Big Data Management
Building a Big Data Architecture

Project guidelines (P1): contextualization, data ingestion and landing zone

1. Stages (P1)

1.1 Contextualize the problem

Domain and value

Before embarking on a big data project, it is essential to clearly **define the domain** you will be working with. You are free to choose any domain that allows you to align your project with your interests (e.g. healthcare, finance, IoT), as long as it contributes to the development of robust data management pipelines. That is, select a subject that excites you, but also presents meaningful challenges to solve by exploiting data assets.

Once the general context of your system is established, the next crucial step is to **determine the business value** it aims to deliver. That is, what do you want to do with the data once it has been processed? What problem(s) does your pipeline solve? What insights or improvements does it provide? Defining a clear and measurable objective is fundamental for designing an effective and well-structured architecture. Logically, you can aim at solving different types of tasks. Additionally, you need to define **how to provide this value**. Are you going to train ML models? Are these going to require interacting with Deep Learning architectures? Are you going to monitor real-time behavior?

Ideally, you would analyze the selected market, detect potential needs and define a solution for the identified problem. The solution should be innovative and, preferably, there should be a numerical quantification regarding the degree of improvement. Nonetheless, we acknowledge that you do not have the resources to perform such analysis, so it is enough to define a solid project that provides some type of value for an imaginary company. Nonetheless, being creative in the development of the context will be positively evaluated.

Important note: BDM is not a modelling/analysis course. Hence we will not evaluate the quality or correctness of the tasks you implement when exploiting the data (more on this in the second part, do not worry about it at this point). The goal is not to obtain high-quality insights or thoroughly train complex models, but rather to properly define the data flows and the interactions. As such, you should still consider what you aim to achieve and how to do so, as it will help to structure your system. However, whether you actually achieve the exploitation goals will not be considered.

Data sources and datasets

Once you have contextualized your project, you have to **select the data sources** to extract the data from. Since you most probably do not have access to data sources, you may want to explore popular repositories to find interesting data you may want to work with. Some examples include Kaggle, UCI Machine Learning Repository, AWS Open Data, or Google Dataset Search. Here it is important to understand what type of data you will be processing, which directly ties with the value you aim at providing. Broadly, data falls into three categories:

- **Structured Data**: highly organized and schema-based, making it easily searchable and processable. It is typically stored in relational databases, where data is arranged in tables with clearly defined rows and columns. Common formats include SQL databases or CSV files.
- **Semi-Structured Data**: has some level of organization but does not conform strictly to predefined schemas. Unlike structured data, it can accommodate changing attributes and hierarchical relationships, but this flexibility can present challenges in consistency, indexing, and querying. It is often stored in adaptable formats such as JSON or XML.
- **Unstructured Data**: lacks a predefined schema and does not fit neatly into traditional databases. It includes raw, complex data types such as images, videos, audio recordings, emails, PDFs, and social media content.

Each category of data is used to capture different domains, serves different purposes and requires specialized tools and techniques for efficient storage, retrieval, and analysis. Hence, before starting the data collection process, you **need to understand what type of data you need** for the specific problem you want to choose and what requirements does the processing of such a structure represent.

Tabular (i.e. structured) data is the most popular variant due to being straightforward to work with. However, you have probably worked with it in several other courses (and will do so in several more). Hence, in this course we encourage you to work with semi-structured or, even, with unstructured data. During the course you will see technologies that will allow you to efficiently store and process these formats, so you can get some hands-on experience on handling non-tabular data. Even better would be to combine the processing of several types into the same architecture, defining different processing paths for each. Towards the end of the document we provide a high-level example design of a complex pipeline to help you in the definition of the architecture.

Tasks

To adequately contextualize your project **you need to**:

1. Select a domain: choose an area that interests you and aligns with Big Data management.
2. Define the business objective: clearly articulate what question your pipeline answers and the value it delivers. Also, state how you aim at providing this value.
3. Identify and acquire data sources: understand what type of data you need and search among available repositories to find it.

Factors that will **positively contribute** towards the grade:

1. Employing several data sources (e.g. fetching data from different APIs) and/or several datasets (e.g. given a social media API, that you fetch both user profiles and posts).
2. Employing semi-structured and/or unstructured data, preferably in combination with structured data to define different tasks or different data flows.
3. Great contextualization of the project (i.e. well thought-out domain and business value selection).
4. Using “Big” Data (see the second point below).

Additional considerations

Synthetic data usage: we acknowledge that you might have a certain idea to implement, but not the data to do so, especially if the envisioned workflow is complex. Given that the priority in this course is the adequate definition of processing pipelines, we accept the employment of synthetic data that you have generated, given that it helps you to implement a more interesting project. Note, however, that we encourage you to first search among popular repositories, as processing real-world data inherently presents more challenges. If you opt for synthetic data, explicitly state the reasoning behind using it and provide the generation scripts you employ.

Data volume: the *B* in BDM stands for *Big*. Hence, in an ideal scenario, the data you employ should be of a decently large size (e.g. not a couple of CSV files of a few KBs) to truly showcase the capabilities of your pipeline. This can apply both to the amount of data (e.g. number of files) as well as its size. Similarly as with the previous consideration, we acknowledge that finding high amounts of data that fits your desired project is not always possible. Hence, it is not a hard constraint to use a lot of data, but doing so will positively influence the evaluation. Once more, synthetic data (obtained, for instance, from interpolating real data) can be used to that end.

1.2 Data ingestion

Data ingestion is the first and foundational step in any big data architecture. It involves acquiring raw data from various sources and transferring it into a system where it can be processed, stored, and analyzed.

Batch vs. Streaming

One of the most critical decisions data engineers must make when designing data ingestion pipelines is determining the ingestion **frequency**. Depending on the rate at which data is fed into the system, ingestion can be broadly categorized in two:

- **Batch Ingestion:** data is collected in fixed-size chunks (batches), often at scheduled time intervals (e.g. hourly, daily). That is, data is fetched from APIs, databases or other sources at specific moments, and the extracted instances (the batch) tends to follow some criteria (e.g. datasets uploaded in the last day). This approach is suitable for scenarios where data processing does not need to be immediate and we can afford to query data at specific moments and store it before using it.
- **Streaming (Real-Time) Ingestion:** data is continuously ingested and processed as it arrives, enabling real-time analytics and applications. Streaming ingestion provides the ability to monitor and react to data as it flows, facilitating decisions and actions based on the most up-to-date information. However, streaming ingestion requires a more sophisticated infrastructure to ensure that data is processed and delivered without delay.

Data paths

Streaming has become a necessity in recent years due to the development of systems that provide a constant influx of data (e.g. sensor data, social media feeds, or financial

transactions), which can be used to monitor certain behaviors in real-time. However, this complicates the management aspect, as it requires a robust infrastructure and continuous resources. Moreover, a large quantity of data does not need to be processed continuously, so batch processing is still fundamental for big data architectures. In most complex pipelines, there exists a **combination of both approaches**, each addressing different data sources and handling data designed to fulfill different goals. That is, different data *paths* are defined:

- **Hot Path** (Real-Time Processing): This path is dedicated to handling time-sensitive data that requires immediate action (e.g. detecting fraud in financial transactions or triggering alerts), ensuring that data is processed and acted upon as quickly as possible. It typically involves high-throughput systems and low-latency processing frameworks that prioritize speed and reliability.
- **Cold Path** (Batch Processing): This approach handles data that do not need to be processed immediately, and that is expected to be accessed long after it has been obtained. It is often obtained through scheduled batch jobs. This can include processing logs, performing large-scale ETL (Extract, Transform, Load) operations, or running complex analytics that benefit from massive datasets and processing power.
- **Warm Path** (Near-Real-Time Processing): Positioned between the hot and cold paths, the warm path processes data that is not necessarily urgent but still requires timely analysis. Data in this path does not need immediate processing and can even be stored temporarily for short periods. For example, this could involve monitoring patterns over the past few minutes or hours to adjust operational workflows or improve machine learning models.

You need to decide which **ingestion policy** works best for your project. If you want to continuously monitor an influx of data to provide real-time processing or updates, you need to set up a streaming ingestion. If that is not relevant for the task you address, you can opt for batch processing. Ideally, you would want to apply both policies to define several processing pipelines. Once that is decided, you need to understand how to apply the selected ingestion pattern to the sources previously selected.

Important note: You are already familiar with batch ingestion from other courses, but data streaming is probably a new topic. You will learn about its theoretical aspects towards the end of the BDM course. Nonetheless, we encourage you to explore streaming hands-on in the project, as it is an interesting alternative. Setting up a simple streaming environment is not difficult, and you can even generate your own streaming data if you can not find any sources (which follows the synthetic data point raised before). Both of these can be done by using Apache Kafka, which is the most popular tool for this task and the one we recommend. To process the streaming (more on this on the second part of the project, at this point is not needed) you can use Apache Kafka Streams.

Ingestion mechanisms

The most straightforward data ingestion approach is to download files (batch) from the sources and manually place them into the pipeline. **This is not acceptable**. We want you to automate, at least, the task of introducing the data into the pipeline. That is, if you download files and store them locally (which we do not suggest either way), you need to implement a mechanism that reads these same files (either batch or via streaming) into

the pipeline. Many modern data sources provide APIs that allow programmatic data retrieval, so a more interesting approach would be to use APIs to obtain your data, which requires scheduling the API calls. In this way you are directly interacting with the sources, which defines a more scalable solution. Logically, other methodologies such as database connection or web scraping are equally valid.

Tasks

To implement the data ingestion **you need** to:

1. Select an ingestion strategy (batch, streaming, or hybrid) for each of the data sources you have selected. This should be in accordance with the type of processing (i.e. path) that each data should undergo to fulfill the desired goals.
2. Choose appropriate tools to automate and manage ingestion.
3. Implement your ingestion tasks.

Factors that will **positively contribute** towards the grade:

1. Implementing both batch and stream processing pipelines and define several processing paths.
2. Ingestion systems that communicate with external sources (i.e. that do not read files from your system, but rather that access non-local environments).
3. Scheduling data ingestion tasks (e.g. read batch data from an API every hour).

1.3 Landing zone

The next step in the pipeline is to store the raw data you have ingested in a single repository, which we define as the landing zone. Having a common repository through which all the data enters into the system provides several benefits. Some of the most important ones are:

- You homogenize the starting point of your processing paths. That is, the system becomes agnostic to the consulted data sources and how to access them, and starts directly from the data.
- It ensures data availability, traceability and reproducibility, as you can always consult the original data.
- It does not force data to adapt to a given model (read-first, model-later) right away, as opposed to traditional databases (model-first, read-later).

Note that you might **not want to store all the data** that you have ingested. Most of the data that comes from batch ingestion is stored, as its use is not immediate and we can afford to curate it before use (cold path). On the other hand, streaming data might be needed immediately (hot path), so rather than storing it, the data is directly processed to obtain some type of insight. As data exploitation is not yet introduced, if you opt for streaming you only need to set up the data ingestion, storing it wherever you want. An interesting alternative is to store aggregations over the streaming data for posterior analysis (e.g. average value across a time period). If these aggregates need further refinement, they can be stored in the landing zone and be later processed. The decision on how to handle the different types of data you ingest depends on your goals, and it is ultimately a design choice on your part.

Data lakes and data lakehouses

These large and heterogeneous repositories we are defining here are commonly referred to as **data lakes**. The easiest way to implement a data lake is to map it to your file system. That is, you can create some folder in your PC that will store the data coming from the ingestion mechanisms. Any management that needs to be done can be executed via Python scripts. This is not inherently distributed, so we propose a couple of alternatives:

- HDFS (Hadoop File System): you will learn about Hadoop and HDFS in the first half of the course, so you can employ it as the data lake. Note that if you run HDFS in your system, it will, logically, not be distributed. As we do not provide you with a cluster to deploy the system, this is fine.
- Cloud technologies such as Amazon S3 or Azure Data Lake Storage, which tend to provide free tiers that are enough for the scope of the course.

With the previous systems you can implement the aforementioned data lakes. However, it is worth noting that in recent years a new storage framework has emerged: the **data lakehouse**. These combine the storage philosophy of data lakes (large, heterogeneous repositories) with features designed to streamline data management and create an engineering experience similar to a data warehouse. This means robust table and schema support and features for managing incremental updates and deletes. Lakehouses typically also support table history and rollback; this is accomplished by retaining old versions of files and metadata. Hence, they are also a good option to implement the landing zone. Once again, they can support distributed data management, but, given the lack of a cluster to deploy them, these can be set up locally.

We encourage you to **use either cloud technologies or data lakehouses**, as they are more interesting and up-to-date methodologies compared to HDFS (which you still have to learn given that it represents the basis of big data storage) or a plain old data file system. If you opt for a data lakehouse, we recommend Delta Lake, which provides a lot of interesting features (ACID transactions, schema evolution, etc.) and natively connects with a lot of processing tools and databases.

Data organization

The first storage layer serves as the primary entry point for data into our system. At this stage, no processing or transformation is performed on the data, ensuring that it remains in its raw form. However, establishing a **well-structured organization** and capturing **essential metadata** are crucial steps to facilitate efficient data management and future processing. If this is not done, rather than a data lake you have a data swamp.

A simple yet effective strategy to provide some structure to the landing zone includes categorizing files based on logical attributes such as date, data source, or region. For example, if you are using a file system, you can generate several folders and store data separately based on these properties. Additionally, implementing a standardized naming convention across all files (which is a very straightforward way of defining and keeping some metadata) enhances consistency, making retrieval and integration more seamless.

Beyond these basic practices, more advanced data management techniques can be employed to improve traceability. For instance, data versioning can be implemented to track changes over time, or maintaining a comprehensive metadata catalog enables better data governance. Finally, you can even include "substages", like a *temporal landing* zone in which you initially store the raw data (with no structure) for a period of time before it is deleted and moved to a *persistent landing zone,* in which the data is further structured and kept indefinitely.

By enforcing these approaches, the first storage layer not only ensures data integrity but also lays the foundation for a scalable and well-governed data ecosystem.

Tasks

To implement the landing zone **you need** to:

1. Select an appropriate storage solution based on the nature of your data.
2. Define an organizational structure for storing raw data (e.g. different folders/buckets, naming convention, data partitioning).
3. Deploy the raw data layer and connect the ingestion tasks to it.

Factors that will **positively contribute** towards the grade:

1. Employing cloud-based systems or data lakehouses as opposed to your local file system.
2. More complex data structuring or metadata management.

2. Deliverables

Main deliverable (P1)

The tasks described in this document correspond to the **first deliverable**, which include:

- Stage 0: project context
- **Architecture design**
- Stage 1: data ingestion
- Stage 2: landing zone

To implement each stage follow the descriptions of Section 1, specifically regarding the *tasks* subsections.

Note that we explicitly ask you to define, in this first deliverable, the architecture. That is, a high level diagram to showcase the flow of data. The goal is to put yourself in the shoes of a data engineer who has to come up with an entire workflow before implementing it. Given that the trusted and exploitation zones have not yet been presented, you can define them as black boxes to be later instantiated. Nonetheless, you should already know what you want to do with the data, so you can already present the processes that will exploit the data that you manage (i.e. you have to include the data consumption stage).

An example of how this design could look is presented at the end of this document. The design you see is quite complex, but you should aim for something similar when

defining the architecture. **This is because you do not need to implement everything you present in the design**. You might create a big structure and then decide to focus on specific parts of it; those that seem more feasible. Or, during the development you might realize that you can not implement everything and just do a subset of the processes. As long as the final work is appropriate, the grade should not suffer. The goal, once more, is that from the start you have an idea of where you want to go and how to get there, laying up a sophisticated blueprint that you can later modify ad-hoc. Communication with your supervisor can help you to know what to prioritize.

Orchestration and containerization

The initial versions of the pipeline are meant to be developed asynchronously. That is, you will first set up the different environments and their respective connections separately, manually testing that the data flows adequately for each specific segment of the architecture. Nonetheless these pipelines are meant to be automated, from the data ingestion to the exploitation of the data. Hence, we ask that, once the stages have been set in place, that you automate (more precisely, orchestrate) the entire execution.

Key aspects of workflow orchestration include:

- Task scheduling: Defining when each process should run, whether on a periodic basis (e.g., daily, hourly) or triggered by specific events.
- Dependency management: Ensuring that tasks are executed in the correct order, preventing downstream tasks from running before upstream processes have completed successfully.
- Error handling and retries: Automatically detecting failures, logging errors, and retrying failed tasks as necessary to ensure robustness.

You will study Apache Airflow, an orchestrating tool, later in the course. Nonetheless, if you want to start experimenting with it, you are free to do so. Alternatively, and specially at this point in the course, you can use simple Python scripts to schedule the tasks.

Note that at this stage you have just the data ingestion to schedule, so the implementation should be straightforward. Note that, given all the other requirements we ask for, the pipeline orchestration is not mandatory, but implementing it will positively influence the grade. If you want to extend the development even further, you can also add the following notions into the architecture:

- Error handling and reporting.
- Execution monitoring (processes performance, disk and memory usage, etc.).
- Tools to control quality.

Recall that we ask for you to provide access to a repository to get the project implementation from. Hence, all the tools should be used via scripts (Python, most commonly) and libraries that can be easily downloaded. If you use a system that requires it to be deployed, we ask that you containerize it via Docker or similar technologies, so the supervisor can more easily deploy and test your pipeline.

Follow-up deliverables

There will be two follow-up deliverables in this first part, each before their corresponding follow-up sessions. This means that you will have roughly two weeks to deliver them.

First deliverable:

- Project context (first version),
- Architecture design (first version).

No implementation is needed at this point. You should have a clear idea of the domain, business value and datasets encompassing your project. This information should be specified in a first draft of the final document to deliver. Also, and employing this knowledge, present the first version of the architecture design.

The purpose of this initial document is for you to have a good understanding of what you want to implement. It does not have to be the final version, but having a first, fully-fledged instantiation of the design will allow for a smoother communication with the supervisor regarding the project.

Second deliverable:

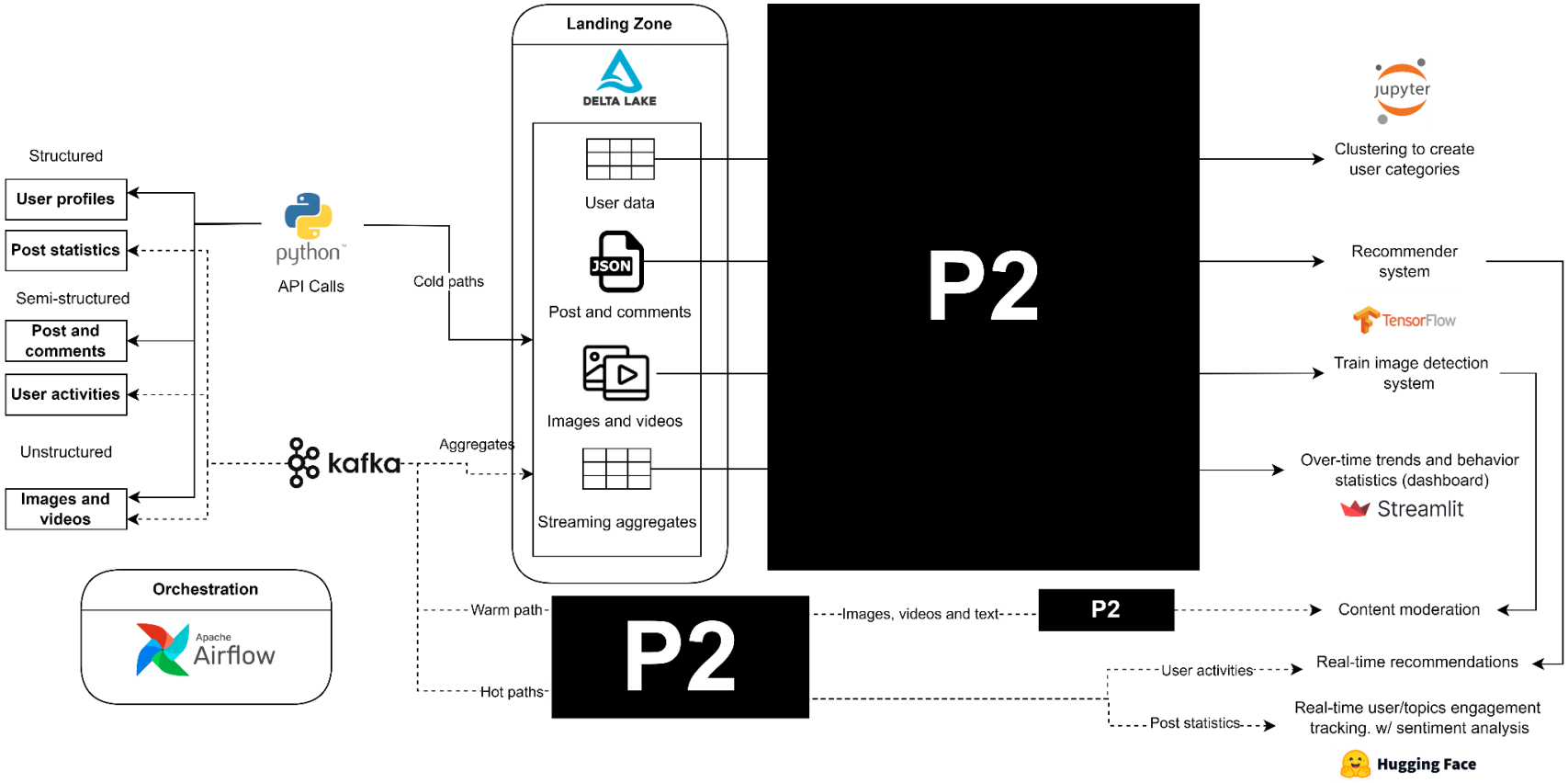
- Project context (updated).
- Architecture design (updated).
- Basic implementation of the data ingestion and the landing zone

First, there should be an updated version of the document provided in the first delivery with the ideas discussed with the supervisor. On the other hand, a link to the project's repository should be provided, with an initial implementation of the data ingestion and landing zone. High-level explanations of these two stages should also be provided in the document.

The goal here is to get familiar with the code and to start setting up the pipeline. The methodology should have been defined and some tests should have been conducted to extract the data from the sources and into the landing zone.

At this point, you will have roughly two more weeks to complete P1. The remaining time should be used to polish the document and prepare the final instantiation of the data ingestion and landing zone.

3. Pipeline design example



This design belongs to a data management architecture that performs social media analysis. Notice the literal black boxes for the aspects not introduced in this first deliverable. You can do something similar or, even better, define on a high level the tasks you plan to do. Based on the generalized descriptions of the trusted and exploitation zones, you can get an idea of what you want or have to do (e.g. necessary transformations or how to arrange the data), so you can already set these ideas in the design.

As for the example we provide, you can observe that different data types are used, different paths are defined and data is being processed in different ways, some of the end tasks even connecting with each other. This type of complex set up is the goal of this project. You can even be more detailed when defining each specific section, including any piece of information you deem relevant.