JOHANNES KEPLER
UNIVERSITY LINZ

Sustainability Data Warehouse - A Solution for Global Warming Data Gathering, Enrichment and Processing

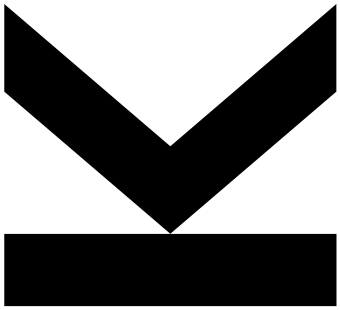
Author
Stefan Haslhofer

Submission
Institute for System Software

Thesis Supervisor
Dipl.-Ing. Dr. Markus Weninger, BSc

External Thesis Supervisor
Dipl.-Ing. Dr. Michael Aichinger

January 2023



Bachelor's Thesis

to confer the academic degree of

Bachelor of Science

in the Bachelor's Program

Informatik

JOHANNES KEPLER UNIVERSITY LINZ
Altenberger Str. 69
4040 Linz, Austria
www.jku.at
DVR 0093696

JOHANNES KEPLER
UNIVERSITY LINZ

Bachelor's Thesis

Global Warming Scoring Solution for Sustainable Investment

Dipl.-Ing. Dr.
Markus Weninger, BSc
Institute for System Software

P +43-732-2468-4361
F +43-732-2468-4345
markus.weninger@jku.at

Student: Stefan Haslhofer
Advisor: Dipl.-Ing. Dr. Markus Weninger, BSc
uni software plus GmbH Supervisor: Dipl.-Ing. Dr. Michael Aichinger
Start date: March 2022

Sustainability and as part of it climate friendly/neutral investing is getting evermore important for private and institutional investors. As such, there is a need for metrics that classify the sustainability of investment instruments based on the behavior of the companies behind them. In a collaborative project, uni software plus GmbH develops a solution to measure a company's global warming potential based on its greenhouse gas emissions and other metrics.

The goal of this bachelor thesis is to model a database and implement services suitable to store provided market data as well as transforming it into an understandable format. An external model will then accept this format and calculates the respective temperature score, which subsequently also has to be stored in the database. This system will be containerized and deployed to the Google Cloud, therefore needing to use resources as efficient as possible in order to minimize costs.

Following tasks must be achieved:

- Design of a cloud-ready micro-service architecture and database.
- Optimization of performance and resource usage/parallelization of different services in order to minimize costs.
- Implementation of a service loading and saving market data as well as providing endpoints to send and receive information to and from the temperature score data model
- Administration of a cloud application
- Ensure scalability in order to support multiuser operation
- (nice to have) small frontend to display the calculated metric

Modalities:
The progress of the project should be discussed at least every two weeks with the company supervisors and at least once per month with the advisor. A time schedule and a milestone plan must be set up within the first 3 weeks and discussed with the advisor and the supervisors. It should be continuously refined and monitored to make sure that the thesis will be completed in time. The final version of the thesis must be submitted not later than 31.09.2022.

Abstract

With climate change posing an increasing threat the need for more sustainability is a pressing topic in nearly all fields of our economy, including the banking sector. For a growing number of investors environmental friendliness is an important decision factor.

Therefore, we introduce the *Tempscore* indicator that informs about a company´s *warming potential*. The warming potential is the approximated amount in degrees Celsius the earth will heat up until a certain year if the entirety of the global economy changes its emission output by the same percentage in relation to its sales volume as the rated company. Our solution is designed for smaller regional financial service providers that do not have the means to implement a more sophisticated *eco-friendliness* rating themselves.

In this thesis we present the technical aspects of a system built to import and process large amounts of market data and environmental data while at the same time staying performant and scalable. To achieve this we use the Java *Spring* framework and in particular the *Spring Batch* library. With Spring Batch we are able to divide our indicator calculation into multiple so-called partitions which act in parallel, thus reducing execution time. Although the mathematics behind the calculation is not part of this thesis, we will still clarify the basic concepts.

Kurzfassung

Die Klimaerwärmung stellt eine immer größer werdende Gefahr für unser aller Wohl dar, weswegen das Bedürfnis nach Nachhaltigkeit in fast allen Wirtschaftsbereichen an Bedeutung gewinnt, darunter auch der Finanzsektor. Für eine steigende Anzahl von Investoren ist die Umweltfreundlichkeit ein wichtiger Entscheidungfaktor.

Deshalb führen wir den *Tempscore* Indikator ein, der Aufschluss über das *Erwärmungspotenzial* einer Firma gibt. Das Erwärmungspotenzial ist der geschätzte Wert in Grad Celcius, um den sich die Erde bis zu einem gewissen Jahr erwärmen würde, wenn die gesamte Weltwirtschaft ihre Emissionsabgabe um die gleichen Prozentpunkte im Verhältnis zum Umsatz wie das bewertete Unternehmen ändert. Unsere Lösung ist für kleinere regionale Finanzdienstleister entwickelt worden, die nicht die richtigen Mittel besitzen um selbst ein ausgeklügeltes System zur Bewertung der Klimafreundlichkeit zu implementieren.

In dieser Arbeit präsentieren wir die technischen Aspekte eines Systems das gebaut worden ist, um eine große Anzahl von Markt- und Umweltdaten zu verarbeiten und dabei noch performant und skalierbar bleibt. Dazu verwenden wir das Java *Spring* Framework und insbesondere die *Spring Batch* Bibliothek. Mit Spring Batch ist es uns möglich, die Berechnung der Indikatoren in mehrere sogenannte Partitionen aufzuteilen, die parallel arbeiten und so die Laufzeit verbessern. Obwohl die Mathematik hinter der Berechnung nicht Teil dieser Arbeit ist, werden wir uns die grundlegenden Konzepte trotzdem anschauen.

Table of Content

Contents

1 Introduction

Sustainability is an important topic in all areas of economics, politics, as well as our daily lives. Global warming is an omnipresent matter in worldwide news; weather anomalies such as extreme heat or heavy storms become more intense and glaciers alongside other bodies of ice decrease rapidly in size [3]. Financial institutions recognize the need for climate friendly investment options for their private and institutional customers but conclusive metrics rating an investment's sustainability are difficult to find and the data situation is sparse. Furthermore, a sophisticated mechanism that is able to assess large quantities of investment options is cost intensive and time intensive and additionally requires software engineering know-how. Such undertakings can only be accomplished by large corporations with dedicated software departments.

Thus, we implemented a *temperature warming scoring solution* (Tempscore) that allows small and mid-sized financial service providers to classify the eco-friendliness of their assets. Our solution enables investors to make purchase decisions based on a score that indicates how much degrees Celsius the earth would warm up until a certain year if the entire world economy increases or decreases its annual emissions by the same percentage as a particular company. The score also takes the change in revenue of the rated company into account. This means that a company that manages to reduce its emissions despite a strong economic growth may be rated better than another company that lowers its emissions even stronger but also experiences a decline in sales volume.

This paper gives an overview over the technical aspects of a system that enables us to handle large amounts of company data and market data that we can utilize to get a meaningful sustainability rating. We are going to have a look into the extraction and the processing of raw data as well as the interaction between distinct components of the system. Additionally, we will briefly inspect the system's performance and the role of parallel execution. Although this thesis only revolves around the data refinement and micro service interactions behind the actual indicator calculation itself, we will still examine the basic mathematical concepts of the necessary computations.

2 Background

This section explains why sustainability is important for the finance sector and the economy in general and why we decided to set foot on this specific topic. We will further illustrate how the results of this project may influence and interact with other products of uni software plus GmbH. Beyond that, we will inspect the basic mathematical concepts that lead us to our Tempscore indicator.

2.1 Sustainable Investment

Sustainability describes a safe co-existence between us humans and nature. A sustainable society minimizes the damage done to the environment and pushes for a resortful handling of natural resources in all aspects of live. Looking at the bigger picture, our economy plays a large role in global warming and climate change. As private individuals or corporate entities we are able

to pressure global markets to become more eco-friendly through consumer habits. Structures provided by our free market allow us to directly support companies with investments of our own capital. With increasing climate awareness around the world the demand for sustainable investments has grown steadily over the past decades which makes it a valuable asset for all industries including the banking sector. However, being able to rate companies by their eco-friendliness is difficult and only a few solutions exist which often still require some form of manual data processing. Hence, we decided to build an application that is able to provide an automated indication for a company's ecological impact based on market data, environment data and company data.

2.2 Integration into Existing Products

Uni software plus GmbH develops multiple products in the financial sector especially for small and middle sized financial institutions. Some of these products already include calculated indicators to classify the investment risk of an instrument based on its attributes. The goal of our Tempscore solution is to not only provide it as a standalone service, but to extend these established systems with our temperature rating.

2.3 Terminology

Instrument Financial instruments are contracts between two parties that can be obtained and traded in exchange for money such as stocks, bonds or even currency swaps.

ISIN (International Securities Identification Number) A twelve-digit globally unique alphanumeric string identifying an instrument traded on a stock exchange.

Market Data All the information, statistics, and metrics that exist for an instrument.

Company Data In this thesis the term company data means every bit of information on the company itself and not just its representative instrument on the stock market. Environment data concerning a certain company, such as produced CO_2 emissions, fall into this category.

Environment Data Information representing the current state of our surroundings in numbers, for example global CO_2 emissions or temperature delta per year.

2.4 Indicator

To understand the application to its full extent we must clarify at least the basic mathematical concepts our computation is built on. Remember, the Tempscore indicator demonstrates the eco-friendliness of a company. It reveals to investors how many degrees Celsius the earth's temperature will approximately increase until a target year (e.g. 2050) if the global economy alters its emission output at the same percentage as the business in question.

Our approach is more sophisticated than simply evaluating a company's sustainability by the absolute number of produced tons of CO_2. We consider the ratio of emissions to revenue (corrected by inflation) over the last two years and calculate the relative change. We assume that the relative change remains consistent in the future. Based on the current yearly company emissions and the relative change (e.g. 4% decrease per year) we can approximate future annual emissions. At last we are able to accumulate the approximated annual emissions up to the target year and apply the result to a climate model representing the temperature increase per gigaton (GT) CO_2. [1]

If the company manages to keep its relative emission change lower than its revenue change it will obtain a more favourable rating. Therefore, a fast growing corporation which inevitably intensifies its ecological footprint can still be classified more environmentally friendly than a smaller business that did not lower its emissions in equal measures.

3 Architecture

This section examines the individual parts of the software system behind the temperature score calculation and how these parts interact with each other. Moreover, we will inspect the route between the application's different services that raw packages of market data must traverse in order to get transformed into meaningful results.

3.1 General Design

Our solution mainly consists of three logically independent processes: a data importer, a calculator and a result exporter, which all make use of multiple components. The following sections explain these main processes and their components. Figure 1 depicts the system in a finer granularity and shows the connections between all parts. The ensuing list briefly outlines all major program components:

1. A `file transfer protocol (FTP)` server stores input data received as files before it is imported into our system. The `FTP` serves as a landing zone and can therefore be regarded as an interface between outside data providers and the Tempscore application.

2. The *Tempscore DB* is a *Postgres DB* used to store imported and transformed data as well as calculation results.

3. *Calculation partitions* are the centerpiece of our calculation process. They receive already imported data and transform them into meaningful indicators. Multiple partitions work in

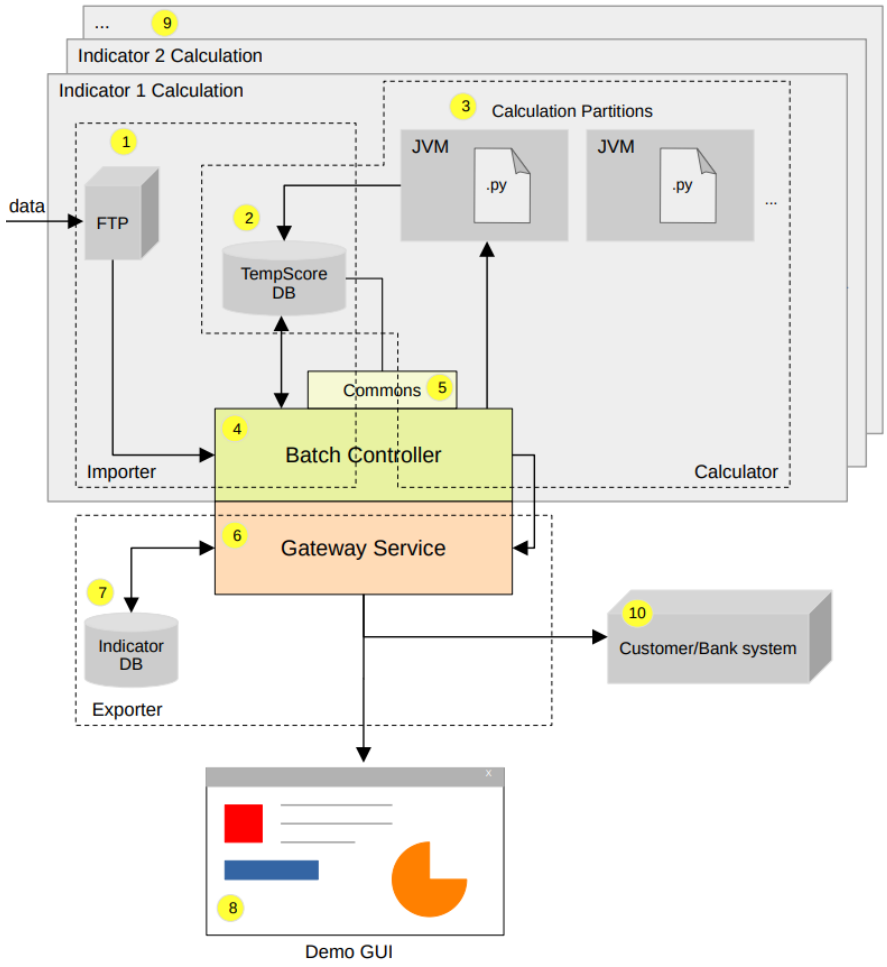


Figure 1: Overview of the Tempscore system. The arrows show possible directions of the data flow between individual components. The dashed lines roughly split the system into three groups of components. Each group is utilized by one of the three main processes described in Section 3.1 (i.e., importer, calculator, exporter).

parallel which divides workload into smaller batches. Each partition is started as a standalone Java process running in its own `JVM`. As mentioned in Section 4.3.3 the calculation partitions call a python script to calculate the indicators.

4. In an effort to centralize management of components in the Tempscore solution we implemented a so-called *batch controller (BC)*. It directs the bulk of the data flow in the importer process and calculator process. Additionally, the BC supplies other services with result data.

5. The *commons* module holds metadata regarded as general knowledge within the Tempscore application, such as database layout, `data transfer object (DTO)` structures, and stored

queries. Other components can use this metadata to exchange data with one another in a correct format or to perform predefined read and write operations on the database.

6. With the intention to make indicators such as Tempscore available to external applications we created the *gateway service (GS)*. It serves as an independent interface that can be adjusted to support multiple types of indicators without adapting the established calculation process.

7. In order to guarantee full independence of the GS from the rest of the application we set up a separate *indicator database* storing indicators for further export.

8. A potential `graphical user interface (GUI)` displays indicators provided by the GS. However, the realization of a `GUI` is outside of the scope of this thesis.

9. This detail of Figure 1 hints the possibility of computing other indicators in the future. A new controller completely unassociated with the BC of our current Tempscore solution will then be able to post its own indicator to the GS.

10. The GS exports the indicator to external bank systems. A bank system (or customer system) represents applications outside the Tempscore solution that take indicators as input.

Note that it is possible for a single component to be used by more than one of the three main processes. Therefore, processes overlap occasionally on a component level abstraction.

3.1.1 Importer

The importer process uses the `FTP` server, BC, *commons* module, and Tempscore DB. At first data providers transmit files filled with company data, market data and environment data that we store on the `FTP`. The BC subsequently fetches these files and starts transforming their contents into a format suitable for our database. In general this means the BC filters out unimportant information and converts the raw data into the relational scheme of our database in order to store it for further use. All metadata about the database's structure originates from the *commons* module.

Data files themselves are depending on the provider either in `.csv` or `.xlsx` format. A download via an `API` is only possible in some cases, hence we decided to stick solely to the file sources. Most of the input data reaches our system in `.xlsx` format, whereas company data arrives as `.csv`.

3.1.2 Calculator

The calculator computes the Tempscore indicator based on previously imported data. The calculation itself is conducted by multiple calculation partitions at the same time in order to fasten the process. Besides, the BC acts as a central management instance and gets to decide how many partitions are started up based on the quantity of instruments we want to asses with a Tempscore rating. Furthermore, the BC sends packages including the parameters necessary for the indicator calculation to each partition. After a partition finishes its computations, it saves

the results directly back into the database. Multiple open connections to our Tempscore DB have drawbacks as they are potentially harder to monitor. On the other side they significantly reduce time consumption. In Section 4.3 we will further discuss how we are able to orchestrate a large number of partitions.

The calculator can not exist completely without the importer because it needs a filled database to be of use. Nonetheless it is largely independent on a logical level. This means the calculation can be executed alone at any time, given the Tempscore DB already holds useful data.

3.1.3 Exporter

Exporting the results of the calculator is the third main part of our application. As implied by Figure 1 this process is not only logically independent from the importer and the calculator but also physically divided as it uses a separate database. This is necessary due to the requirement to expand the exporter for supporting a new indicator at any time without the need to alter anything beyond the GS and the *indicator database*.

In our current solution the BC posts the results to the GS. Subsequently, the GS then persists the results in a database solely made to store different types of indicators. If we want to add a new indicator besides Tempscore we would need to add an endpoint where a different BC can send its data to. The BC also serves as an `API` for external applications and even a small `GUI` we are going to build for showcase purposes. However, the `GUI` is not part of this thesis.

3.2 Lifecycle

Our three main processes not only differ in their tasks but also in their active times. This section discusses the trigger mechanisms of each process. In addition we investigate the data flow between the components in more detail.

3.2.1 Trigger

Since the data files do not reach the `FTP` server at a fixed time we would need to trigger the importer ourselves shortly after a file upload to make sure processing commences as fast as possible. However, one of our requirements is to reduce the manual interactions with the system to a minimum. Thus, we equipped the BC with a file watcher that quickly reacts to new files arriving on the `FTP`. The file watcher reads in file contents and passes them on to other services within the BC for further refinement which starts the import.

The calculation process is either started by the BC directly in connection with the import process (after new file contents from `FTP` are stored) or can optionally be initiated manually if needed. The manual start is used for test purposes and as a safety feature to recover from possible failures. For example if the calculation halts due to corrupted import data needing to be modified by hand, a new calculation would only be possible after the arrival of new data files. Similarly, the export process begins automatically after the calculation or alternatively by hand.

3.2.2 Data Pipeline

In general, data passes from the importer over the calculator to the exporter. Figure 2 depicts this information flow in depth on a component level:

At first the file watcher in the BC (1) reads in files landing on the `FTP` and (2) persists the transformed contents to a database. Afterwards, the BC (3) queries the now refined data that can be handed over to the calculator. It further starts up multiple partitions to compute the indicators (as mentioned in Section 3.1.2), splits the data into smaller packages and (4) deploys each package to a distinct partition. Results take a straight route without any components in between and are (5) returned directly to the database. The BC then collects the results and (6) posts it to the GS. Similar to the BC the GS (7) persists to and (8) queries from its dedicated database. But in addition (9) provides all indicators for external use through various `API` endpoints.

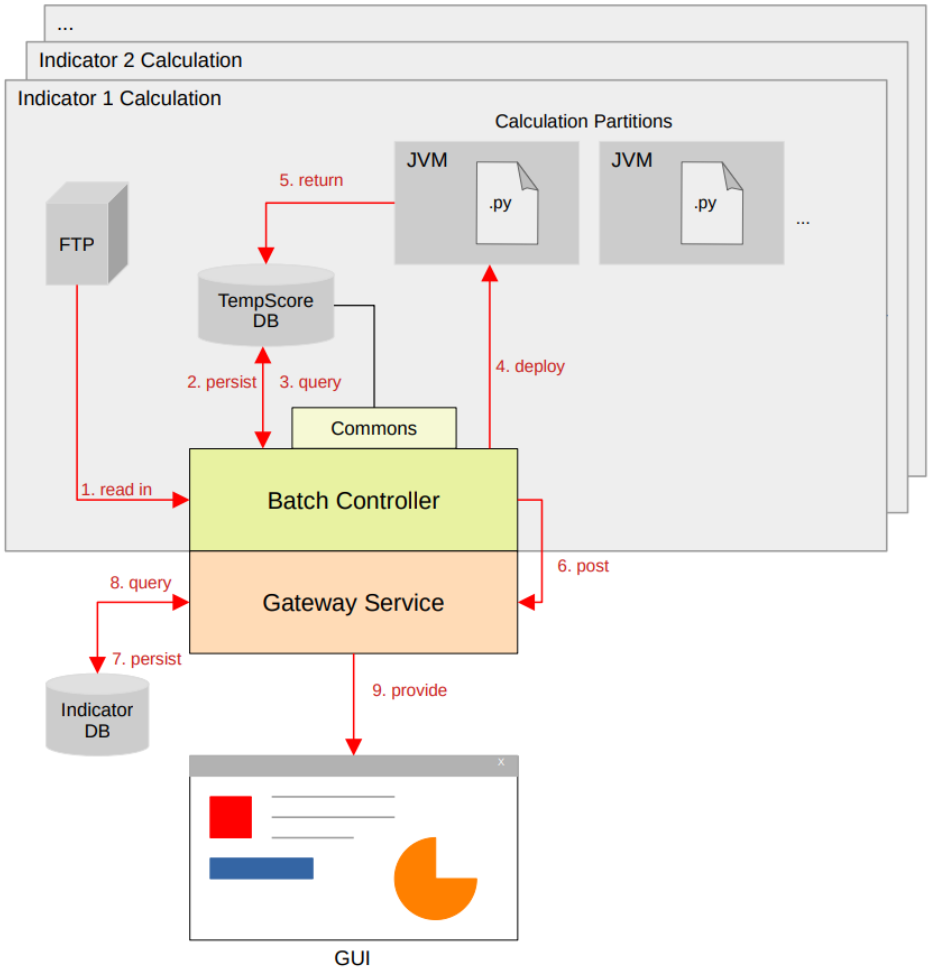


Figure 2: Visualization of the data pipeline within the Tempscore application. The numbering shows the successive steps in between the raw input data and the final indicator. The direction of the arrows resembles the direction of the data flow.

4 Implementation

To further examine our Tempscore solution from a technical perspective we will dive into the details of the implementation. This section briefly discusses the technologies we built our solution on and afterwards analyzes the most crucial parts behind the data import, the indicator calculation and the result export on the source code level.

4.1 Technologies

Spring Most of our source code is written in Java assisted by the *Spring Framework*. Spring is one of the most sophisticated Java frameworks currently available and offers a broad variety of tools supporting `object relational mapping (ORM)` functionality, batch operations as well as RESTful web services [2] out of the box. Additionally, our file watcher uses the *Spring Integration* library.

Every Spring application has a so-called *application context* that instantiates and manages objects. A Java object registered within the application context is called a *Spring bean*. To keep it simple we call a Spring bean just *bean*. A method annotated with `@Bean` is executed at startup and returns an object that gets registered in the application context (i.e. it becomes a bean). [2]

Docker and Kubernetes To be able to orchestrate our different components and partitions in the cloud we use *Docker* to containerize our application in combination with *Kubernetes*. Kubernetes is an open source system used to administer and scale these containers. The support for scaling is one key aspect of achieving parallel calculation in multiple partitions in the cloud.

Flyway Flyway is an open-source migration tool we are using to roll out database updates as well as the initial schemes of our Tempscore DB and Indicator DB. All migrations are listed within the commons module.

4.2 Data Import

This section presents the software features used by the importer process. It covers how to realize a file watcher with Spring and inspects the logic behind the file parser. Figure 3 showcases that all of the file processing logic as well as the management of the database connection is located within the BC. Additionally, the BC is responsible for partitioning the calculation process we examine in Section 4.3. The commons component features the database structure and other metadata.

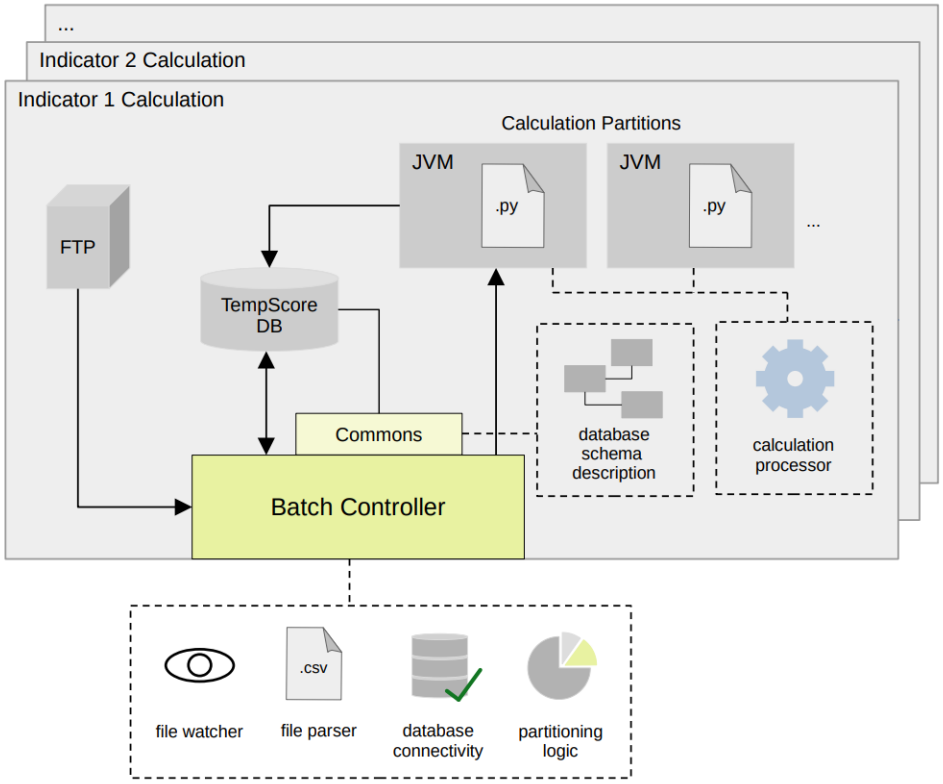


Figure 3: Most crucial modules for data import and indicator calculation. The dashed connectors imply where each functional program element is situated.

4.2.1 File Detection

We created our file detection mechanism based on the Spring Integration's `IntegrationFlow` class. It will be registered in the application context as a bean. In general the file detection includes the following components:

- The `SessionFactory` interface, which creates `Session` objects that enable us to read remote file contents via an `InputStream`. The current solution employs the `DefaultSftpSessionFactory` that also allows us to use more secure `SFTP` connections.
- The `SftpInboundChannelAdapterSpec` poses as a wrapper object for adjustable settings such as the path to the remote directory a target file is expected to be in. It takes a `SessionFactory` as argument and applies the specified settings to it.
- The `SourcePollingChannelAdapterSpec` defines the polling policy. The polling policy is the interval in which the file watcher checks for new files.
- A handler method specifying import actions when the `IntegrationFlow` detects a new file during a poll.

Listing 1 shows our realization of a file watcher. The file watcher consists of two methods. Both are annotated with `@Bean`, which means they get executed on application startup and the application context will manage the returned objects as described in Section 4.1.

Note that because of the `@Bean` annotation a `DefaultSftpSessionFactory` bean is registered in the Spring application context (line 1). At first we define a method that returns the `DefaultSftpSessionFactory` (line 2). The `DefaultSftpSessionFactory` has the ability to open a more secure `SFTP` connection. Within the method we instantiate the `DefaultSftpSessionFactory` (line 5). Afterwards we set crucial connection properties such as: private `SFTP` key file, hostname, port, and username (line 6 to line 10). At last, we return a `DefaultSftpSessionFactory` object (line 13).

The second method creates an `IntegrationFlow` object and takes a `DefaultSftpSessionFactory` as parameter (line 17). Spring recognizes the `DefaultSftpSessionFactory` bean in the application context and is able to pass it as an argument (i.e. Spring *injects* the `DefaultSftpSessionFactory` bean into the second method). We need to know where to scan for arriving files on the `FTP`. Additionally, we have to copy new files from the `FTP` to the local machine for further processing.

Hence, we create a `SftpInboundChannelAdapterSpec` object that encapsulates our `DefaultSftpSessionFactory` bean, the local directory as well as the `FTP`s remote source directory */ghg* (line 19 to line 20). We pass the `SftpInboundChannelAdapterSpec` object to the `IntegrationFlow` (line 22) alongside a poller that sets the interval in which we scan the `FTP`s remote directory for new files to ten seconds (line 23).

Based on all this, the `IntegrationFlow` now automatically opens a message channel used for the data transfer. We call such a construct *inbound adapter*. Moreover, we are able to specify import actions to perform when we indeed discover a fresh file (line 24 to line 26). These actions will be explained in more detail in Section 4.2.2.

```
1  @Bean
2  DefaultSftpSessionFactory defaultSftpSessionFactory() {
3      /* ... */
4      var defaultSftpSessionFactory
5          = new DefaultSftpSessionFactory();
6      defaultSftpSessionFactory
7          .setPrivateKey(/* path to private sftp key */);
8      defaultSftpSessionFactory.setHost("localhost");
9      defaultSftpSessionFactory.setPort(22);
10     defaultSftpSessionFactory.setUser("user1");
11     /* ... */
12
13     return defaultSftpSessionFactory;
14 }
15
16 @Bean
17 IntegrationFlow ghgDataInbound(DefaultSftpSessionFactory ftpSf) {
18     /* ... */
19     SftpInboundChannelAdapterSpec spec
20         = getSpec(ftpSf, localDirectory, "/ghg");
21
22     return IntegrationFlows.from(spec, pc -> pc.poller(pm ->
23             pm.fixedRate(10, TimeUnit.SECONDS)))
24         .handle((file, messageHeaders) ->
25             handleRemoteFile(file, messageHeaders, ftpSf, "/ghg",
26                 this.importFileDataService::importEmissionScopes))
27         .get();
28 }
```

Listing 1: Set up of an `IntegrationFlow` bean polling an FTP running on localhost in an interval of 10 seconds.

4.2.2 Data Extraction

We determine the appropriate import actions for new files in the `IntegrationFlow`'s handle function (Listing 1 line 24 to line 26). The *handleRemoteFile* method defines the import actions for files that arrive in the */ghg* directory (line 25). Its parameters are:

- a copy of the new file cached on our local system
- file metadata in form of a `MessageHeaders` object
- a `DefaultSftpSessionFactory` object opening the `SFTP` session
- the arriving file's remote directory name
- a reference to a consumer method extracting and processing the data (*importEmissionScopes*, line 26)

Our current approach is to consume files differently based on their location on the `FTP` server. Thus, at the moment, the Tempscore solution employs multiple inbound adapters with different consumer methods, one for each remote directory. Listing 1 only shows the inbound adapter

for the */ghg* directory. Although more sophisticated solutions are possible, we found it to be straightforward and efficient enough. Unfortunately, this method has some limitations: we must define a fixed directory structure and additionally need to implement another inbound adapter each time we add a new source.

In our implementation, the handler method calls the referenced consumer method. Each consumer method is a file parser responsible for extracting and processing content. The consumer method receives the copied file as a `FileInputStream`. We append the ending `.bak` after a file has been processed, rendering it invisible for our handler method because it verifies the type of a new file to be either `.csv` or .xlsx. This ensures a file is not processed a second time in the future. Finally, the duplicated local file gets removed. Only the `.bak` version of the original file on the `FTP` remains. Note that we have chosen the file ending `.bak` because it resembles the term backup.

The files on the `FTP` are currently divided into six separate directories as shown in Table 1. This means that we have exactly six different types of source files and exactly six inbound adapters. We periodically retrieve new files from the data providers and store them in their designated folder. Files stored in the same folder are required to have the same structural layout (e.g. identical headers in all .csv files) because otherwise the import logic may not extract contents correctly. Unfortunately, the complexity of the data renders it exceptionally hard to write a fully generic importer which is also maintainable. Note that under normal circumstances the source files are not subject to change.

Directory	Type	Content Description
/ghg	company data	CO_2 emissions in metric tons per company
/model	environment data	annual global emissions
/emissions	environment data	annual national emissions
/fxrates	market data	currency exchange rates and inflation
/tempchange	environment data	annual temperature anomaly
/instrument	market data	industry data/revenue/identifiers for instrument

Table 1: File system structure on the `FTP` server. Each directory holds exactly one type of files which are periodically added by a data provider.

After data is extracted from a source file the import process converts it to a format compatible with the structure of our relational database. We achieve this with the `ObjectMapper` of the *Jackson* library that is able to convert a `.csv` file to a Java object. For `.xlsx` we use a company internal library. Eventually we store Java objects derived from `.csv` files and `.xlsx` files in the database with the help of the *Spring Data JPA* tool that provides the necessary `ORM` functionality. Spring Data JPA features persistence entities and data repositories:

- Persistence entities are Java classes representing a table and are annotated with `@Entity`, which tells Spring instances of the class are `JPA` entities that will be persisted to the database. Persistence entities must have an `@Id` annotation symbolizing the primary key.
- The goal of data repositories, is to reduce boilerplate code by managing data access for a certain persistence entity. It takes a persistence entity's class and the type of its id attribute as type arguments [2] which means there should exist a data repository for every persistence entity. It is possible to predefine queries as well as pagination and sorting methods.

Spring registers a bean of each repository in the application context which makes it accessible to use. Furthermore, Spring Data JPA also handles the database connectivity and allows us to execute `CRUD` operations.

4.3 Temperature Score Calculation

Based on the gathered data we are able to assign a sustainability rating to imported instruments. We have already discussed that the calculation alongside its input is distributed among multiple processes in Section 3.1.2. This section clarifies how the partitioning logic of our application works.

4.3.1 Partitioning

A cluster of calculation partitions carries out the same calculations for different blocks of instruments at the same time. In Figure 4 we can see the BC conducts these parallel operations. A managing instance called *partition step* within the BC is able to divide the data for the Tempscore calculation into multiple batches. Each data batch gets deployed to a different calculation partition (called *calculation step*) running in its own `JVM`. Every calculation step subsequently runs a calculation processor using a python script to perform the indicator computation. The python script implements the mathematical logic discussed in Section 2.4. The calculation processor as well as the definition of the calculation step is provided as a standalone Java module within the Tempscore project and will be explained in more detail in Section 4.3.3.

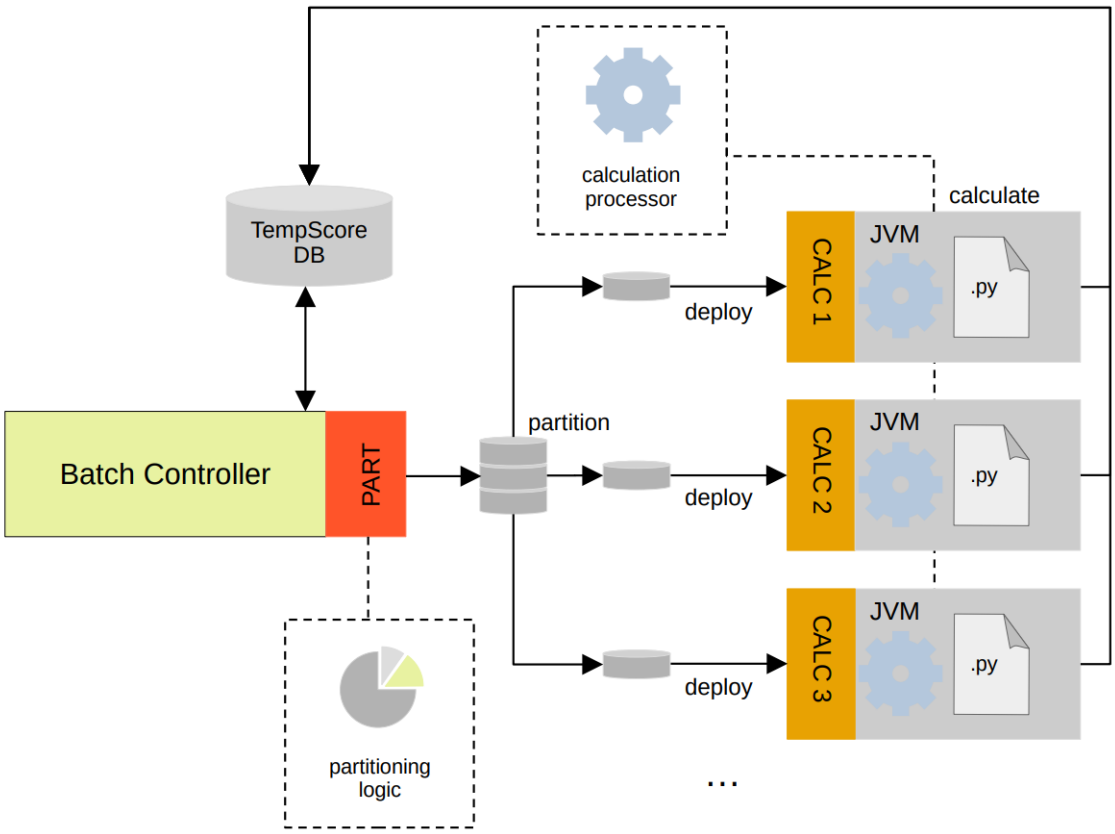


Figure 4: The procedure behind the step division begins with the partitioning logic in the BC, where a partition ($PART$) step creates a calculation ($CALC$) step as its own Java process for each data batch. The calculations themselves are performed within the standalone processes.

To fully understand the splitting operations in the BC we first need to know about some basics of Spring Batch we use in our implementation shown in Figure 5. Spring Batch allows us to define jobs consisting of one or more steps. A `Job` object schedules its assigned steps. It knows when each step concludes and starts the next in line. We initially declare one job with a single partition step. A `Step` object receives data as input and performs arbitrary tasks. The input data of a step can be further subdivided into multiple input data batches by a Spring Batch `Partitioner`. Afterwards, Spring Batch is able to attach each input data batch to a calculation partition that will process the payload. Because the calculation partitions are realized as Spring Batch steps we also refer to them as calculation steps (especially when discussing implementation details).

We are now confronted with four questions:

1. How can we create a job?
2. How is the Spring Batch `Partitioner` able to split a step?
3. How do we pass usable data to a calculation step?
4. How can we execute a calculation step?

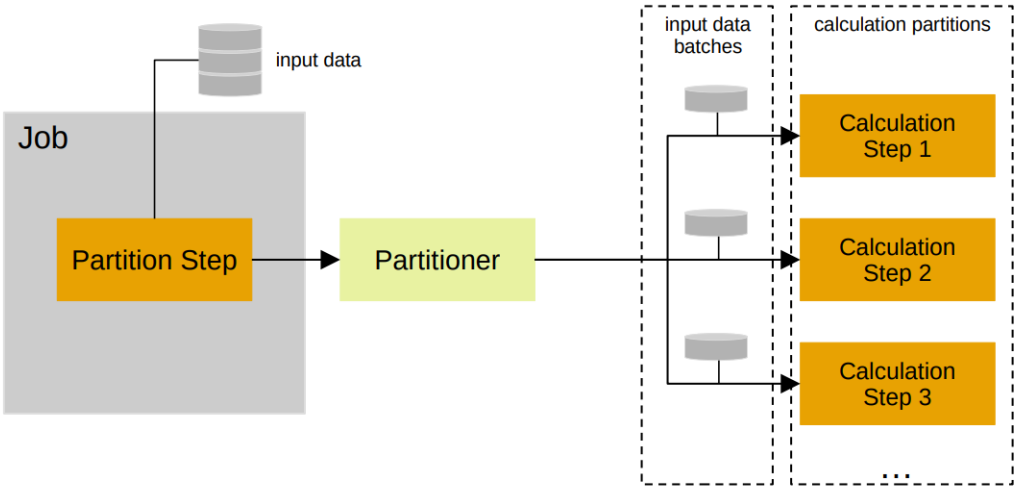


Figure 5: A job with one partition step uses a `Partitioner` to split the input data. The `Partitioner` creates a calculation step for each chunk of input data.

In order to generate a job linked to one step we created two methods annotated with `@Bean` as shown in Listing 2. The annotation tells Spring to execute both methods at startup and to register the returned objects in the application context as described in Section 4.1.

1. The *job* method (line 2) creates the desired `Job` instance. The only parameter is the partition step that should be assigned to the job. We use a `JobBuilderFactory` to configure and build our `Job` object (line 3).

   Each job must have a string identifier, set by the `JobBuilderFactory`'s *get* method. We append the output of a random number generator to the job identifier to assure that the job identifier is unique (line 4). Additionally, we have to link the partition step parameter to our job (line 6). Finally, we create our job instance (line 7).

2. The *partitionStep* method (line 11) returns the partition step for our job. The *partitionStep* method also takes one parameter being a `PartitionHandler` object. The `PartitionHandler` defines the deployment configuration needed for starting the calculation steps and will be evaluated further in Section 4.3.2. Similar to the *job* method we use a factory, in this case a `StepBuilderFactory`, to instantiate our step (line 12).

A step also needs a unique identifier but due to the fact that Spring Batch identifies a step not only by name but also by job affiliation and our jobs only consist of a single step, each partition step is already unique. Hence, no random number generator is needed. We simply label the step *partitionStep* (line 13).

Still, we need to split the work and the `Partitioner` is the key to the parallel calculation we desire. The *partitioner* method takes a string and a reference to another method holding the partitioning logic as parameters (line 15). The partitioning logic will be described in more detail later. The string parameter represents the name prefix of the created calculation steps. In addition, the `Partitioner` automatically appends a random number to each name making them unique. At last, we set the partition handler (line 17) and instantiate the partition step (line 19).

```
1  @Bean
2  public Job job(Step partitionStep) {
3      return this.jobBuilderFactory
4              .get("job" + new Random().nextInt(Integer.MAX_VALUE))
5              /* link step to job */
6              .start(partitionStep)
7              .build();
8  }
9
10 @Bean
11 public Step partitionStep(PartitionHandler partitionHandler) {
12     return this.stepBuilderFactory
13             .get("partitionStep")
14             /* register partitioner */
15             .partitioner("calculationStep", partition(/* ... */))
16             /* define tasks to perform in each calculation step */
17             .partitionHandler(partitionHandler)
18             /* ... */
19             .build();
20 }
```

Listing 2: Example of a job with one step, split by a Spring Batch `Partitioner`. The `@Bean` annotation tells Spring to register a `Job` object and a `Step` object as beans in the application context.

The `Partitioner` registers calculation steps with a `String` value as unique identifier and splits the input data into multiple data batches. Moreover, the `Partitioner` sets each calculation step's *execution context* (i.e. assigns a data batch to each calculation step). In Spring Batch, the execution context is a map serving as a wrapper for a step's payload.

In our solution we pass a reference to a *partition* method containing the partitioning logic to the `Partitioner`. The partition method takes all the calculation input data, splits it into partitions of fixed size based on a predefined batch size value, and derives one execution context per input data batch. Further, the `Partitioner` creates a calculation step for each data batch and links the calculation step to the corresponding execution context created by the partition method. The calculation input data is a list of instruments mapped to their company data.

Thus, we have now clarified how the input is distributed between multiple calculation steps. Nevertheless, we currently have not defined how to start these calculation steps. This is the task of the `PartitionHandler`.

4.3.2 Partition Deployment

Remember, the partition step normally employs a `PartitionHandler` to define the necessary deployment configuration to start the calculation steps as mentioned in Section 4.3.1. More specifically, we use the `DeployerPartitionHandler` class inheriting from `PartitionHandler` to control the execution of all calculation steps.

The `DeployerPartitionHandler` takes a resource as an argument, which can be an arbitrary `.jar` file. This is an immensely helpful Spring Batch feature, enabling us to initiate a step defined in another Java program completely independent from the BC. Therefore, the Java module containing the definition of the calculation step and the calculation logic itself can be implemented as its own isolated microservice. Furthermore, the `DeployerPartitionHandler` starts a Java process in a distinct `JVM` instance for each calculation step such that they can work in parallel. Besides, when using the `DeployerPartitionHandler` we can also specify command line arguments for the Java processes. In our current solution the BC utilizes this feature to pass the connection string of the Tempscore DB to each starting calculation step. The `DeployerPartitionHandler` is even able to start a `JVM` inside a Docker container in the cloud.

Note that Spring Batch persists metadata, input data (i.e. execution contexts) and hierarchies of all known jobs and steps as `JSON` to automatically generated tables in the Tempscore DB. Entries can be identified by a unique `String` value that in the case of jobs and steps represents their name. This is extremely favourable regarding our architecture. Both the BC and the calculator instances have access to the same database. By that, information about steps can be shared across multiple Java processes. A remote calculation step running on another `JVM` knows its identifier thus being conveniently able to search for its workload (assigned by the Spring Batch `Partitioner`) in the database on its own.

4.3.3 Calculation Processor

In the previous section we have learned how the workload is divided into multiple partitions and how the calculation steps are started. Yet, we don't know how to start the indicator calculation itself. Listing 3 illustrates the calculation step launched for each input data batch.

Analogous to the *partitionStep* method (Listing 2) the *calculationStep* method creates a calculation step bean (line 2). We also use a `StepBuilderFactory` to build our `Step` object (line 3). In contrast to the partition step the calculation step is no subject to partition and processes items directly. We pass instructions as a `Tasklet` bean (line 4). The `Tasklet` itself is an interface for a callback method, which we can use to define the course of action and finally kick off our indicator calculation.

We get our `Tasklet` bean from the *pythonCalcTasklet* method (line 10). The `@StepScope` annotation allows the bean to directly access the current execution context. In combination with

the `@Value` annotation we are able to inject a value stored in the execution context with the following syntax: `"#{stepExecutionContext['<value_name>']}"`

Hence, we are able to access the input data batch which was assigned to the calculation step by the `Partitioner` as discussed in Section 4.3.1 (line 11 and line 12). Moreover, we inject the unique partition number which identifies the calculation step for logging purposes (line 13 and line 14). Remember that the `Partitioner` automatically appends a random number to each calculation step name. This number is also stored in the execution context.

Finally, the callback function starts the calculation processor which creates a Java `ProcessBuilder` capable of executing a python script with the *calculationInput* `ArrayList` stored in the execution context as input (line 16 to line 23). The elements of the input `ArrayList` are `JSON` strings. The python script then calculates the indicator and prints the results. Simultaneously, the calculation processor reads in the printed data. Ultimately, the results are transmitted to the Tempscore DB.

The `ProcessBuilder` in the calculation processor does not pass all of the input at once but rather pages it. Smaller batches of a predefined size reduce the impact on the bulk of the data when the python script fails as it only affects a single batch. However, choosing a chunk size too small will increase execution time drastically, because continuously restarting the python process produces an immense overhead. We currently pass 100 instruments at once to a single python process.

```
1  @Bean
2  public Step calculationStep() {
3      return this.stepBuilderFactory.get("calculationStep")
4              .tasklet(pythonCalcTasklet(null, null))
5              .build();
6  }
7
8  @Bean
9  @StepScope
10 public Tasklet pythonCalcTasklet(
11         final @Value("#{stepExecutionContext['partitionNumber']}")
12             Integer partitionNumber,
13         final @Value("#{stepExecutionContext['calculationInput']}")
14             ArrayList<String> calculationInput) {
15
16     return (/* ... */) -> {
17         /* call process builder with
18         calculationInput array as input */
19
20         /* read in printed data */
21
22         /* store results in database */
23     };
24 }
```

Listing 3: The calculation step directly performs the indicator computations in a `Tasklet`.

4.3.4 Indicator Computation

We distinguish between two calculation types based on the amount of data we are able to acquire for a certain company:

Standard Case The standard indicator is the potential global warming in degrees Celcius up to a target year as explained in Section 2.4. Under normal circumstances, we have gathered enough data about a company to supply the calculation processor with the necessary input. At first, we calculate the company's emissions to revenue ratio over the last two years and assume this change is a stable trend. Based on the trend we can now approximate the absolute emissions up to a target year. We then apply the absolute emissions to another model representing the temperature increase per gigaton (GT) CO_2. [1]

Surrogate Case In contrast to the standard method, the surrogate method is applied if the calculation for an instrument has insufficient input data. Sometimes a data provider is not able to deliver information in the desired quality. This is especially the case for the sparsely available environment data that is sometimes completely missing at all. For this reason we use similar companies operating in the same industry sector to estimate the ecological footprint of an otherwise uncertain instrument. In general the surrogate method consists of multiple successive steps:

1. After the application finalizes the standard calculation it tags all instruments which are still not rated yet.
2. The BC then initiates the follow up surrogate calculation that maps a tagged instrument to a list of companies operating within the same sector.
3. The ten closest companies, in terms of a normalized Euclidean distance with respect to the reported emission data, are chosen as surrogate candidates.
4. Afterwards, the surrogate candidates are ranked by their Tempscore indicator and the best as well as the worst are removed from the candidate list to omit outliers.
5. Finally, we return the average of the remainders' indicators as warming potential for the unrated instrument.

4.4 Result Export

This section briefly summarizes how the indicator export to the GS is implemented. At the core of this component lies the *Spring Web* library supporting us with everything we need to implement a `RESTful` web service in Java.

After the calculation has come to an end, the BC posts the results to a `RESTful API` endpoint residing in the GS. The GS operates unaccompanied from the rest of the program and is used to store and provide different indicators. As mentioned in Section 3.1.3 it even uses its own database. Spring Web supports us with the mandatory features we need to implement the `RESTful` service. Currently there exists only one endpoint which takes Tempscore results as input

and another one providing these results to external applications.

In Listing 4 we can see the endpoint used to access all Tempscore indicators. In line 1 we define the `TempscoreController` class as a Spring Web `RestController` that holds various endpoints. Through the `@RequestMapping` annotation in line 2, we are able to define a base path serving as a prefix for all endpoint URLs in this `RestController`. Spring Web gives us the possibility to use every major `HTTP` method with the correct annotation. An example of this can be seen in line 6. The endpoint annotated with `@GetMapping` returns all Tempscore indicators.

```
1  @RestController
2  @RequestMapping("/api/v1/tempscore")
3  public class TempscoreController {
4      ...
5
6      @GetMapping("/all")
7      public ResponseEntity<...> getTempscoreIndicators() {
8          ...
9      }
10 }
```

Listing 4: Implementation of a Spring Web `REST` controller with a `GET` endpoint. The endpoint can be reached at the URL *<hostname>/api/v1/tempscore/all*.

5 Evaluation

In this section we assess quality factors such as performance or vulnerability against faulty data. We also try to find the moment from where on our approach outperforms a sequential method.

5.1 Performance Test

We designed the system to use the advantages of parallel execution providing high performance with a large amount of data. However, this does not come without an administration overhead. We have learned in Section 4.3.1 that each calculation step is attached to a different process, which consumes a substantial amount of time at startup. We will now compare execution time of a partitioned job to a singular step instance with different quantities of input data.

Our measurements rely on the built-in logging of Spring Batch that stores the execution time of each step to an automatically generated database table. For the absolute `CPU` time we add up three factors:

1. The duration between the calculation trigger and the start of the last partition.
2. The accumulated value of each partition's execution time.
3. The time span from the moment the fastest partition finishes until the partition step terminates.

Table 2 shows that splitting up the calculation is indeed much faster. Positive effects even apply to a relatively small input size of 1000 instruments. Nevertheless, the expected overhead leads to increased overall `CPU` usage visualized in the last column. The absolute `CPU` time consumption is increasing steadily alongside the number of partitions in use. Furthermore, we can observe the partitioning for 1000 instruments is becoming impracticable for a factor greater than eight. All of the findings also apply to 5000 input instruments. Despite, we should not forget to mention the growth of the time span between a computation of four and eight partitions is significantly slower for the larger data sample. This hints that a large number of partitions is only reasonable for a large quantity of data.

5.2 Robustness

To test the affect of faulty data on the application we intentionally pass nonsensical data as input and analyze the outcome. The input data itself is transmitted to a python script in the calculation processor in batches of fixed size as mentioned in Section 4.3.3. If the script encounters an input it can not process it will fail. But due to the data division a failure will only affect a single batch.

Even a far more serious exception on partition level does not lead to an unsuccessful termination of the whole application. This is because the partitions act independently from one another. Only a fraction of the indicators will be lost. Spring Batch further persists each partition's status which allows an exception handler to restart failed calculation steps.

# instruments	# partitions	real time	CPU time
1000	1	2m 4s 211ms	2m 4s 211ms
1000	2	1m 21s 821ms	2m 13s 301ms
1000	4	1m 9s 410ms	2m 33s 692ms
1000	8	2m 3s 424ms	6m 52s 600ms
5000	1	8m 7s 595ms	8m 7s 595ms
5000	2	5m 7s 133ms	9m 32s 535ms
5000	4	2m 59s 570ms	10m 9s 481ms
5000	8	3m 29s 353ms	19m 53s 221ms

Table 2: Real time next to the absolute CPU time the system needed to calculate the Tempscore indicators for different sizes of input based on the degree of workload division. The tests were conducted on a *Dell Inspiron XPS 15 9560* containing an *Intel Core i7-7700HQ* 2.80GHz quad core `CPU` with 32GB `RAM` on *Windows 10*.

6 Current Limitations and Future Work

Although the Tempscore application itself has been finalized we still have tasks in the backlog. The system can be improved in several ways that are currently too time extensive or not necessary. Furthermore, we will discuss possible uses of the Tempscore indicator within the uni software plus GmbH.

6.1 Cloud Integration

One requirement is to lift the application into the *Google Cloud*. The cloud deployment currently has the highest priority in the backlog and we are looking forward to conclude this task in the near future. With the help of the *Spring Cloud Task* library we are able to bring our Spring Batch partitioning logic to the cloud.

Remember, the `DeployerPartitionHandler` described in Section 4.3.1 is able to start an arbitrary Java executable. With Spring Cloud Task it is also feasible to define a container holding a `.jar` as the deploy-able resource. In our case we build a Docker container around the calculation processor and provide it within the Kubernetes cluster. Each partition will be started as a separate instance of a *pod*. A pod in the context of Kubernetes represents the smallest unit that can be created, managed, and most importantly for us, scaled.

6.2 Further Improvements

This section explains what aspects of our program will be improved or are possible candidates for a rework.

File Watcher In Section 4.2.2 we discussed the drawbacks of our file watcher implementation. Although we need to configure another inbound adapter for each new data source, our method has proven to be sufficient. But this may be not the case anymore if we need to extensively extend the number of data sources. Therefore we could be required to perform refactorings in the future.

Partition Preprocessing Currently all of the input is gathered at once before the calculation is partitioned, which takes a significant amount of time. To further reduce the duration calculation partitions could try to query for necessary data themselves. A possible solution would be to not pass the entirety of the input but just the identifiers of the instruments each partition needs. All the heavy lifting such as joins would therefore happen in parallel within calculation processor instances. This would also reduce the amount of data transferred between the partition step and his calculation steps. However, the database utilization would increase significantly.

6.3 Areas of Application

The Tempscore indicator will be available as a standalone product. However, it is designed to serve as a complementary feature to enhance established finance software developed by uni software plus GmbH. Targeted customers are regional financial institutes. Some of them even

possess banking software that do, in fact, consider sustainability as an investment factor. Yet, these ratings are mostly kept simple.

7 Conclusion

In this thesis, we introduced a system capable of processing large amounts of market data, company data and environment data and merges them to a meaningful sustainability rating for individual companies. At first we had an in depth look at our micro service architecture and analyzed each component alongside its communication paths in detail. We mentioned the data has to traverse three main processes leading us to our final indicator, being import, calculation, and export. Particularly, the importer using the file watcher in combination with the file parser and the `FTP` server is a good example for the collaboration of multiple components within the Tempscore application.

Afterwards, we discussed how the split into multiple calculation partition works on a source code level and dived deeper into the functionalities provided by the Java Spring Framework. Due to our efforts to commence calculations in parallel we achieved a more than reasonable execution time. Besides the software engineering aspect we also obtained some insights into the basic mathematics, that can be considered as the centerpiece of the indicator computation itself.

Aside from being a software project, the Tempscore solution is a chance. A chance for private and corporate customers to respect our precious blue planet by devoting capital to more sustainable investments.

List of Tables

List of Figures

Listings

List of Acronyms

BC batch controller

FTP file transfer protocol

DTO data transfer object

GS gateway service

GUI graphical user interface

ORM object relational mapping

JPA java persitance API

CRUD create;read;update;delete

API application programming interface

REST representational state transfer

References

[1] M. Panholzer, M. Aichinger, and M. Aeberhard. In *A transparent climate rating based on the warming potential*, pages 1–11, 2022. [unpublished].

[2] I. Pivotal. Spring framework documentation. `https://docs.spring.io/spring-framework/docs/current/reference/html/`, 2022. [Online; accessed 27-August-2022].

[3] H.-O. Pörtner, D. Roberts, M. Tignor, E. Poloczanska, K. Mintenbeck, A. Alegría, M. Craig, S. Langsdorf, S. Löschke, V. Möller, A. Okem, and B. Rama. In *Climate Change 2022: Impacts, Adaptation, and Vulnerability. Contribution of Working Group II to the Sixth Assessment Report of the Intergovernmental Panel on Climate Change*, pages 1–33. Cambridge University Press. In Press., 2022.