Data Engineering Design Patterns

Bartosz Konieczny

Published by O'Reilly Media, Inc.

Chapter 1. Introducing Data Engineering Design Patterns

Design patterns are well established in the software engineering space, but they have only recently begun getting traction in the data engineering world. Consequently, I owe you a few words of introduction and an explanation of what design patterns are in the context of data engineering.

What Are Design Patterns?

You may be surprised at how many times you rely on patterns in your daily life. Let's take a look at an example involving cooking and one of my favorite desserts, flan; if you like creamy desserts and haven't tried flan yet, I highly recommend it! When you want to prepare flan, you need to get all the ingredients and follow a list of preparation steps. As an outcome, you get a tasty dessert.

Why am I giving this cooking example as the introduction to a technical book about design patterns? It's because a recipe is a great representation of what a design pattern should be: a predefined and customizable template for solving a problem. How does this flan example apply to this definition?

- The ingredients and the list of preparation steps are the *predefined template*. They give you instructions but remain customizable, as you might decide to use brown sugar instead of white, for example.
- There can be a single use or many uses. The flan can be a dessert you'll share with family at teatime, or it can be a product that you'll sell to make a living. This is the *contextualization of a design pattern*. Design patterns always respond to a specific problem, which in this example is the problem of how to share a pleasant dessert with friends or how to produce the dessert to generate business revenue.
- You can decide to prepare this delicious dessert once or many times, if it happens to be your new favorite. For each new preparation, you won't reinvent the wheel. Chances are, you'll rely on the same successful recipe you tried before. That's the *reusability of the pattern*.
- But you must also be aware that preparing and eating flan has some implications for your life and health. If you prepare it every day, you'll maybe have less time for sports practice, and as a result, you might have some health issues in the long run. These are the *consequences of a pattern*.

- Finally, the recipe *saves you time* as it has been tested by many other people before. Additionally, it introduces a common dictionary that will make your life easier when discussing it with other people. Finding a recipe for flan is easier than finding one for caramel custard, which is a less popular name for flan.

Now, how does all this relate to data engineering? Again, let's use an example. You need to process a semi-structured dataset from a continuously running job. From time to time, you might be processing a record with a completely invalid format that will throw an exception and stop your job. But you don't want your whole job to fail because of that simple malformed record. This is our *contextualization*.

To solve this processing issue, you'll apply a set of best practices to your data processing logic, such as wrapping the risky transformation with a try-catch block to capture bad records and write them to another destination for analysis. That's the *predefined template*. These are the rules you can adapt to your specific use case. For example, you could decide not to send these bad records to another database and instead, simply count their occurrences.

Turns out that the example of handling erroneous records without breaking the pipeline has a specific name, *dead-lettering*. Now, if you encounter the same problem again, but in a slightly different context—maybe while working on an ELT pipeline and performing the transformations in a data warehouse directly—you can apply the same logic. That's the *reusability of the pattern*. The Dead-Letter pattern is one of the error management patterns detailed in Chapter 3.

However, you shouldn't follow the Dead-Letter pattern blindly. As with eating a flan every day, implementing the pattern has some *consequences* you should be aware of. Here, you add extra logic that adds some extra complexity to the codebase. You must be ready to accept this.

Finally, a data engineering design pattern represents a holistic picture of a solution for a given problem. It then *saves you time* and also introduces a common language that can greatly simplify discussions with your teammates or data engineers you have just met.

Yet More Design Patterns?

If you write software, you've heard about the Gang of Four's design patterns[1] and maybe even consider them as one of the clean code pillars. And now, you're probably asking yourself, aren't they enough for data engineering projects? Unfortunately, no.

Software design patterns are the recipes that you can use to keep an easily maintainable codebase. Since the patterns are standardized ways to represent a given concept, they're quickly understandable by any new person in the project.

For example, a pattern to avoid allocating unnecessary objects is *Singleton*. A newcomer who is aware of the design pattern can quickly identify it and understand its purpose in the code.

Writing maintainable code does indeed apply to data engineering projects, but it's not enough. Besides pure software aspects, you need to think about the data aspects, such as the aforementioned failure management, backfilling, idempotency, and data correctness aspects.

Common Data Engineering Patterns

The failed record management from the previous section is only one example of a data engineering design pattern. The others are part of the book, which follows a typical data flow from data ingestion to final data exposition with monitoring and alerting. Therefore, in the book you will find:

Chapter 2, "Data Ingestion Design Patterns"

Bringing data to your system will always be the first technical step in your architecture. After all, it guarantees that you have data to work on!

Chapter 3, "Error Management Design Patterns"

Errors, just like data, are an intrinsic part of data engineering. Errors may result from coding mistakes but may also come directly from data providers, who might not respect their initial engagements, for example, by sharing a dataset without required fields defined.

Chapter 4, "Idempotency Design Patterns"

A natural consequence of errors is retries that are either automatic or manual. In case of an automatic retry, part or all of your data pipeline will rerun and probably try to rewrite already saved records. If you trigger a pipeline manually, you start a backfill[2] that will execute one or more past pipelines. Thanks to idempotency, the multiple runs will generate unique outputs.

Chapter 5, "Data Value Design Patterns"

Once you're able to deal with errors and retries, you can take care of the data and generate meaningful datasets for your business users. To do so, you may need to summarize the dataset or combine it with other data sources. All of this creates extra value that is important for your end users.

Chapter 6, "Data Flow Design Patterns"

After providing a direct value to your consumers by exposing an enriched dataset, you can move to the next step and include the dataset generation as part of a data flow. The data flow defines how the pipeline that generates data value interacts with other data components in your organization.

Chapter 7, "Data Security Design Patterns", "Data Security Design Patterns"

After the first six chapters, you should know how to bring the data to your system and how to enhance it to meet your business needs. However, you must also ensure that the dataset is securely stored and that it meets data privacy requirements.

Chapter 8, "Data Storage Design Patterns"

Security is crucial, but leveraging data storage techniques to reduce the latency of processing the data is also important. That's why in this chapter you'll see how to leverage your data storage to improve the user experience.

Chapter 9, "Data Quality Design Patterns"

The bad news is that even though you have implemented all the previous chapters, your data may still be irrelevant to your consumers if you don't get rid of data quality issues, or worse, if you're not even aware of them.

Chapter 10, "Data Observability Design Patterns"

This is the last step in your journey, where you'll define various monitoring metrics that will be important to the data you work on. Alongside the data quality design patterns, the data observability design patterns help make your data trustworthy by alerting you whenever something bad is happening or is about to happen.

Case Study Used in This Book

The design patterns in this book are not tied to one specific business domain. However, understanding them without any business context would be hard, especially for less experienced readers. For that reason, you'll see each pattern introduced in the context of our case study project, which is a blog data analytics platform.

Our project follows common data practices and is divided into the layers presented in Figure 1-1.

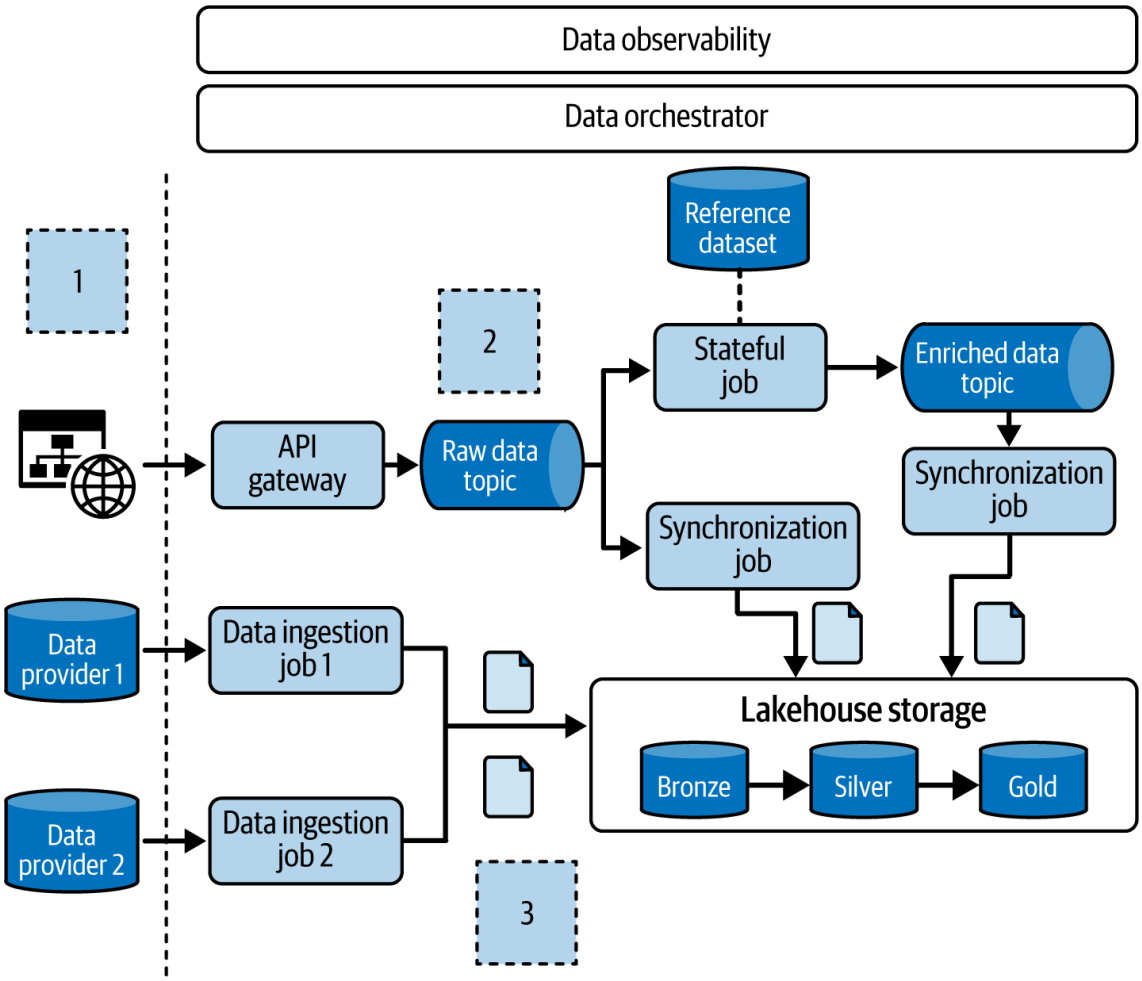

Figure 1-1. Blog analytics platform used in the case study

Figure 1-1 highlights the three most important parts of the project:

Online and offline data ingestion components

The online part applies to the data generated by users interacting with the blogs hosted on our platform. The offline part, marked in the figure as "Data provider," applies to the static external or internal datasets such as referential datasets, which are produced on a less regular schedule than the visit events (for example, once an hour).

The real-time layer

This is where you can find streaming job processing events data from a streaming broker. The jobs here may be one of two types. The first is a business-facing job that generates data for the stakeholders, such as a real-time session aggregation. The second type is a technical job that is often a technical enabler for other business use cases. An example here would be data synchronization with the data at-rest storage for ad hoc querying.

The data organization layer

This layer follows a now-common dataset structure that's based on the Medallion architecture[3] principle, in which a dataset may live in one of three different layers: Bronze, Silver, and Gold. Each layer applies to a different data maturity level. The Bronze layer stores data in its raw format, unaltered and probably with serious data quality issues. The Silver layer is responsible for cleansed and enriched datasets. Finally, the Gold layer exposes data in the format expected by the final users, such as data marts or reference datasets.

Why are these three storage layers interesting in the context of this book? Each layer represents a different data maturity level, exactly like the design patterns presented here. The patterns impacting business value will mostly expose the data in the Gold layer, while the others will remain behind, in the Bronze or Silver layer. Problem statement sections for the patterns may reference those layers to help you better understand any issues you encounter.

The schema doesn't present any implementation details on purpose. Focusing on them could shift your focus to the technology instead of the universal pattern-based solutions that are the main topic of the book. But it doesn't mean you won't see any technical details in the next chapters. On the contrary! Each pattern has a dedicated Examples section where you will see different implementations of the presented pattern.

Summary

Now that you've read this chapter, you should understand not only that flan is a great creamy dessert but also that its recipe is a great analogy for the data engineering design patterns that you will discover in the next nine chapters. I know it's a lot, but with a cup of coffee or tea and your favorite dessert (why not flan!), it'll be an exciting learning journey!

[1] The 23 software engineering design patterns introduced in *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma et al. (Addison-Wesley Professional, 1994) are colloquially known as the Gang of Four design patterns because of the book's four authors.

[2] This is known as reprocessing. So as not to confuse you, from this point forward, we'll refer to any task processing past data as *backfilling*, whether the data has already been processed or not. Technically, there is a small difference between reprocessing and backfilling, which you can learn about in the glossary available on GitHub.

[3] You can learn more about the Medallion architecture in Chapter 4 of *Delta Lake: The Definitive Guide* by Denny Lee et al. (O'Reilly, 2024).