection and their reuse in DAGs. The details about the connections are accessible from the web admin interface and through code interface. In the case of standard HttpOperators, the developer is only required to specify the name of the connection, and the operator will handle all logic of getting the data about the connection by itself.

To globally change the details of the connection, the admin interface pictured in **Figure 7.7** can be usedthus, enabling anyone with the right permission to alter the details of connections without the need to change code.



Conn Id * http_EPO
Conn Type HTTP
Host http://ops.epo.org/rest-services/
Schema
Login
Password
Port
Extra

		Conn Id	Conn Type	Host	Port	Is Encrypted	Is Extra Encrypted
□		airflow_db	mysql	mysql		⊖	⊖
□		http_EPO	http	http://ops.epo.org/rest-services/		⊖	⊖
□		mongo_local	mongo	localhost	27017	⊖	⊖
□		postgres_default	postgres	postgres		✓	⊖

Figure 7.7. An admin interface of connections in Airflow

Variables

The Airflow variables serve as an interface to the centralized storage of data. Variables are accessible to the processes through the DAO layer shown in **Figure 7.8**. Any data that can be pickled using a python **pickle** module can be stored in the variable.

```
Variable.set(value="foo", key="bar")
Variable.get(key="bar")
```

Code snippet 7.8. Variable DAO interface

These variables were used to store secrets and credentials needed for authorizing requests to OPS. The data stored in the variables, are encrypted and if the key of variable contains keywords such as *password* or *secret*, it won't be available to view through the admin interface. This provides some sort of security as only people that have access to the encryption key can get hold of the stored value. Encryption is handled entirely by the Airflow framework and is therefore transparent to the developer.

7.3 Testing

Testing pipelines in data applications is very challenging, and results have low value because of the inability to simulate conditions in the production environment. Therefore, integration tests served as “sane checks” and are not meant to cover all possible scenarios since format of the source data can change over time.

7.3.1 Unit tests

The ETL interface for patent data set was specifically designed to allow easier testing through the means of decoupling the transforming and loading stage of the process using the DTO layer. The unit-tests, therefore, tested only the parsing and saving logic of the DTO layer.

Parametrized tests were established for the purpose of testing data cleansing. Parametrized tests allow for testing multiple cases in one test run and clearer definition of inputs and outputs.

```
country_in_out = [
    (" [CZ] ", "CZ"),
    (" [CZ]", "CZ"),
    ("[CZ] ", "CZ"),
    ("[CZ]", "CZ"),
    ("* [CZ]* ", "CZ"),
    ("[CZ]* ", "CZ"),
    ("- [CZ] -", "CZ")
]


@pytest.mark.parametrize("raw, output", country_in_out)
def test_country_dto_code_setter(raw, output):
    country = CountryDto()
    country.code = raw

    assert country.code == output
```

Code snippet 7.9. Parametrized test example

7.3.1 Integration tests

The integration tests were mainly developed to test custom operators and perform smoke tests on the DAGs. The testing environment introduced in **Chapter 6.3** was reused and another fixture was provided for the initialization of Airflow metadata DB.

Airflow metadata database

For initiation of Airflow metadata database, a different database engine, sqlite, was used than the one used in the production . Sqlite is a database stored entirely on disk using a file. It provided an easy way to create and destroy the test databases to guarantee a fresh state of the database for each test session. This is somehow contrary to the points discussed in **Chapter 6.3**. However, it is an acceptable approach because the only limitation it brings is the inability to run processes in parallel.

To only initialize the airflow database for the tests requiring it, pytest fixtures were used. Instead of blindly initializing it for all tests in pytest_sessionstart, a fixture, described in **Code snippet 7.10** with **session** scope was created. The session scope of the fixture ensures that the fixture will only be created once per test session (if used in tests) and cached for use in all tests. After the session has finished, the database will be destroyed along with all of the other dependencies. This was achieved by yielding the fixture, code after yield statement will be executed, when fixture goes out of the scope.

```
@pytest.fixture(scope='session')
def airflow_test_db():
    _setup_airflow()
    yield
    _kill_airflow()
```

Code snippet 7.10. Initialization/destruction of airflow metadata DB

7.3.3 Test suite

The complete test suite, shown in **Table 7.11**, was composed of integration and unit tests for the implemented functionality.

Integration tests defined in the test_airflow_operators suite were testing logic associated with generation of authentication tokens on every request to OPS, retrieval of data from mocked sources and later saving them in destination databases.

The tests in test_dto suite were testing logic responsible for parsing of raw JSON data and asserting that saved objects correspond to the parsed data.

Test suite	Test case
test_dto	test_country_dto_code_setter
	test_country_save test_inventor_save_multiple test_citation_parse test_citation_save test_non_citation_parse test_non_patent_citation_save test_classification_parse test_classification_persist test_patent_parse test_patent_save test_event_parse test_event_save
test_airflow_operators	test_EPOSimpleHttpOperator_success_no_previous_token
	test_EPOSimpleHttpOperator_success_previous_empty test_EPOSimpleHttpOperator_success_previous_not_valid test_EPOSimpleHttpOperator_success_previous_not_changed test_EPOSimpleHttpOperator_fail test_get_EPOHttpOperator_single_entity test_get_EPOHttpOperator_post_multiple_entities test_EPOToMongoOperator_no_response_key_nor_id_field test_EPOToMongoOperator_response_key_and_id_field test_get_MongoToPostgres_aggregate_success test_get_MongoToPostgres_aggregate_additional_data_success test_MultipleEPOToMongo_operator

Table 7.11. Test suite related to PATSTAT integration

Chapter 8
Conclusion

The key objectives of this thesis were:

- perform an analysis of available orchestration solutions,
- automate current ETL scripts,
- integrate a new data set into a data warehouse using the chosen solution, providing a way to monitor the whole process.

After performing an analysis of the available industry-standard solutions for orchestrating of scripts in Python, described in **Chapter 5**, and defining principal requirements for the final ETL framework in **Chapter 4**, the Apache Airflow was chosen as the best fit for the job of automating ETL processes in a DAFOS data warehouse. Simple automation of scripts was not enough, and the problem as a whole produced many sub-problems discussed in **Chapters 6.2** and **7**. MongoDB was utilized to provide a more robust solution for the data staging area and to allow querying and an analysis of staged data. All of the implemented functionality was tested using unit tests and integration tests; a basic testing strategy had to be established to allow that. Testing is described in greater detail in **Chapters 6.3** and **7.3**.

All of the requirements were satisfied with one exception: refactoring of the data model. It was decided that this task is out of the primary scope of this thesis because it does not explicitly relate to ETL in general; it just provides a cleaner way to structure the data in the database. The main idea behind this requirement is to allow further decoupling of the apps present in the project. Common entities that are imported from different sources would be saved in separate tables and later joined into one table containing foreign keys to the records of each separated tables.

Data from source systems is integrated periodically without the need for manual scheduling. Even though the dependencies between tasks and processes are still present, all of the individual tasks are idempotent. The resulting solution provides a developer with ETLs that are self-describing, i.e., their dependencies and state are accessible from the web interface. The developer team is not only capable of tracking the state of DAGs but also changes happening in data in real-time. It is possible to query logs of tasks and share insights using URLs to the web interface instead of sharing files containing this information.

8.1 Further refinements

The patent data set accessible from OPS provides data that is not thoroughly sanitized (errors occur during the conversion of character encoding, many unwanted and unpredictable characters are present in data). Therefore, a great deal of effort needs to be put into data cleansing. Collective entities have to be found and later connected on the database level, in order to provide the end-user with a better product covering information on across the whole data warehouse. Even though all key implementation points were covered by the tests, some of them expect data schema that is in control of data providers; if the schema of data in the source systems changes, these changes will not be handled by tests. It would be beneficial to assert data expectations in tests to allow better maintainability of the project.

The old ETLs managing the IS VaVaI integration are still present in the project. However, it is necessary to rewrite them into the related Airflow abstractions, and further divide them into smaller units to have a unified and idempotent ETL approach and leverage Airflow's use across the whole warehouse.

As a developer is no longer in charge of the execution of ETLs, some ill-structured or corrupt data can get integrated into the data warehouse. A solution providing a way to version changes in the relational database would be beneficial to the maintainability of the final solution. The standard approach to this problem [20] could be utilized.

Chapter 9

List of abbreviations used in the document

AWS	Amazon Web Services
DWH	Data warehouse
GCP	Google Cloud Platform
ODM	Object document mapping
PATSTAT	EPO Worldwide Patent Statistical Database
R&D	Research and development
TA CR	Technology agency of the Czech Republic
USPTO	U.S Patent and Trademark Office
WIPO	World Intelectual Property Organization

References

[1] VAISMAN, Alejandro and Esteban ZIMÁNYI. *Data Warehouse Systems* [online]. 1. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014 [cit. 2020-05-12]. DOI: 10.1007/978-3-642-54655-6. ISBN 978-3-642-54654-9. Available at `https://www.springer.com/gp/book/9783642546549`.

[2] IMHOFF, Claudia, Nicholas GALEMMO and Jonathan G. GEIGER. *Mastering Data Warehouse Design: Relational and Dimensional Techniques.* 1. Indianapolis, Indiana: Wiley Publishing, 2003. ISBN 0-471-32421-3.

[3] POUR, Jan, Miloš MARYŠKA and Ota NOVOTNÝ. *Business intelligence v podnikové praxi.* Praha: Professional Publishing, 2012. ISBN 978-80-7431-065-2.

[4] KIMBALL, Ralph and Joe CASERTA. *The Data Warehouse ETL Toolkit: Practical Techniques for Extracting, Cleaning, Conforming, and Delivering Data.* 1. Indianapolis: Wiley Publishing, 2004. ISBN 0-764-57923-1.

[5] KIMBALL, Ralph and Margy ROSS. *The data warehouse toolkit: the definitive guide to dimensional modeling.* 3rd ed. Indianapolis: Wiley, c2013. ISBN 978-1-118-53080-1.

[6] Evaluation report of Proeval project. Available at: `https://www.tacr.cz/o-nas/interni-projekty-ta-cr/projekt-proeval/`

[7] ARIYACHANDRA, Thilini and Hugh J WATSON. *Which data warehouse architecture is most successful?* Business intelligence journal. The Data Warehouse Institute, 2006, 11(1), 4.

[8] Luigi documentation. Available at: `https://luigi.readthedocs.io/en/stable`.

[9] Apache Airflow documentation. Available at: `https://airflow.apache.org/docs/stable`.

[10] Dagster documentation. Available at: `https://docs.dagster.io/`.

[11] Publicly available data from IS VaVaI. Available at: `https://www.rvvi.cz`.

[12] Documentation on parallelized Mongo bulk operations. Available at: `https://docs.mongodb.com/manual/reference/method/db.collection.initializeUnorderedBulkOp/`

[13] Mongoengine - object document mapping library. Available at: `http://mongoengine.org/`

[14] Why ORM shouldn't be your best bet. Available at: `https://medium.com/ameykpatil/why-orm-shouldnt-be-your-best-bet-fffb66314b1b`

[15] Django contenttypes framework. Available at: `https://docs.djangoproject.com/en/3.0/ref/contrib/contenttypes/`

[16] System-specific parameters and functions. Available at: `https://docs.python.org/3/library/sys.html`

[17] Data Class Python enhancement proposal. Available at: `https://www.python.org/dev/peps/pep-0557/`.

[18] ROSSEL, Sander. *Continuous Integration, Delivery, and Deployment.* 1. Birmingham: Packt Publishing, 2017. ISBN 978-1-78728-661-0.

[19] Publicly available data on business entities from Ministry of Finance of the Czech Republic `https://wwwinfo.mfcr.cz/ares/ares_es.html.cz`

[20] PostgreSQL: Continuous Archiving And Point-In-Time Recovery. Available at: `https://www.postgresql.org/docs/12/continuous-archiving.html`

Appendix A
Contents of enclosed CD

The CD enclosed with this thesis contain digital copy of this document and all sources needed to compile it in the *thesis* folder.

The *src* folder contains minimal set of modules and packages needed to test the implemented functionality. It was needed to include many parts of the DAFOS project not covered by this thesis in order to run implemented functionality, notes on the self-written code are present in the *README* file

```
src/ # code sources
  - README.md # notes on installation procedure and package contents
thesis/ # sources of the tex document
  - main.tex
  - main.pdf # this document
```