Data Engine Thinking

FastChangeCo's journey towards a fully flexible data solution

Imprint

Data Engine Thinking
By Roelant Vos and Dirk Lerner

Published by
Vos & Lerner Data Engine Thinking GbR
Astrid-Lindgren-Straße 4
64331 Weiterstadt
Germany

https://dataenginethinking.com
info@dataenginethinking.com

Layout, typesetting, cover
Romi Klockau | linkedin.com/in/romi-klockau

Graphics and illustration
Matthias Seifert | www.matthias-seifert.com

Printing history
First edition, 2025

ISBN 978-3-9827180-0-2
Print softcover edition

ISBN 978-3-9827180-1-9
Kindle Replica-eBook edition

ISBN 978-3-9827180-2-6
PDF edition

ISBN 978-3-9827180-3-3
Print hardcover edition

ISBN 978-3-9827180-4-0
Print hardcover limited edition

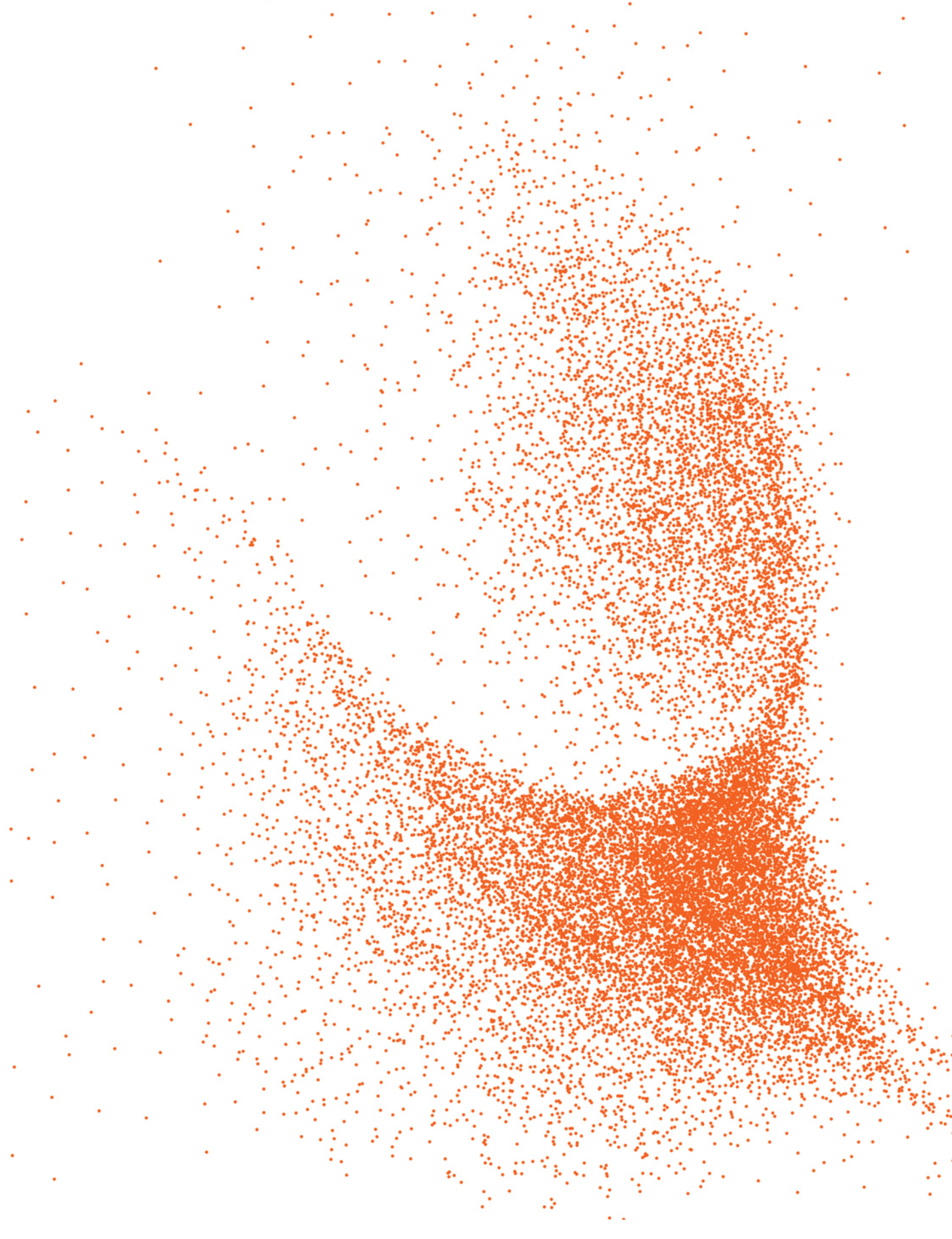
Exclusion of liability

The authors and publisher have taken great care in preparing this book, but they make no express or implied warranties regarding its content. They assume no responsibility for errors, omissions, or any incidental or consequential damages arising from the use of the information or programs contained herein.

The techniques, ideas, and recommendations in this book, including any associated code, are provided in good faith based on the authors' expertise and experiences. However, technology and individual circumstances vary, and the applicability of the provided solutions should be carefully assessed.

Readers are encouraged to use their own judgment and discretion when applying the concepts presented. While every effort has been made to ensure accuracy, the authors do not guarantee the effectiveness or suitability of these approaches for all scenarios.

The authors disclaim any liability for unintended consequences, errors, or omissions that may result from applying the book's content. By using the information in this book, readers acknowledge and accept this disclaimer, recognizing the importance of individual responsibility in its application.



02
the world of data

It's no surprise that 'data' sits at the heart of data solutions, particularly within Data Warehouse ecosystems. Data shapes the fundamental capabilities that a data solution must deliver.

To truly understand this, let's take a closer look at what data actually is and how it behaves within our systems and processes.

Fundamental assumptions when dealing with data

When working with data, it is often helpful to make assume that —at least for the purposes of designing a data solution— any initial requirement may be 'wrong.' Instead, business requirements and questions are better viewed as starting points for a deeper process of data discovery, clarification, and understanding in collaboration with subject matter experts.

This cautious approach applies not only to requirements but also to the data itself, which should be approached as potentially unreliable.

In short, when interpreting data, we assume that the questions that we're asked may be wrong, the data we have available may be unreliable, and the operational systems that captured data are inherently imperfect.

We believe that a data solution that is designed to *solely* to meet an upfront requirement is unlikely to achieve the intended business outcome.

This may sound somewhat negative, but it simply reflects a commitment to building flexibility in the data solution, so that we can adapt when the results don't align with expectations.

This flexibility enables gradual testing, learning from the available data, improving data quality, and building a knowledge base together with subject matter experts.

We assume that individuals work to the best of their abilities, and with the right intentions, but that no single person has a 'full picture' of the complete history of systems, processes, and workarounds across the entire organization.

Over time, organizations may accumulate considerable 'technical debt' from implementing workarounds that were never fully corrected. Sometimes, the original reasons for these fixes may even be lost entirely. These past decisions are still reflected in data, and are often considered 'data quality' issues.

Data teams often have limited control over the operational systems, and may not be fully across the history and evolution of these systems and their data. Fixes to data which have been applied in the past may not have been recorded, or changes were allowed through back-end access or APIs that should not have been allowed.

Sometimes, all that remains is the trace left in the data itself, and it's up to us to make sense of it.

Given this complexity, we consider it a best practice to design for repeatability in data clarification until a shared understanding is reached. And, as in scientific fields, this understanding may evolve as new evidence or perspectives emerge.

However, and this is a key guiding principle, the data itself —the events that happened and were recorded in the data solution— remains *immutable.* The data will never change. It is just our perception and understanding of them that shifts.

By embracing this, we can design data solutions that support ongoing data clarification, continuous learning, and an adaptive approach to data management.

A machine designed for change

The data solution defined in this book is fundamentally different from what traditional data solutions were perceived to be.

Because of the nature of data and the need to progressively understand it, we can't define the data solution by its reporting, Data Warehouse, Business Intelligence, or Advanced Analytics requirements alone. Rather, to be successful, the data solution needs to be able to adjust constantly to facilitate continuous clarification of these requirements.

For example, take the first iteration of a given reporting requirement. As per our fundamental principles, we will assume that both the question itself as well as the data we have to answer it are not fit for purpose. Of course, we do our best to assist the consumer in question, but always keep the possibility open that adjustments will be required.

The solution may deliver an initial data set based on what is available in the various relevant operational systems, but without any interpretation applied. When this initial report is presented to a subject matter expert, he or she is likely to respond with feedback that highlights issues and proposals for adjustments with some sort of business rule.

Thank you for your requirement!

This feedback then can be incorporated into a subsequent version — as a new interpretation on the raw data.

And so the process continues.

The dynamics of data are so varied that, without this continuous *feedback loop,* it will remain unclear if a given finding may simply be a data quality issue, or a genuine insight. The ability to reprocess the original data —the immutable events— with different interpretations, or lenses, is a key capability that we believe the data solution must have.

This can be achieved by separating the data *ingestion* from its *interpretation.*

This is an aspect of *separating concerns,* something that will be covered in-depth later.

Reprocessing of data when new understanding becomes available —supported by code-generation and automation— is a powerful way to assist organizations in their journey of improving their knowledge of the data and the involved business processes.

But, it is not only the capability for reprocessing of *data* when new clarifications emerge that is essential. Systems and infrastructure often change over time as well, both for the data solution itself and for the organization as a whole.

A successful data solution is adaptable at a technical level as well. New technologies and concepts become available all the time — the only constant is change. These may change the way data can be managed. It pays off to be adaptable and scalable to be able to make the most of these opportunities when they arise.

For example, being able to scale out to different technical architectures for the components where this makes sense. Or switching out loading technologies (i.e., Extract-Transform-Load, or ETL, platforms) for different approaches.

In these cases, it is just the technology that may change. As we will see in this book — the design, concepts and metadata will remain.

Lastly, the organization itself is not static either. Many organizations are continuously evolving, conquering new markets, introducing new products and business models — divesting or sun-setting other parts of the

business. This too has an effect on the data that is available, and the way it is used.

In a way, an organization can be seen as an organism that is continuously adapting, and a data solution should try to match the pace and agility of these changes as much as possible. This means that the intent, direction, goals and delivery of the solution may change often, and fast.

The closer the data solution can stay to this ever-moving goalpost, the more successful it is likely to be.

Many products, regardless if they are consumer goods or software products, are designed for a specific purpose and created exactly as per the design. If you design a dining table, the specifications are usually very clearly defined. You would expect the finished product to be exactly as was intended — the dining table may be returned to the store if not!

Designing a data solution following the practices explained in this book is done with a very different mindset. The data solution is not designed for a specific deliverable, it is a machine that is designed for change.

The role of a data solution

The previous sections highlighted many of the challenges in working with data and emphasized that data interpretation is often a gradu-al, iterative process, typically undertaken with subject matter experts. With each iteration, our understanding of what the data means and how it behaves becomes clearer.

When these understandings are established, there's a natural tendency to solidify them in the data solution as 'business logic,' using calculations or code to apply specific transformations. However, it can be helpful to consider how much of this complexity the data solution should absorb.

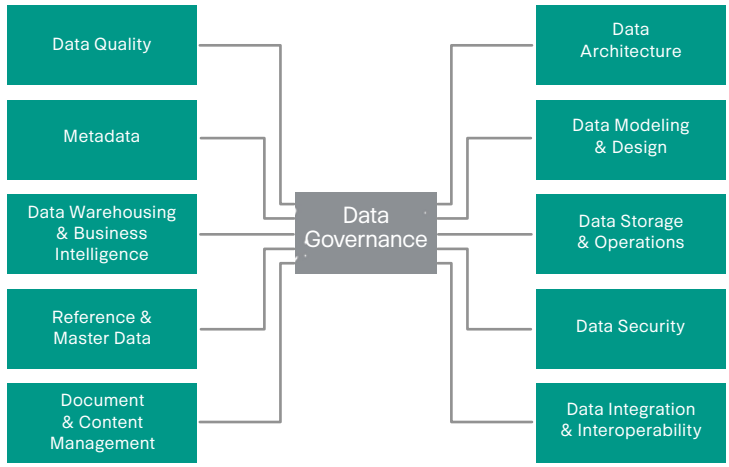
What issues can, and should, the data solution address? Where should architectural decisions be made to make data fit for its intended purpose? To what extent must the data solution implement complexities to address shortcomings in the system landscape?

These can be important considerations, but they are often influenced by specific contexts and personal preferences. Regardless, decisions must be made with a full understanding of their potential ramifications.

The Data Management Association (DAMA) offers helpful guidance here through its Data Management Body of Knowledge (DMBoK).

This reference provides various definitions and context around common terms related to 'data.' From the perspective of the data solution, it is often good to remember the context we work in, and what the relationships and roles are to related areas of expertise.

Inspired by the 'DAMA Wheel,' the illustration below shows these areas:



Here, the implementation of the data solution falls under the data governance umbrella as part of 'Data Warehousing and Business Intelligence.' This can be a sobering realization for many professionals involved in these disciplines, but it is also a very helpful one: not every data problem need to be solved by the data solution.

Consider, for example, the varying definitions of a 'customer,' 'policy,' or 'premium' in different parts of a business. Such disparities can be challenging for traditional Data Warehouse solutions, especially if the goal is to deliver a *single version of the truth.*

This objective often requires extensive upfront alignment on interpretations. These outcomes are subsequently cemented into the information model, and data must be mapped or transformed to this definition — a heavy and onerous upfront design and development effort.

When interpretations change or disagreement occurs at a later stage, the efforts to accommodate these changes can be significant.

An alternative approach is to acknowledge that different business areas may have different requirements, and that it may not be immediately needed to align or agree on interpretation. It might be better to 'agree to disagree' initially. As long as the data solution accurately represents the *facts* —the raw events as they occurred— interpretations can be adjusted iteratively over time.

Often, shared definitions for common business entities emerge only after prolonged exploration, and a process of 'guided democracy' where senior leadership, perhaps even a Chief Data Officer, strives to align all stakeholders.

Rather than placing the burden on the data solution to find a common solution, it should be able to accommodate and even report on competing interpretations, providing a basis for further discussion rather than forcing a single interpretation upfront. This way, the burden of finding a common definition shifts from the data solution to the data governance process.

It may be possible that there are even multiple definitions of information that are considered 'core' such as revenue or customer. This can be an eye-opener to: how is it possible that there are so many different views on revenue?

When agreement is reached, the updated interpretation can be reported back to the data solution and addressed in future iterations.

Thank you, again, for your business requirement.

One key advantage of this approach is that it both increases the speed to value, as well as engagement of the stakeholders. The approach can be made fully transparent.

While the goal of achieving a single version of the truth is noble and should remain part of the long-term strategy, it may not be the responsibility of the data solution to deliver this. Instead, it is better suited as an objective for a comprehensive data governance program.

This is why a fundamental design decision is that the data solution is *not* designed to be the source of truth. Rather, it is designed to be the source of facts as they occurred, without any specific interpretation.

Throughout this book, we refer to this as the *single version of the facts.*

Data represents the events created by systems when certain processes occur. It is the core purpose of the data solution to record this correctly, timely, and completely.

Resolving differences in interpretation, however, is ultimately a task for data governance.

Solutions for ongoing data interpretation

A data solution, such as a Data Warehouse, can be considered to be a necessary evil.

In an ideal world, there would be no need for it. Ideally, optimal data governance and near real-time multi-directional data harmonization would have created an environment where it is easy to consume data across systems without any ambiguity. The same applies to business logic — if everything is neatly organized in an all-encompassing data architecture, how many transformation rules would still be required?

This ideal scenario remains beyond reach for most organizations and, more broadly, for the data industry as a whole.

However, treating data solutions as mechanisms to facilitate communication can help move an organization closer to this ideal. Achieving this is yet another reason that a data solution needs to be flexible, quick to adapt, and easy to manage.

Herein lies a paradox, which will be a recurring theme.

To make a data solution sufficiently flexible, easy to manage, and able to adjust at the speed the business operates, the solution requires multiple well-defined layers, areas of design, and frameworks.

At first glance this may appear to be a complex configuration.

A paradox of complexity

The flexibility required for a data solution often introduces complexity, typically manifesting as distinct 'layers' and 'frameworks' within the solution design, each focused on implementing specific concepts.

A common question that arises is whether it's worth investing time and effort upfront in what might appear to be a complex system involving various layers and concepts.

Wouldn't it be simpler to just query the raw data directly, without the hassle of managing multiple data management and interpretation layers? How can we strike the right balance between necessary complexity and desired functionality?

It is our opinion that a necessary baseline of functionality needs to be in place to ensure efficient and effective management of a data solution — including the *refactoring* necessary to adapt to changing data interpretation.

We will refer frequently to the term 'refactoring.' In this context, refactoring involves redesigning parts of the solution to optimize it for a given scenario. It's a concept borrowed from software development, where it typically refers to improving existing code without necessarily adding new features.

Refactoring may not always yield visible changes to the end user, but it often results in code that is more flexible, maintainable, and robust. We believe that refactoring is a fitting term for the restructuring of the data solution, aimed at delivering the intended optimization outcomes.

Our goal is to define and standardize this necessary complexity upfront, so that it becomes a natural part of the solution rather than a barrier to implementation.

This foundational baseline, with its well-defined layers, design areas, and supporting frameworks, is critical for making the data solution easy to operate and maintain. It facilitates refactoring —even automated— and supports quick adaptation.

While these concepts may add complexity, they can be standardized and integrated into automated code generation and delivery. By establishing this baseline, we believe that data solutions can become a machine designed for change, capable of evolving alongside business needs.

We will explain our approach using the concept of *separating concerns,* the mindset of *data solution virtualization,* and the implementation framework of *Engine Thinking.*

Together, these three concepts form the specification of the data solution we propose.

Separation of concerns

A key concept to ensure that the inherent complexity of data solutions can be managed in a simple way is the *separation of concerns.* Separating concerns means breaking down the tasks that a data solution must perform into smaller, modular, and atomic processes, each with a defined role within the overall data and IT architecture.

This approach stands in stark contrast to traditional methods, which often feature complex, monolithic data logistics processes. These processes typically perform numerous functional steps in a single operation.

For example, typical two-layered solution designs approaches (e.g., dimensional models or Kimball-style Data Warehousing) load data into a staging area (layer 1) that resembles the target model (layer 2). Data is then merged with the already existing information.

The data logistic processes that execute this logic have to support a broad variety of functions, including but not limited to:

- → Implementing a variety of business rules
- → Integrating multiple data sources
- → Managing changes over time
- → Key distribution
- → Defining structure and hierarchies
- → Balancing performance requirements
- → Handling granularity and aggregation
- → Detecting data changes

This may result in solutions that can be difficult to extend, and tend to suffer from increasingly complex interdependencies. The resulting effort to implement changes pushes out delivery time, and creates a gap between the required functionality of the solution and the degree that these requirements have been implemented.

This can lead to critical issues and the emergence of alternative solutions that may eventually render the data solution obsolete, especially as requirements evolve over time and become harder to keep pace with.

The larger the system grows, the wider this gap tends to become.

'Hybrid[4]' modeling approaches, which combine elements of normalized and dimensional modeling, tackle these challenges by separating the fundamental housekeeping of data solutions from business logic, maximizing flexibility and ease of maintenance.

Unlike more traditional forms of dimensional modeling, where a single process often handles multiple functions, hybrid modeling techniques decompose these tasks into distinct, atomic operations.

For instance, key distribution is managed by core business concept entities representing the central business keys, and historization is embedded in context entities. Additionally, relationships between business concepts are modeled separately using natural business relationship entities.

Business rules —or the interpretation of data— are layered on top of the raw data, but only after it has been safely recorded — rather than applied when the data is loaded *into* the data solution. This approach preserves the integrity of the original data while enabling flexible interpretation.

Requirements inevitably change over time, especially during the early phases of data projects and more generally as business needs evolve. By separating the delivery of information (via business rules) from the storage and management of raw data, hybrid modeling techniques —supported by a matching solution architecture— make it easier to accommodate different perspectives and reduce the complex interdependencies often seen in traditional approaches.

Hybrid modeling techniques acknowledge that the 'truth' is subjective and may differ across consumers of data, even within the same organization. While there may not be a single version of the *truth,* there is always a single version of the *facts* — the immutable transactions that serve as a reliable foundation for delivering multiple interpretations.

This approach forms a core part of our methodology, and we will explore it in detail throughout the following chapters.

Data solution virtualization

In the data modeling community, discussions often focus on specific, technical topics closely tied to the physical implementation of data solutions.

For instance, in Data Vault methodology, questions like 'Do we still need Hubs?,' 'Should we create separate tables for each attribute?,' or 'Is this transaction better modeled as a Hub or a Link?' tend to center around physical implementation details.

While these are interesting considerations, it can be much more valuable to separate the principles and concepts from their technical implementation choices — and use these to agree on how to best create value for a business using data.

Physical data modeling concepts and methodologies are subject to evolution themselves, it's not just the business, models, and technology that evolves. Newer, sometimes better, ideas emerge regularly, and it is essential to periodically reassess and adapt existing approaches. It's also perfectly acceptable to experience a form of atavism —where certain practices resurface after a period of absence— if they provide a good solution for today's problems.

How do we keep the things that work in the current environments, and move away from what doesn't make as much sense anymore?

Inevitably, there will be some tension between adhering to prescribed standards and this evolutionary mindset. Rigidly following an implementation standard may lead to dead ends. Some ideas lose relevance, don't work as well using certain technology, or prove less effective than anticipated. Methodologies must evolve to remain practical; otherwise, they risk becoming obsolete — along with any solution that strictly follows them. However, by adopting a mindset of continuous improvement, where designs are iteratively adjusted, the likelihood of long-term success and relevance increases significantly.

This is where the concept of *data solution virtualization* comes into play.

Data solution virtualization is a mindset where changes in data models, concepts, patterns, and business logic are directly and automatically translated into the implementation and deployment of the data solution. It's refactoring taken to the extreme: when you update your model or design, the data solution adjusts itself automatically.

The *virtualization* aspect alludes to the idea that —given sufficient resources— any interpretation of data can be generated at runtime. In essence, the entire data solution could be ephemeral, recreated as needed. However, in the real world, resource constraints mean that the actual implementation may vary. This could involve recreating views, dropping and rebuilding database tables, or regenerating and executing the data logistics processes needed to repopulate tables with up-to-date data.

Regardless of the specific implementation, the outcome remains the same. Even if development and deployment follow a controlled, gated approach —which is often the case— this method allows for previewing a design before formally committing to it. It enables direct examination of the data outputs resulting from data modeling and interpretation logic, without needing to fully develop the entire solution first.

This can be particularly helpful during the early stages of development, where there typically is more volatility in the design.

A paradox

At first glance, data solution virtualization may give the impression that a physical data solution is unnecessary, which presents another paradox. If you can represent your target data model as a set of views and automatically allocate infrastructure resources for optimal runtime performance, does it still make sense to implement the solution using physical tables and corresponding data logistics processes?

The answer, in most cases, is still *yes* — at least for now. This is due to various factors, such as the available technical infrastructure, software capabilities, performance considerations, and the size of data sets.

However, the purpose of data solution virtualization is not necessarily to render traditional data logistics obsolete. Instead, it serves as a litmus test to verify that all the necessary capabilities of the data solution are in place, and properly configured.

A useful way to think about this is: if you can generate and deploy your entire data solution using queries or views based on metadata, then you can *also* use the same metadata to deploy the solution with physical tables and data logistics processes.

A core concept here is the ability to process data *deterministically.* Deterministic processes always produce the same outcome when given the same input values. In databases, deterministic queries consistently yield the same results when run against the same data set.

When your data solution is guaranteed to be deterministic, it provides flexibility in choosing *how* the results are delivered. In essence, when processes are deterministic, data solution virtualization treats the main functionality of the data solution as a commodity that can be deployed into a specific physical design and supporting data logistics. The physical implementation becomes a parameter.

Returning to the vision of data solution virtualization, what seems realistic is that many functions of the data solution will increasingly move into the background. The exact specifications of the physical model become less relevant, as these are driven (and automatically refactored) by defined optimization rules. For example, automated refactoring might eliminate the need for manual decisions about splitting Satellites due to high change rates in a few attributes. The engine might decide to create multiple physical tables for a single context of a business key—or not—depending on the data dynamics at the time.

Instead, the focus can shift towards the information model, including the definition and interpretation of data entities, with everything else derived automatically.

Data solution virtualization, as a test of fundamental capabilities, demonstrates that the data solution can evolve alongside the business, continuously bridging the gap between raw data and its consumption. The exact requirements for achieving this make up the core of the engine, which will be detailed in the Engine Thinking section.

Engine Thinking

To implement the vision of data solution virtualization, and fully embrace flexibility through automation, we can integrate the necessary components for delivering a reliable and adaptive data solution into a cohesive system capable of operating autonomously.

This capability —to treat the data solution as a dynamic communication asset while maintaining principles like layered design and separation of concerns— is what we refer to as *Engine Thinking.*

The engine concept acknowledges that many technical implementations share a common foundation in terms of their information (e.g., conceptual and logical) models and essential components. The resulting physical data model and data logistics processes can be viewed as by-products — outcomes driven by the *optimization* needs of the solution.

Consider the analogy of a database optimizer, which uses statistics and parameters to determine the most efficient way to execute a query. The engine operates in a similar manner, but at the application level. It continuously refines the physical model based on usage patterns and performance considerations, optimizing the implementation automatically in the background.

Imagine the utility of embedding these optimization decisions as parameters, letting the engine drive the creation of the most suitable physical structures. Or even better, allowing an intelligent optimizer to decide the optimal approach. If we elevate the data models (and corresponding design choices) to a higher, abstract level, many low-level physical model implementation decisions can be streamlined, or even eliminated.

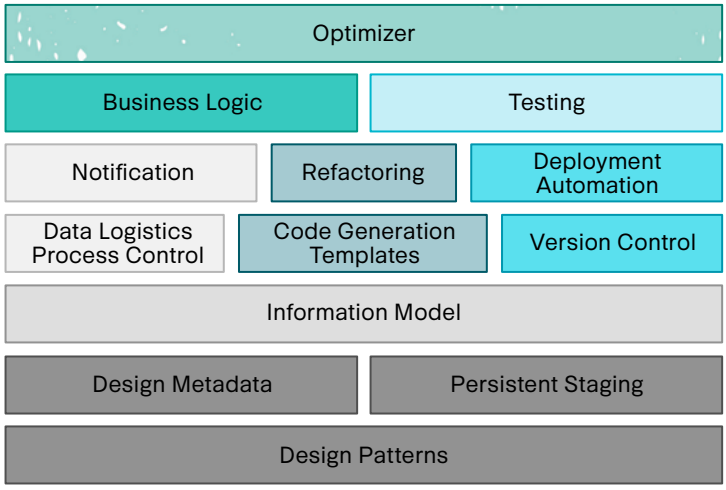
Do we really need to worry about how relationships are modeled physically when we agree on their existence at a conceptual level? Is it necessary to physically instantiate key tables if they are rarely used in queries? Could we generate them only on demand?

Does it matter to the information model if certain columns are moved into separate objects for performance reasons?

Should relationships always be represented as unique instances (e.g., core business concepts or key tables), or can they be managed as many-to-many tables? Should we select a specific hashing approach across all tables, or can we balance out necessity and performance versus risk appetite for collisions in the algorithm?

These questions, and many others like them, can be addressed through the engine concept. The key lies in connecting the components required to build the data solution effectively from the outset.

The engine combines a suite of related capabilities that facilitate a flexible approach to designing and managing data solutions. The diagram below outlines the core components of the engine. Each concept is explored in-depth in subsequent chapters.



The engine is governed by a combination of *design* metadata and *environmental* metadata that decides and manages how the physical models and corresponding data logistics processes are generated, deployed, and maintained.

The core premise of Engine Thinking is to establish a mechanism that enables (re)modeling of data, while the technical solution is automatically refactored in the background to accommodate new or revised designs. Technically, this includes capabilities like automatically redefining physical tables and reloading them with data using dynamically generated data logistics processes — and more.

Now, let's begin constructing our own engine by diving deeper into these concepts and aligning them with the right technologies and frameworks.

Design patterns

The foundation of any code generation or automation efforts is a clear understanding of what needs to be accomplished. Design patterns offer a structured framework for documenting the core concepts of the solution, including data logistics processes.

They serve as a 'what' and 'how-to' guide, detailing these concepts while remaining technology agnostic — independent of any specific implementation. Design patterns are explored in depth in the ⚐ data solution architecture sections.

For now, it's important to emphasize that any key decision made by the engine should be grounded in a well-documented design pattern. Code generation templates, in turn, act as the practical implementations of these design patterns, translating the documented concept into executable artifacts.

Design metadata

Design metadata is software and platform agnostic, and focuses on conventions and source-to-target mappings — including data definitions and transformation rules.

An example of design metadata is the definition of the core business concepts and their relationships (a convention), or a transformation to conform a specific data set to a defined target (a lineage, business rule, or source-to-target mapping).

This type of metadata is closely tied to the concept of design patterns. These essentially guide the delivery of the data solution by using design metadata as input.

Design metadata defines how many data logistics processes are expected, captures how data is transformed, and determines what data needs to go where. In the code generation chapter, we will introduce a standard format to record design metadata.

Design metadata encompasses all metadata necessary for deploying a solution, including:

- Process and data logistics details, recorded as 'mappings' or lineage information capturing where data is moved to — or interpreted from
- Storage metadata, describing objects such as tables, views, formats, domains. These can either be derived using conventions, or specified directly

Managing design metadata is the cornerstone of the engine, particularly defining the source and target models (structure metadata) and establishing source-to-target mappings (process or data logistics metadata).

Each organization has its own systems and processes, and mapping these to a data model is always tailored and specific. This is why design metadata is the true intellectual property (IP) of the data solution. It encapsulates the unique and custom interpretations of data specific to an organization.

When combined with code generation templates and guided by design patterns, design metadata drives the automated creation of data logistics processes. Design metadata defines the *what,* while the code generation templates provide the *how.* Together, they form the essential components that fuel the engine.

At this stage of the engine, a *standardized framework for recording design metadata* is in place. This repository will serve as a foundation, gradually enriched as additional components of the design are finalized — particularly the definition of the *information model.*

As the design evolves, the metadata will capture and reflect these developments.

Persistent staging

A *persistent staging area (PSA)* forms the backbone of a flexible data solution. It acts as an immutable archive, capturing all original transactions that have been presented to the data solution *before* any transformation or interpretation is applied, including data modeling.

Often referred to by other names like 'history zone,' the PSA can be broadly defined as an insert-only, time-stamped log of all transactions (events) received by the data solution and organized by arrival time. The PSA can be implemented using databases, tables, logs, streams, files, or a combination of these technologies.

Because the transactions stored in the PSA are original and immutable, the PSA enables deterministic (re)processing of data. Reprocessing simply means running the input data through updated logic, recalculating the output. This capability is essential for replaying historical transactions against an revised target data model or modified transformation rules.

The main purpose of the PSA is to store raw, unaltered transactions for potential reprocessing when design metadata changes. This includes changes to the target model itself, which may evolve as details are refined and reconsidered.

This aligns with the iterative approach of clarifying data meaning over time. Even with the best intent and skills, initial models may not fully capture all nuances, and are therefore subject to change.

The PSA provides a reliable foundation for reconsidering raw transactions from different perspectives. It allows an organization to solve the inherent overall complexity of its business incrementally.

An application-readable log

A helpful way to think about a PSA is as an event-based system — an application-readable 'log' that is similar to a transaction log in a relational database[5].

In database systems, a transaction log is an insert-only, time-ordered sequence of data events (inserts, updates, deletes). It is central to how databases work. Data events are first written to the transaction log and then propagated to (potentially) various data structures such as tables or indexes, which serve as *representations* of these events. Events can even be forwarded to other systems through techniques like log shipping or transaction log-based replication.

For a database, the log is the ultimate source of truth — the main record of all changes. It is also essential for implementing the ACID[6] principles. In a mature system, this is handled almost autonomously in the background — but it remains a fundamental mechanism for ensuring data consistency and reliability.

Pub/Sub

A PSA operates as a 'Write Once, Read Many' (WORM) system, where multiple data logistics processes can independently 'read' from the log. Each consumer can track its own processing state using pointers like inscription timestamps, which indicate the time of the event's arrival in the PSA. This way, consuming data flows, selections or streams manage their own load window and can therefore operate completely independently.

The PSA does not need to 'know' which processes use the log, or how up to date they are in their processing.

This independence aligns with the principles of the publish-subscribe (pub/sub) pattern, where producers (data sources) and consumers (data processes) operate independently without direct knowledge of each other.

This approach even makes it possible to support applications that only require to be online from time to time, and allow them to sync their data when required. Applications control the way they consume data. It is only necessary to track the event up to which processing is completed.

For the data logistics processes of the data solution, this works the same way. Essentially, every data logistics process can be independent and maintain their own state and consistency. In a sense, they are all individual consumers of the application-readable log. This supports an almost endless degree of parallelism and flexibility in data processing, something that will be explored throughout this book.

Implementing the PSA

A PSA can be implemented using a wide range of technologies. Regardless of using database tables, files, or streams, the core concept remains the same: an insert-only, time-stamped sequence of events.

In databases, this may involve physical tables, while other implementations might use formats like Parquet, Avro, Iceberg, JSON, or HDFS.

Contemporary technologies support this concept, including bridging solutions like PolyBase and distributed commit log systems such as Apache Kafka, Azure Event Hub, Amazon Kinesis, and Google Pub/Sub. These are just some of the technologies available at the time this book was written. In the future, different tools and techniques are likely to be available. However, the concept is expected to remain the same.

These tools offer scalability, high throughput, fault tolerance, and features like log retention and partitioning. They allow consuming applications to subscribe via APIs or services. This way, the PSA effectively becomes a 'stream' with potentially infinite retention (Time to Live, TTL) — an institutional memory.

We have explored the role of the PSA within the context of a data solution, where it acts as a central repository capturing transactions from various operational systems. However, the possibilities of the PSA extend beyond just supporting the data solution.

Besides supporting the myriad of automatically generated atomic data logistics processes that consume data from the PSA into the data solution, the PSA can also serve as a subscription source for any information-consuming system.

This positions the PSA as a central 'data hub' that provides the infrastructure to provide raw data change events to *any other* system that requires it — often an early win in data projects. Downstream processes can consume events independently and at their own pace, and as supported by the available technology.

Information model

The information model serves as the brain of the data solution, shaping how data flows through the system, and ultimately unambiguously describing what the data represents.

At this stage of the engine, design patterns detail how concepts need to be managed, an approach for capturing design metadata is available to start documenting what objects exist and what data needs to be mapped to them, and the PSA has begun recording transactions from the connected operational systems.

However, it has not yet been determined how all of this should be applied. This is the role of the model. It is the model that defines what the data should look like at every stage of the solution.

Levels of abstraction

There is a distinction to make between different levels of modeling abstraction. Without going into too much detail, these can be classified as more conceptual, more logical, or more physical.

Conceptual models focus on high-level representations of terms and concepts, their definition, and their interactions. A logical level is more focused on clearly-specified entities, their types, attributes, and relationships. The physical model covers the technical implementation tailored to a specific database or technology.

The relationships between these different levels of abstraction can be recorded in design metadata — for example as a mapping between a logical model entity and a physical model table. These mappings can then be used to generate physical model structures using code generation templates. The physical model is often convention-based and can be derived from its more abstract logical version.

In the context of the engine, this distinction is important. We often experience discussions that focus on differences of opinion at the physical level (e.g., table structures, specific columns), when there is sufficient agreement at the logical level.

Using the engine, the focus is placed more on the logical level and above — the physical level can be automatically generated, and is subject to improvements as determined by the optimizer and its directives.

We will predominantly use the term 'information model' (or simply 'model'), particularly when contrasting certain design decisions with the physical model. We'll leave the exact level of abstraction intentionally flexible, as this is subject to different opinions and does not materially affect how the engine operates.

Developers can choose to work directly with physical models, apply these to design metadata, and generate the solution accordingly. Alternatively, they can define a more abstract, business driven representation —what we refer to as the information model— and use conventions to derive a physical version.

Mapping data to the model

An important piece of design metadata to capture is the relationship between data sources and model objects. What data needs to go where? This relationship often becomes apparent during the data modeling efforts, but can also be added later. This key piece of information is required to generate the data logistics processes that populate the model.

When the model is complete, it effectively has become part of the design metadata. In other words, the design metadata contains a representation of the model.

When the model changes, these changes must be reflected in the design metadata as well, so that the solution can be automatically adjusted to reflect this updated definition of data.

Code generation templates

Code generation automates the production of data solution components.

To generate the necessary code, both code generation templates and design metadata are required as inputs. Together, these inputs can be compiled into executable data logistics processes or object artifacts tailored to a specific platform or environment.

A design pattern defines the goals and outcomes for a concept or component, and the code generation template specifies the corresponding technical implementation. Templates are typically described using a *Domain Specific Language,* making them interpretable by a compiler or runtime engine.

The outputs of this process can include programming code, SQL scripts, or proprietary objects for various data technologies. They cover all aspects of the data solution, including data objects (tables, views, files), data logistics processes, scripts, and even infrastructure and connectivity.

We refer to this as *model-driven* code generation. The model, as encapsulated in the design metadata, drives the output based on the selected code generation templates.

The challenges of manual development

Historically, developing data logistics processes has been a manual and resource-intensive task. This made it one of the most time-consuming aspects of data solution development. The overhead caused by manual (re)development, maintenance, and testing of the many involved data logistics processes has traditionally been a major barrier for refactoring.

Pattern-based approaches, combined with code generation, can mitigate many of these issues.

Manual development also often imposed restrictions on developing truly scalable data solutions. For instance, in modeling techniques such as Data Vault or Anchor Modeling, each data set associated with a context table and its corresponding core business concept table typically requires its own independent data logistics process. These processes, while following the same pattern, are unique and may number in the hundreds or thousands.

With manual development, the trade-off between creating potentially hundreds of seemingly redundant data logistics processes for long term scalability and short term delivery would usually favor the latter. However, code generation shifts this paradigm. By automating the production of these repetitive processes, code generation enables the scalability and high degree of parallelism in data processing that manual approaches struggle to achieve.

Following evolution

Defining a collection of well-structured design patterns and code generation templates to deliver the data solution is essential for achieving flexibility in delivery. Patterns and templates evolve over time, and this can sometimes lead to a desire to refactoring part, or even the entire data solution.

Investing in code generation supports a continuous evolution of the data solution. It allows you to easily incorporate and test new ideas and improvements.

This reduces the time to value because the output data logistics processes can be delivered faster, and more consistently. Code generation also helps to dramatically reduce technical debt when tweaks to the patterns are applied, since updates can automatically propagate throughout the solution.

It is a best practice to ensure the *entire* solution can be generated to effectively combat the inevitable accumulation of technical debt.

Lastly, one of the most important benefits is the consistency in delivery both in terms of time and quality. In our experience, businesses value consistent quality and reliable delivery timelines over speed with variability (though speed is still important).

By automating the generation of data logistics processes, teams can ensure predictable results while maintaining flexibility and short delivery timelines.

Data logistics process control

A data logistics process control framework (or simply, 'control framework') is a structured set of procedures designed to govern the execution, orchestration, monitoring, and logging of individual data processing and integration tasks.

A robust control framework is an essential requirement of any data solution, and intends to:

- → Orchestrate data logistics process executions
- → Provide logging and audit capabilities
- → Simplify the management of the data solution
- → Enforce application-level transaction control (based on ACID principles)
- → Enable recovery and restart in the event of failures

This section outlines the core requirements of a control framework. Further implementation details are covered in the ⚑ data logistics control framework section.

Orchestration of process execution

In some scenarios, it may be necessary to define dependencies between processes. Some processes have to run before others. While the approach outlined in this book aims to minimize dependencies where possible, you may find cases where dependencies between data logistics processes are still required. This may be due to interface wait-states (e.g., waiting for call-response), performance reasons, interdependent business rules, or specific pattern design (e.g., key lookups).

For example, a separate delta-detection process might be required to capture data changes in a staging area before subsequent transformations can proceed.

Beyond managing execution order, orchestration and understanding dependencies can also be used to detect and prevent issues such as race conditions, cache staleness, and referential integrity violations.

A good example is populating core business concept (key) tables in a parallel environment. Potentially many data integration processes can insert new keys in these central tables, but if two or more processes attempt to insert the same key at the same time a constraint violation may occur. A smart control framework can temporarily suspend conflicting processes such as these.

Logging and audit capabilities

Regardless of whether the solution delivers data via views, functions, scripts or materialized objects (e.g., tables, indexed views, files), the control framework must record all activity for transparency and accountability.

The control framework tracks every unique process execution in a log or repository, and issues a unique execution instance identifier. The data, processed via a data logistics process execution, will be stored with this unique identifier so that it is always traceable which data was handled by which process, and when.

This attribute is called the *Audit Trail Id.*

Using the Audit Trail Id, all materialized data is 'tagged' with a pointer corresponding to the unique process execution that inserted or modified the records.

For example, if a data logistics process inserts 100 records into a target table, the control framework logs the start and end times of this unique process execution, records the number of rows processed, and assigns a unique Audit Trail Id to those 100 records. As a result, the 100 records that have been inserted will all have the same Audit Trail Id.

In virtualized environments, the control framework logs view and function executions, recording the user who issued the query and the exact version of the object accessed. This enables precise auditing and tracking of all interactions with the data solution.

Application-level transaction control

This 'link' between the control framework and the data will also become important in preparing the data for downstream processing, including delivering data for consumption.

At various stages, it will be necessary to assert which data is consistent and available for next steps. Since the control framework 'knows' what data is still being processed and what data is ready for further use, it supports the implementation of transaction control at the 'application level.'

This is similar to the ACID principles as mentioned in the previous PSA section. ACID encompasses a set of properties of database transactions intended to guarantee validity in events such as errors, outages and power failures. The control framework is used to apply similar concepts across the data solution.

This can be used to prevent 'dirty reads' from the solution, and to implement associated locking strategies for data integration. Dirty reads occur when data can be accessed that has not been fully committed by the solution — when data logistics processes have not yet completed successfully.

Another use case is the ability to report on data latency, providing insights on data freshness — an important metric for managing data solutions.

Simplifying data solution management

Ideally, a process control framework is designed to be *idempotent.* In this context this means that the system remembers which tasks were run successfully, and re-runs only the failed tasks. To support this, the control framework captures the state of a process; whether it is running or completed, and if failures were encountered that require attention or can be reprocessed automatically.

For example, imagine a workflow running five tasks in a certain order. When a failure is encountered while running the fourth task, both the failing process as well as the workflow will report failure. Upon rerun, the first three tasks —which succeeded previously— can be skipped. The fourth task will then retry and, if successful, the fifth task will be executed.

This mechanic helps in simplifying day-to-day management and monitoring, but is also important for protecting the consistency of the overall solution. For example, if the first process in the above workflow detects and loads change data delta (differential) from an operational system using a truncate & load pattern, then this data delta must be *fully committed* to all targets before it can be rerun. Otherwise, the data delta might be lost forever.

Version control

With the fundamental components now in place, the engine is able to run on its own. However, in order to adjust independently, additional 'operations[7]' components should to be added — the first of which is versioning.

Versioning is the practice of tracking and saving changes made to solution artifacts. If you save something, the previous versions should be retrievable.

There are many versioning tools, plug-ins, and concepts available, and they should be used in the context of release management — where a defined group of changes can be grouped, tested and deployed as a 'release.'

With the engine concept, we take versioning a step further.

In the engine, versioning goes beyond standard version control for individual artifacts. Instead, it manages snapshots across the design metadata including the information model, the code generation templates *and* the data itself as an integrated unit.

This holistic approach captures the state of the entire data solution —including data integration logic and data— at any given point in time.

Engine versioning is based on two core premises:

→ We can version all our design metadata and code generation templates together
→ We can generate all our code, and rebuild the entire data delivery using the PSA concept

If both are true, then we can always (re)deploy the entire solution as it existed at a particular point in time.

In many cases, it is not even necessary to version-control the *outputs* of the code generation templates. After all, artifacts like data logistics processes can always be re-created as they were in a given version. Versioning only the design metadata and code generation templates suffices.

This capability even allows the solution to host multiple versions simultaneously. These versions can be compared, tested, and optimized to determine which performs best or meets current requirements most effectively.

This approach transforms the data solution into a time-machine, enabling not just point-in-time restoration of the solution's structure and logic, but also facilitating dynamic experimentation and validation across different solution versions. Unlike traditional bitemporal data systems, this versioning component applies to the *entire* solution, offering unparalleled flexibility in managing both design and operations.

Refactoring

Imagine you start capturing transactions (events, records) early in the development process using a PSA, but finalize your overall data solution design and model some time later.

In such cases, the ability to 'replay' these transactions *as if they were processed at the time of their initial capture* would be incredibly valuable.

After all, these transactions have already existed in the PSA for some time and represent the 'transaction log' of what happened.

Loading this historical data into the data solution requires that the involved data logistics processes have the capability to process data *deterministically.* A deterministic process is one that, when executed, consistently produces the same result given the same input values.

For a data solution, this means processing the same raw data will always yield the same result in the target table. Adding this capability introduces some complexity to the pattern. However, while the pattern becomes more complex, this is offset by the ability to generate the code automatically.

Having a PSA is essential to do this. It drives the ability to *refactor* the solution in a deterministic way — a controlled process of restructuring existing code or design.

Re-initialization

The ability to replay history is referred to *re-initialization.* This involves truncating parts of the model and reloading the corresponding data from the PSA. If the process is deterministic and nothing has changed, the results will be identical to the original output. If the code or model *has* been modified, the historical transactions will be applied to the new version. With proper version control in place, it is even possible to revert to an earlier state and reproduce the data exactly as it was.

To support re-initialization, data logistics processes must be able to handle multiple data changes in a single processing pass. This is one of the fundamental principles of data logistics that underpin the data solution.

The ability to refactor and re-initialize is essential for iterative development. Solution designers can refine models and definitions over time, knowing they can refactor and reload the environment, or parts thereof, as needed.

Following the mindset of this book, change is inevitable and must be anticipated. Over time, we are likely to find flaws in our approach, our patterns, our understanding of a given technology and let's face it — our models and definitions of business terms as well. This is simply human nature, and goes back to the fundamental assumptions when dealing with data. Over time, our understanding of this complex matter will increase, and imperfections will be addressed while further developing and testing the system.

Embracing refactoring, supported by re-initialization, means you can afford some flexibility while designing use cases. You always have the option to change your mind and refactor the design whenever it makes sense to do so.

The myth of a perfect model

Some argue that refactoring simply means you didn't get it right the first time. There seems to be a deeply rooted mindset in the data community that a data model should be 100 % correct after the initial design phase. Indeed, many data solution architectures rely on this, and choose not to have fallback mechanisms —like a PSA— in place.

In reality, interpretations of data change often and few, if any, data solutions are perfect from the start. Have you ever encountered a data solution that was 100 % correct on the first attempt, with the perfect data model and interpretations? In hindsight, haven't you looked at past models and realized a different approach would have been better?

In some cases, a data solution can be refactored even after interpretations have been applied, but this can be complex and cumbersome. In other cases, refactoring may be impossible if the required original data is not available anymore. This can happen when calculations or aggregations have transformed the original values into a new data element, but one that cannot be reversed into the original values — a 'destructive' transformation.

Contemporary modeling techniques aim to delay business logic application until later in the architecture, after raw data integration and closer to the consumption of the data. These techniques advocate that, while it takes time to deliver the single version of the truth, there is at least the notion of the 'single version of the facts.' Pushing business logic to delivery layers allows iterative exploration and refinement of requirements.

However, even the core model and its corresponding data integration patterns can contain design flaws. For instance, a raw Data Vault model might have decisions around business concepts or business keys that, in hindsight, could have been better. Modeling data requires making interpretative decisions at every stage. This is why a PSA serves as the ultimate safety net.

In essence, downstream layers of the data solution become a form of schema-on-read applied to the raw data in the PSA. While data can still be persisted in various layers, the tools now in place allow teams to evolve their thinking and adapt designs as the organization grows and changes.

Shifting the mindset

This shift requires modelers to embrace the idea that mistakes will happen, and designing for change is better than attempting to achieve a perfect model upfront. The reality is that in every business there is diverging and often limited understanding of what data means, and it 'is a process' to get clarity and understanding how data should be accurately represented in models.

This perspective applies to design patterns and code generation templates as well. Models, concepts, technologies, and *even methodologies* evolve. Based on our own experience, we can say that even after working for decades in the field, we still find the occasional bug or encounter progressive thinking that would make us want to reload the environment in a slightly modified version.

In our opinion, refactoring is not a failure but an acknowledgment of progress — a way to adapt to new insights and evolving needs.

At this stage in the engine, the foundation is set for tweaking models, design patterns, or code generation templates to deliver updated versions of the data solution — even at runtime.

Notification

In addition to the data logistics process control framework, a monitoring framework is essential for *pro-actively* informing developers and support teams about the integrity of the system. Building a data solution requires to create trust in the available data for its users, and a robust monitoring framework is a powerful way to *do* so.

Functional and technical errors such as missing data, duplicates, large delays etc., will erode trust in the system and hinder its adoption and overall effectiveness. This problem becomes worse when users themselves have to identify and report these issues.

A monitoring framework assists in preventing many of these issues from happening, and also fosters trust when issues are pro-actively investigated when they do occur.

The framework involves detection mechanisms to flag specific behavior based on predefined rules. When issues are detected, the notification feature delivers these results to users and administrators for timely awareness and actioning.

The monitoring framework consists of a number of exception checks on the solution, as defined in the testing framework. For example, asserting referential integrity or consistency for logical groups of data.

The data logistics control framework also plays a central role in enabling this functionality. It schedules and executes the tests and processes responsible for monitoring the system.

Monitoring tasks

Monitoring tasks are designed as standalone executables that can run independently, either manually or via the data logistics control framework. These tasks:

- → Collect outcomes from various test cases
- → Report on the health and quality of the environment
- → Provide proactive insights into the solution's state

Monitoring outcomes can also feed back into the engine as *environmental metadata.* These results can direct the engine to refactor parts of the solution automatically, based on predefined directives. For example, system information such as CPU usage, memory pressure and disk use can inform the optimizer to adjust settings or processes dynamically.

Notification rules, as part of the monitoring tasks, can address a wide range of scenarios related to data solution integrity and performance, including but not limited to:

Adherence to conventions

- → Are table names consistently prefixed or suffixed according to agreed conventions?
- → Do core business concept tables include key attributes but exclude context attributes?
- → Are all names in lower case?
- → Are file names appropriately structured, such as including a timestamp indicator?

Infrastructure and environment information

- → Do tables have the correct compression settings?
- → Is index fragmentation exceeding acceptable thresholds?
- → Are query wait states increasing?
- → Does the I/O subsystem work as expected? Is disk space, latency, or read/write performance within expected parameters?

Latency and availability

- → How long does it take to complete one full refresh cycle across the data solution, where all involved processes have run at least once?
- → What is the current latency between receiving data deltas and making them available for reporting?
- → Are there any data logistics processes defined but inactive for over a month?

Data consistency

→ To what point in time can referential integrity be assured in a continuously loading environment?
→ Are there any orphan tables?
→ In case of 1:M relationships, does a relationship really change or was it a case of 'ghost hunting?'
→ In case of N:M relationships, is this valid, or should it be a 1:M?

Data platform optimization

→ Are there full row duplicate records across the system?
→ Are certain areas experiencing a 'flip-flopping' effect (e.g., repeated inserts and logical deletes)?

Any functional checks on data content

→ Does every invoice have an associated customer?
→ Are daily sales figures within 10 % of yesterday's values at the same time?

An effective approach for notification is establishing a common schema, and publishing events to a centrally accessible location. Examples of these could be the data logistics control framework event log, a database table exposed via a web page, dashboards, Slack, MS Teams, or Kafka topics.

Interested parties can then subscribe to these events, and treat these notifications with high priority.

Deployment automation

At this stage, the engine can confidently deliver specific versions of the data solution, including change or releases, ensuring that the entire solution can be generated and processed.

The next step is managing these releases in a deployment operations framework, ultimately aimed at achieving continuous deployment. By incorporating a workflow that is able to operate autonomously, the solution's day-to-day deployment can be handed over to the engine, reducing human error.

A typical workflow might look like this:

→ Commit changes to a central repository, using a feature branch
→ Build and test the changes in a development environment
→ Initiate a deployment to a pre-production or integration environment to detect any unforeseen conflicts
→ Release to production once validations are complete

This process can range from manual to semi-automated (e.g., trigger by commits) or fully automated (optimized by the engine). Implementation details are covered in the later sections on ⚑automating deployment.

Continuous deployment

So far, this is a fairly straightforward 'DevOps' approach. But using the engine and its available metadata, we can further automate the development and refactoring efforts.

The design metadata 'knows' which upstream tables are impacted by changes. With this information, the engine is able to automatically generate the necessary code, update or truncate associated target tables, and adapt the solution accordingly.

This requires a robust, automated deployment mechanism that can perform the following actions:

→ Deploy physical model changes
→ Generate and execute data logistics code
→ Run test cases, log results and notify interested parties of exceptions
→ Perform rollback in case of failure

The engine can trigger these 'builds' based on certain events. This can be a commit to a specific feature or release branch, or as a result of rules captured in the ⚑optimizer. For instance, the monitoring framework may detect a high demand for certain datasets and refactor the code to optimize against these usage patterns.

'Ops' in the engine concept

Deployment automation aims to shorten the delivery cycle and produce higher quality results. But it is also intended to foster collaboration between data professionals and consumers. Most importantly, it intends to facilitate communication between involved parties about the process of data design, integration and delivery.

It is about understanding deployment practices and see how these can be applied to data, for example:

- → Finding a suitable deployment frequency that meets business expectations
- → Involving stakeholders in the release process, by defining tests, performing post-implementation reviews and monitoring the outputs over time
- → Enabling stakeholders to contribute to the information model and corresponding design metadata, supported by versioning
- → Discuss directives for ongoing optimization of the data solution, for instance analyzing usage patterns to spot new opportunities and support decision making on prioritization

An automated deployment mindset helps engaging stakeholders early in data design, involving process owners and operational systems administrator to work towards a full end-to-end data management approach that is running smoothly.

For example, you could define two alternative designs for a subject area to see what works best. You could prototype different approaches for surrogate key distribution, and evaluate what works best in a given technical environment or concept. Or, you could consider table elimination from the model based on usage patterns. It is also possible to assert what physical delivery of your data solution has a better outcome for compute cost or I/O.

The engine can then automatically generate and deploy these improvements.

By embedding deployment automation within the engine, the data solution becomes a living, evolving system capable of adapting to new requirements. The engine not only streamlines deployment but also supports iterative experimentation, enabling teams to explore alternative approaches and optimize for performance, cost, or specific business needs.

Testing

Traditional data solutions often adopt the principle of *judging data on the way in,* requiring data to meet strict quality standards before being accepted. This contrasts sharply with the philosophy of this book.

In our vision, all data is welcome. We don't judge. In fact, we go to great lengths to make sure *all* data has a place, including data that can be considered 'bad' quality. We may not have the right context (at the right time) to make an upfront call on what is considered 'bad' and 'good' for the consumers of the data.

And, what may be 'bad' for one consumer may be 'good enough' for another. In exceptional cases it may exactly be the 'bad' data that turns out to hold golden nuggets of value.

This is why we separate the collection of raw facts (e.g., via the PSA and the back room concept) from their interpretation, and it's also where testing comes in. Testing, especially related to data validation and asserting if the data conforms to specific requirements, is a form of business logic — an interpretation of data.

However, the testing mechanism in the engine does *not* judge data and then bars it from entering the solution. Rather, it provides a framework to monitor and understand the state of the data across multiple perspectives.

The role of testing in the engine

Testing in the engine serves two primary purposes:

1) Tests capture the knowledge gained during design and implementation, and store this in a shared repository so that this can be reused during unit- and regression testing

2) Tests are reused as ongoing controls to verify that the data solution continues to behave as expected in the monitoring framework

Both purposes share in an important feature; a central repository to which tests can be added. Test can be developed during the development process as unit tests or in response to issues, and act as a permanent record of requirements and expectations.

As a guideline, a test should be created every time an exception has been encountered to make sure this specific scenario can be monitored in the future. Developing the test also ensures that you properly understand the violation in requirements encountered.

In this sense, the terms 'test' and 'control' are used interchangeably. While testing is used more in a unit-testing context, and typically focuses more on correct business interpretation, the same artifact (test case) is used as a control (check) to ensure ongoing consistency of the data solution.

There is no hard and fast rule on what should be tested, but it pays off to embed any understanding accumulated during development in the testing framework. At design and development time, a lot of business understanding is gained in a relatively short amount of time, and capturing this knowledge in a test case that outlines the expected behavior is a powerful way to embed this knowledge in the solution. This aligns with Gojko Adzic's 'Specification by Example' (2011), where examples are used to define and validate system behavior.

Tests can be developed for a wide variety of scenarios, including but not limited to:

→ Validating data against domain values
→ Detecting uniqueness constraint violations
→ Ensuring completeness of timelines in temporal data
→ Identifying outliers in sales or volume data using statistical thresholds (e.g., two standard deviations from the mean)
→ Verifying referential integrity

Testing framework

A testing framework, of which many are available either as open source or as part of commercial software, at a minimum has the following functionality:

→ A repository to store test cases
→ A standardize format for writing test cases
→ A set of evaluation functions (e.g., assertions, range checks, binary checks)
→ A mechanism execute tests and display the results

Simply put, a functional testing framework allows one or more tests to be executed and the results of the performed tests to be returned. This allows tests to integrate with deployment automation for regression testing and with the monitoring framework.

The testing framework can be combined with the event log of the data logistics process control framework, providing a unified monitoring point for the entire data solution.

Labeling data

Validating data is not the same as rejecting it. So how then should we act when the tests inform us of issues?

The solution is to 'label' the data that does not pass certain tests. This allows the data solution to manage data both in the incoming layers ('back room' in the data solution design) as well as the delivery layers (interpretation, 'front room') without physically filtering or removing the data.

Optimizing using environmental metadata

As the system operates, *environmental metadata* is created. While design metadata drives the logical structure of the solution, environmental metadata records system performance, such as available resources, processes execution times, memory and disk space.

Environmental metadata functions like a 'sensor,' monitoring system performance and apply this to determine the optimal physical implementation for a given directive or use case.

This is the final component of the engine, the *optimizer*.

Through the optimizer, the engine is made aware of technological constraints and directives (parameters). By interpreting the environmental metadata, and taking into account the directives, the engine can automatically refactor data structures and data logistics processes to make the most efficient use of the available technical environment.

Examples of this are:

→ Normalize or denormalize data structures
→ Select optimal aggregation strategies
→ Choose the best key distribution technique

For example, the engine might assess whether hash keys, sequence values, or natural business keys are best fit for a Data Vault implementation. Depending on the data profile, hash keys can be costly to store and retrieve. A typical hash key quickly requires 16 or 20 bytes storage per key, whereas integer sequence keys typically only require 4 or 8 bytes.

However, sequence keys introduce dependencies which may impact overall data delivery requirements. A 'middle' option of using natural business keys can also be considered. This might require less storage space and does not incur processing dependencies.

Based on environmental metadata, the engine can dynamically select or even combine these methods for optimal performance.

Unless specific overrides are in place, decisions are driven by data profiles and optimization directives (e.g., cost, compute, storage) rather than subjective preferences.

This is possible by interpreting the statistics and execution times from the data logistics control framework, as well as outputs received from the monitoring framework. With this, the code can be re-generated using a different template, and the updated solution can be automatically deployed and re-initialized.

This process modifies the physical delivery while maintaining consistency in the design metadata — automated refactoring.

Directives

The optimizer applies rules, *directives,* to determine how the data solution should operate. These optimization goals can be set by the administrator, and may include:

→ Reducing compute or storage costs
→ Achieving specific data latency thresholds (data freshness, availability)
→ Achieving specific data latency thresholds
→ Balancing I/O, storage, or compute utilization
→ Improving query performance for selected domains

The optimization process can be a fun and engaging way to manage the data solution.

Different implementation approaches can be simulated and compared (e.g., hash keys versus sequence keys) to determine the most effective approach for the environment. The optimization outputs also provide transparency that can be used to manage the environment — for the data team as well as the consumers of the data.

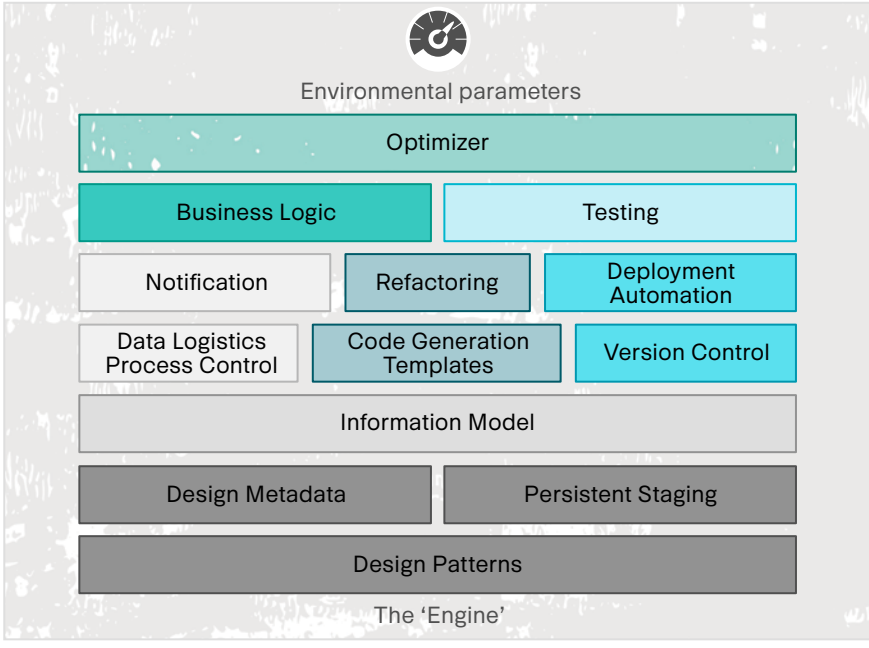
After all, delivering data solutions often involves a careful balance between resource (cost) constraints and business requirements.

Finite resources can be applied for specific outcomes, but it also makes the point that the sky is not always the limit, and that restricting certain resources will have an impact on important data solution metrics, such as latency.

Starting the engine

When the engine components are fully configured, the data solution becomes an active system capable of rapid, reliable delivery. While information models and project plans still need thoughtful design, we know that the engine can deliver fast and consistent results and supports our journey in uncovering the meaning behind the data.

From the moment the first data is processed, we can start understanding the effects the patterns have on the data. We can immediately see the initial results of our modeling decisions. Environmental metadata begins flowing in, allowing the optimizer to suggest or trigger refinements.



With the engine, data professionals no longer need to focus on low-level physical data model decisions. Instead, methodologies and best practices are embedded into the engine as conventions, freeing teams to focus on higher-level goals such as innovation, collaboration, and alignment with business objectives.

Data Engine Thinking — taking the next step

Ready to put Data Engine Thinking into action?

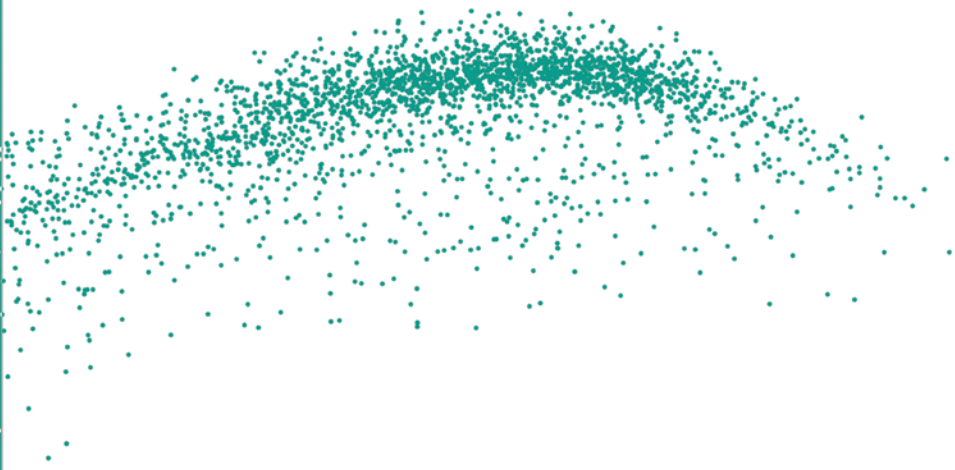
You've explored the principles—now it's time to bring them to life in your organization. Whether you're just getting started or ready to scale, we're here to help you go further, faster.

As your journey continues, we invite you to connect on https://dataenginethinking.com.

Our website has the latest information about the services we provide, including:

→ Our training schedule
→ Our coaching opportunities, including individual and team-based coaching
→ Talent mentoring
→ Assessments and reviews
→ Consultancy

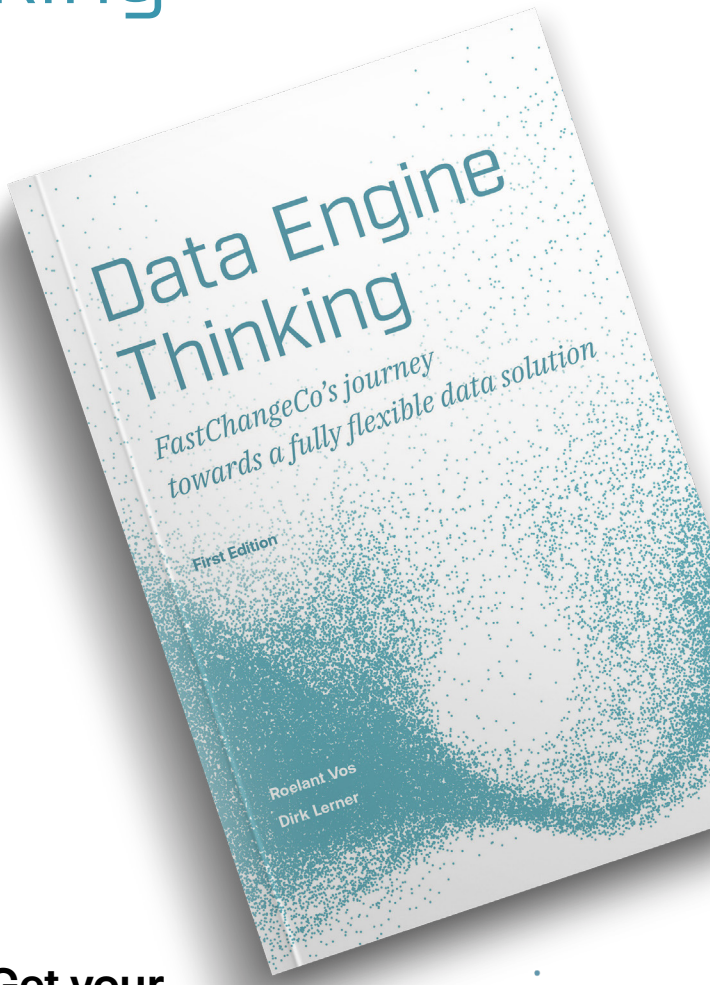
Let's unlock the full potential of your data together.



Data Engine Thinking

dataenginethinking.com

Data Engine Thinking covers the end-to-end methodology to deliver a data solution that is truly designed to adapt to progressive understanding - and ultimately meet the business‘ needs.



- Design and implement a solution that is designed for change
- Solve real-world problems encountered when working with data
- Fully automate your delivery

Get your copy here

Roelant Vos & Dirk Lerner

The Authors

Having worked as a consultant, trainer, software vendor, and decision maker in the corporate world over the years, **Roelant** has observed data management from many different points of view.

roelantvos.com

Dirk is an experienced independent consultant and managing director of TEDAMOH. He is considered a global expert on BI architectures, data modeling and temporal data.

dirklerner.com