



**Università
di Genova**

DIBRIS DIPARTIMENTO
DI INFORMATICA, BIOINGEGNERIA,
ROBOTICA E INGEGNERIA DEI SISTEMI

A Catalog of Reusable Conceptual Resources for Machine Learning-Fueled Systems

Zeinab Gholampour Dadaei

Master Thesis

Università di Genova, DIBRIS Via Opera Pia, 13 16145 Genova, Italy
<https://www.dibris.unige.it/>



MSc Computer Science
Software Security and Engineering Curriculum

A Catalog of Reusable Conceptual Resources for Machine Learning-Fueled Systems

Zeinab Gholampour Dadaei

Advisor: Prof. Gianna Reggio

Examiner: Prof. Filippo Ricca

September, 2026

Abstract Machine learning-fueled systems combine traditional software engineering concerns with concerns that are specific to data and models. The knowledge needed to address these concerns is available as reusable conceptual resources, such as design patterns, best practices, guidelines, anti-patterns, and data smells. This knowledge is spread over many scientific and grey literature sources, which present it with different and inconsistent types, and this makes it difficult to find and to reuse.

This thesis contributes to a catalog of reusable conceptual resources for the development of ML(AI)-fueled systems. The resources were collected from scientific and grey literature and through an LLM-assisted search. Their analysis showed that the classification used in the sources is not stable: more than one third of the resources received at least two different types, and new types specific to machine learning were introduced. The Catalog therefore classifies the resources according to the role they play, with five kinds (Solve, Do, Don't, Consider, and Be aware), a template for each kind, and tags for the supported activities and the affected qualities.

The thesis compiles the data engineering part of the Catalog, which contains 47 resources concerning the quality, the preparation, and the management of the data used by ML(AI)-fueled systems. It also designs the evaluation of this part: a set of evaluation dimensions and a questionnaire covering all its resources were prepared and submitted to domain experts. The execution of the evaluation was delayed, and only the answers of a first round could be collected and analysed. They already indicate what has to be revised, in particular the obligation expressed by the Do and Don't kinds and the way their resources are named.

Table of Contents

Chapter 1	Introduction	9
Chapter 2	Background	12
2.1	Types of Conceptual Reusable Resources for Software Development	12
2.1.1	Design Pattern	12
2.1.2	Architecture Pattern	17
2.1.3	Deployment Patterns	19
2.1.4	Anti-pattern	20
2.1.5	(Architectural) Tactics	25
2.1.6	Best Practice	25
2.1.7	Guideline	26
2.1.8	Rule	26
2.1.9	Smells	27
2.1.10	Reference architecture	29
2.1.11	Technical debt	30
2.1.12	Checklists	30
2.1.13	Pitfall	31
2.1.14	Failure	32
2.1.15	Mistake	33
2.2	Machine Learning Overview	34

Chapter 3	State of the Art	36
3.1	Reusable Resources for ML(AI)-Fueled Systems	36
3.2	Evaluation Methods	37
3.2.1	Literature Reviews	38
3.2.2	Illustrative Scenario Evaluation	38
3.2.3	Experimental Evaluation	41
3.2.4	Expert Evaluation	45
3.2.5	Logical Argument Evaluation	49
3.2.6	Additional Evaluation Approaches	51
Chapter 4	The Catalog of Reusable Conceptual Resources	54
4.1	Selection of Resources	54
4.1.1	Literature and Repository-Based Search	54
4.1.2	LLM-Assisted Search	56
4.1.3	Identification and Consolidation of Catalog Entries	57
4.1.4	Final Selection	57
4.1.5	Classification of the Found Resources	57
4.2	Kinds of Reusable Conceptual Resources	58
4.2.1	Solve Resources	61
4.2.2	Do Resources	64
4.2.3	Don't Resources	66
4.2.4	Consider Resources	69
4.2.5	Be aware Resources	70
4.3	Mapping Traditional Kinds of Resources to the New Taxonomy	73
4.4	Classification and Tagging	75
4.4.1	Activity Tags (Support)	75
4.4.2	Quality Tags (Affect)	78
4.5	Catalog Structure	80

4.6	The Data Engineering Part of the Catalog	80
Chapter 5	Catalog Evaluation	82
5.1	Evaluation Dimensions	82
5.2	Catalog Evaluation Method	83
5.3	Results	84
5.3.1	The obligation expressed by the Do and Don't resources and the way they are named	85
5.3.2	The relations between resources are not recorded	87
5.3.3	The resources are confirmed, their relation to AI is not	87
5.3.4	A fault in the questionnaire document	88
5.3.5	Changes suggested by the evaluation	88
Chapter 6	Conclusion	90
6.1	Summary of Contributions	90
6.2	Limitations and Future Work	91
Appendix A	Data engineering	92
Appendix B	Answers of the Experts	150
	Alphabetical Index	155
	Bibliography	155

Chapter 1

Introduction

Machine learning (ML) has become an increasingly important component of modern software systems, enabling applications to make predictions, support decisions, and adapt their behaviour based on data. Unlike conventional software systems, ML(AI)-fueled systems are not defined solely by executable software components. Their development and operation involve a combination of data, data-processing pipelines, feature engineering, model development, evaluation, deployment, and monitoring. Consequently, the engineering of ML systems requires practitioners to address concerns that span both traditional software engineering practices, which are themselves reshaped by the presence of ML components (for example, in how testing and requirements engineering are carried out), and ML-specific activities. In particular, the quality and management of data play a fundamental role throughout the ML lifecycle, making data engineering an essential part of the development of reliable and maintainable ML systems.

Software engineering has long addressed the complexity of system development through the accumulation and reuse of conceptual knowledge. Design patterns, architecture patterns, deployment patterns, best practices, guidelines, rules, anti-patterns, smells, technical debt, checklists, pitfalls, and documented failures are examples of resources that capture knowledge about recurring problems, recommended practices, undesirable decisions, and potential risks. Such resources allow developers and architects to benefit from knowledge acquired through previous projects rather than repeatedly solving similar problems from first principles. The value of these resources lies not only in the solutions they provide, but also in their ability to communicate engineering knowledge, support decision-making, and preserve experience that can be reused in subsequent development activities.

The emergence of ML(AI)-fueled systems, however, expands the range of engineering knowledge that must be captured and reused. In addition to established software engineering resources, ML development introduces concerns that are specific to data and machine

learning workflows. Examples include recurring data quality problems, data versioning and management practices, training-related strategies, and other forms of knowledge associated with the ML lifecycle. The thesis literature review shows that existing research has already produced a variety of specialized collections of reusable resources addressing such concerns. For example, previous studies have proposed catalogs of data smells, ML-specific code smells, ML design patterns, MLOps anti-patterns, software engineering best practices for ML systems, and architectural patterns for specific ML settings. These contributions demonstrate that reusable conceptual knowledge is already being produced across multiple areas of ML engineering, but they also illustrate the diversity of terminology, purposes, and organizational structures used to describe it. This diversity creates a challenge for systematic reuse. Existing resources are commonly organized according to their traditional names, such as “pattern”, “anti-pattern”, ‘smell’, ‘best practice’, or “tactic”. These terms are meaningful within their respective research traditions, but they do not necessarily provide a common perspective for practitioners who need to identify relevant knowledge according to the problem they face or the decision they need to make. Moreover, the same broad engineering objective may be addressed by resources belonging to different traditional types. Conversely, resources sharing the same traditional label may play substantially different roles in engineering practice. This issue becomes particularly relevant in ML systems, where knowledge originating from software engineering, data engineering, ML engineering, and governance needs to be considered together.

To address this heterogeneity, this thesis develops a catalogue of reusable engineering knowledge that brings together resources originating from software engineering, and ML(AI) engineering. The catalogue adopts the notion of a *reusable conceptual resource* as a general abstraction for engineering knowledge that can be applied, consulted, or reused across development contexts. This abstraction provides a common perspective for organizing resources that may otherwise be described using different terminology and resource types. Rather than organizing resources primarily according to their traditional terminology, the catalogue classifies them according to the role they play in engineering decision-making. To establish this classification, existing proposals for reusable resource types were studied and analyzed, and this thesis contributed to refining and finalizing the resource-type definitions and their corresponding templates. The resulting classification distinguishes five categories: *Solve*, *Do*, *Don't*, *Consider*, and *Be aware*. A *Solve* resource provides a reusable solution to a recurring problem; a *Do* resource prescribes an action or practice; a *Don't* resource identifies an approach that should be avoided; a *Consider* resource identifies an option whose adoption depends on context and trade-offs; and a *Be aware* resource signals a condition or symptom associated with potential risk.

Because there was a need for a tagging system to help practitioners effectively locate specific resources, the initial idea was to work along two dimensions: the supported engineering activities and the affected qualities. Activity tags connect resources to activities such as architecture, software design, coding, testing, deployment, data acquisition, data prepara-

tion, data management, feature engineering, model training, experimentation, operations, governance, and project management. Quality tags capture the qualities affected by a resource across software, model, and data domains e.g., maintainability and reliability for software, accuracy and robustness for models, and completeness and consistency for data. This multidimensional organization is intended to support context-oriented retrieval and quality-driven reasoning rather than merely providing a list of resources.

The scope of this thesis is specifically concerned with the data-related portion of this broader catalog. Data engineering is particularly important in ML(AI)-fueled systems because data must be acquired, prepared, managed, and maintained in forms that can support subsequent ML activities. The resulting catalog includes resources addressing, among other concerns, data cleaning, data profiling, data versioning, data governance, data quality, data pipelines, data retention, data usage consent, incremental data processing, and responsible data management. The focus on data engineering provides a concrete and bounded setting in which the proposed organization of reusable conceptual knowledge can be examined and evaluated. The resources included in the catalogue were collected by surveying scientific and grey literature. This process, and the resulting Catalog, are described in Chapter 4.

Accordingly, the main objective of this thesis is to provide a structured and extensible catalog of reusable conceptual resources relevant to data engineering in ML/AI-fueled systems, together with a classification scheme that makes heterogeneous resources easier to understand, locate, and reuse. The contribution is not intended to replace specialized pattern catalogs, data-quality taxonomies, or other existing collections. Rather, it provides a common organizational layer through which different forms of reusable engineering knowledge can be represented and related. In doing so, the thesis aims to support systematic reuse of existing knowledge and provide a basis for further analysis and extension of reusable resources for ML systems.

The remainder of this thesis is organized as follows. Chapter 2 introduces the relevant background concepts and types of reusable conceptual resources in software engineering. Chapter 3 provides a literature review covering current attempts at providing specific resources for ML(AI)-fueled system development. Chapter 4 presents the structure of the whole catalog, including its resources, tagging system, and the process used to collect them. A specific contribution of this work was compiling the section on data engineering and revising the remaining sections of the broader catalog. Due to time constraints and the strategic need to identify structural issues early on, Chapter 5 presents the evaluation strictly for the data-related portion of the catalog, allowing us to improve the structural framework before evaluating the remaining parts. Chapter 6 summarizes the contributions of the thesis and discusses its limitations and possible future work. Finally, Appendix A reports the complete data engineering part of the Catalog.

Chapter 2

Background

Software engineering involves a wide range of knowledge, practices, and solutions that can be reused across different projects and development contexts. Reusable resources capture such knowledge in a form that can be applied beyond the specific context in which it was originally developed. They are usually described through a template, that is, a fixed set of fields that gives every resource of the same type the same structure and makes different resources easier to compare and to reuse. They can address recurring problems, support development activities, guide decision-making, and provide established knowledge for different stages of software development. Depending on their purpose and the type of knowledge they capture, reusable resources may differ in their level of abstraction, scope, structure, and intended use. Documenting this knowledge in a structured and accessible form facilitates its retrieval, adaptation, and reuse, thereby enabling developers to benefit from existing knowledge rather than repeatedly developing solutions from scratch.

2.1 Types of Conceptual Reusable Resources for Software Development

2.1.1 Design Pattern

Design patterns represent solutions to designing good-quality software for software design problems that occur frequently. These solutions operate primarily at the design level, but can also guide implementation decisions by influencing how software modules and classes are structured in code. “*Design patterns are claimed to provide various benefits, including (i) support for better design decisions, (ii) improve communication among developers, (iii) improve or improve software maintainability and reusability, (iv) help to satisfy non-*

functional requirements of the system, and (v) save cost, effort, and time by reusing existing proven solutions". [WMA19]. The Gang of Four design patterns were the first comprehensive attempt to systematically introduce design patterns, presenting a collection of 23 classic software design patterns categorized into Creational, Structural, and Behavioral patterns. [GHJV94a].

Templates for Describing Design Patterns Design patterns are documented using structured templates that aim to present problems, contexts, and solutions in a uniform and comprehensible way. Several influential templates exist in the literature, each offering different levels of detail and emphasis. This section summarises the most widely used templates. The Gang of Four (GoF) template proposed by Gamma et al. [GHJV94a] is the most influential and widely adopted pattern documentation format. It provides a structured way to describe object-oriented design patterns by focusing on intent, applicability, structure, collaborations, and consequences. This template remains the de facto standard for many pattern catalogues.

Table 2.1: Gang of Four (GoF) design pattern template [GHJV94a].

Field	Description
Name	Official name of the pattern.
Classification	Creational, Structural, or Behavioral category.
Intent	Purpose of the pattern; what it achieves.
Also Known As	Alternative names or aliases.
Motivation	Concrete scenario illustrating the problem.
Applicability	Situations where the pattern is appropriate.
Structure	UML representation of classes/objects.
Participants	Roles and responsibilities of pattern elements.
Collaborations	Interactions between participants.
Consequences	Trade-offs and impact on quality attributes.
Known Uses	Real systems or frameworks applying the pattern.
Related Patterns	Patterns commonly used together or alternatives.

The Pattern-Oriented Software Architecture (POSA) template by Buschmann et al. [BMR⁺96] extends pattern documentation to the architectural level. It emphasises problem analysis through forces, demonstrates solutions through examples and dynamics, and is frequently used for documenting large-scale architectural patterns.

Table 2.2: Buschmann et al. (Pattern-Oriented Software Architecture) template [BMR⁺96].

Field	Description
Name	Pattern identifier.
Also Known As	Other names used for the same pattern.
Example	Concrete motivating example.
Context	Situational context in which the pattern applies.
Problem (with Forces)	The problem and conflicting constraints.
Solution	Architectural arrangement resolving the forces.
Structure	Static design representation.
Dynamics	Runtime behaviour and interactions.
Implementation	Practical implementation guidelines.
Example Resolved	How the initial example is solved.
Variants	Alternative versions of the pattern.
Known Uses	Documented applications in real systems.
Consequences	Benefits and liabilities.
See Also	Related or complementary patterns.

Meszaros and Doble [MDVC98] introduced a meta-pattern structure that generalises how patterns are described across domains. Their template highlights the underlying problem, surrounding context, competing forces, and resulting context after applying a solution, making it especially useful for pattern writers and organisational pattern languages.

Table 2.3: Meszaros & Doble meta-pattern template [MDVC98].

Field	Description
Title	Name or title of the pattern.
Problem	Core recurring issue requiring resolution.
Context	Conditions influencing the problem.
Forces	Constraints, trade-offs, and competing concerns.
Solution	General arrangement addressing the forces.
Indications	When the pattern should be applied.
Resulting Context	State of the system after applying the pattern.
Rationale (optional)	Why the solution works.
Aliases (optional)	Alternative names.
Acknowledgements (optional)	Credits and attributions.

Minimal and micro-pattern templates are lightweight documentation formats designed for

teaching, fast communication, and agile environments. Cunningham’s micro-patterns and Tahara’s mini-patterns [Cun95, TT07] focus on capturing only the essential elements of a pattern—typically name, problem, solution, and a few contextual factors.

Table 2.4: Minimal / Micro-pattern templates [Cun95, TT07].

Template	Fields
Micro-Pattern	Name, Problem, Solution.
Mini-Pattern	Name, Context, Forces, Solution, Benefits, Consequences.

Bradapp’s “Patterns in a Nutshell” template [App98] provides a succinct pattern summary aimed at practitioners who need fast lookup rather than full narrative detail. It captures the problem, forces, solution, consequences, and examples in a compact, easy-to-scan structure commonly used in online repositories.

Table 2.5: Bradapp “Patterns in a Nutshell” template [App98].

Field	Description
Name	Pattern name.
Context	Background conditions and setting.
Problem	Problem motivating the pattern.
Forces	Key trade-offs.
Solution	Structural and behavioural description.
Examples	Real instantiations of the pattern.
Resulting Context	System state post-application.
Rationale	Explanation of why the solution is appropriate.
Related Patterns	Complementary or alternative patterns.
Known Uses	Systems where the pattern is applied.

Example – Abstract Factory In this thesis, a hybrid template is used to document Abstract Factory design patterns. This template combines the structural elements of the GoF pattern format with the problem–context–forces reasoning proposed by Meszaros and Doble. Table 2.6 shows this pattern documented using the hybrid template.

Table 2.6: Abstract Factory pattern described using the hybrid template.

Field	Description
Name	Abstract Factory
Classification	Creational Pattern
Problem	Systems that require the creation of families of related objects risk inconsistency and tight coupling if clients construct concrete components directly.
Context	Cross-platform UI toolkits where multiple look-and-feel families (e.g., Motif, Windows) must coexist and remain interchangeable.
Forces	• Clients should not depend on concrete product classes. • Product families must remain consistent across instantiations. • Replacing a family should not require modifying client code.
Intent	Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
Motivation	Each look-and-feel standard defines its own set of widgets. Abstract Factory enables switching families without modifying client code and ensures consistency across the interface.
Solution	Define an abstract factory declaring creation methods, concrete factories implementing those methods, and product abstractions ensuring interchangeability.
Structure	See Figure 2.1.
Participants	AbstractFactory, ConcreteFactory, AbstractProduct, ConcreteProduct, Client.
Collaborations	The client interacts only with AbstractFactory; ConcreteFactory creates ConcreteProducts that implement AbstractProduct interfaces.
Consequences	Supports product family consistency, isolates concrete classes, enables swapping families, but makes adding new products harder because interfaces must expand.
Resulting Context	Decoupled and consistent product families with improved configurability.
Known Uses	InterViews toolkit (<i>WidgetKit</i> , <i>DialogKit</i>) [LVC89]; ET++ framework [WGM88]; multiple GUI toolkits.
Related Patterns	Often combined with Factory Method or Prototype; concrete factories often implemented as Singletons.

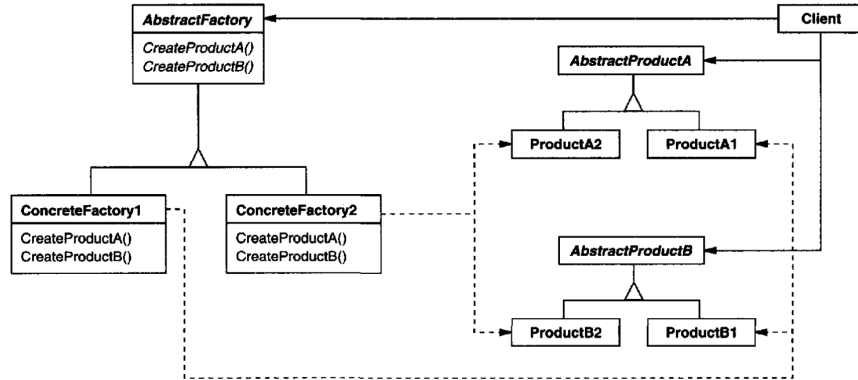


Figure 2.1: Structure of the Abstract Factory pattern (adapted from [GHJV94a]).

2.1.2 Architecture Pattern

A *software architecture pattern* is a reusable, high-level solution for organizing the fundamental structure of a software system. It provides a blueprint that defines the system’s major components, their responsibilities, the constraints governing their interactions, and the overall configuration that enables desired qualities such as scalability, maintainability, availability, and performance. Architecture patterns explicitly capture the *forces*—for example, distribution, data consistency, latency, and modifiability—that shape system design, and they propose proven system-level arrangements that balance these forces in predictable ways. They encompass both classical architectural patterns (e.g., Layers, Pipes-and-Filters, Client–Server) and architectural styles (e.g., Microservices, Event-Driven, Service-Oriented), providing actionable guidance for structuring complex systems [BMR⁺96].

Template for Architecture Patterns Architecture patterns are documented using a *POSA-style template* derived from the *Pattern-Oriented Software Architecture* (POSA) catalogue by Buschmann et al. [BMR⁺96]. This template extends the GoF design pattern format [GHJV94a] to address system-level concerns by explicitly capturing architectural context, forces, structural organization, and resulting consequences. Table 2.7 presents the standard template used to describe architecture patterns.

Table 2.7: Architecture Pattern Template

Field	Description
Name	Official name of the architecture pattern.
Classification	Category of the pattern (e.g., Layered, Client–Server, Event-Driven, Microservices).
Problem	The recurring architectural problem addressed by the pattern.
Context	Situations or environments in which the pattern is applicable.
Forces	Conflicting constraints or design concerns (e.g., scalability, latency, consistency, maintainability).
Solution	System-level arrangement that resolves the problem and balances the forces.
Structure	Static organization of components and connectors; how elements are arranged.
Dynamics	Behavior and interactions at runtime; how components communicate and collaborate.
Consequences	Benefits, trade-offs, and potential liabilities resulting from the pattern.
Known Uses	Real-world systems or frameworks that implement the pattern.
Related Patterns	Patterns that are complementary, alternative, or commonly used together.

Example – Model-View-Controller (MVC) The MVC pattern organizes an interactive application into three main components: the *model*, which holds the core data and functionality; *views*, which display information to the user; and *controllers*, which manage user input. Together, views and controllers form the user interface, while a change-propagation mechanism keeps the interface and model consistent[BMR⁺96]. Table 2.8 shows the MVC pattern described using the POSA-style template.

Table 2.8: Model-View-Controller (MVC) described using the POSA-style template [BMR⁺96]

Field	Description
Name	Model-View-Controller (MVC)
Classification	Architectural Pattern
Problem	User interfaces often change, requiring adaptability without affecting core logic. Tight coupling between interface and logic increases maintenance effort and risk.
Context	Interactive applications with flexible human-computer interfaces.
Forces	• Changes to the interface should not affect core logic. • Multiple views must remain consistent with the same model. • Independent development of interface components should be possible.
Solution	Separate application into model (data/logic), views (presentation), and controllers (input handling). Model notifies views and controllers of changes to maintain synchronization.
Structure	Model maintains core logic and data; views present model data; controllers translate user input into operations on model/view.
Dynamics	Views and controllers respond dynamically to model changes; controllers propagate commands to model.
Consequences	Supports multiple synchronized views, improves maintainability, but increases system complexity.
Known Uses	Modern web frameworks: Django, Ruby on Rails, Angular, React (with Redux/MobX).
Related Patterns	Observer, Strategy, Command; variants: Model-View-Presenter (MVP), Model-View-ViewModel (MVVM).

2.1.3 Deployment Patterns

In the context of software development, deployment refers to the installation of a specific version of software into a targeted environment—such as an integration testing stage or a production system—and it may or may not be directly tied to releasing features to end users [KHDW16]. A deployment pattern, therefore, is a systematic and repeatable method for executing this deployment process across different environments throughout

the software development lifecycle. These patterns extend beyond the technical act of launching applications; they encompass the complete workflow, including sequence of tasks, configuration management, quality validation, and process controls that ensure the software transitions smoothly from development to production.

Deployment patterns serve as a structured roadmap that reduces errors, enhances maintainability, supports scalability, and ensures reliable and consistent software releases [Alu23]. These patterns address essential operational and business objectives: reducing or eliminating downtime, improving software quality and performance, minimizing technical and operational risks, enabling rapid incident response and rollback, accelerating time-to-market through automation (e.g., CI/CD pipelines), reducing manual effort and operational costs, and fostering collaboration across development, testing, security, and operations teams.

Example – Blue-Green Deployment Pattern Blue-Green Deployment Pattern is a software deployment strategy that enables zero-downtime updates to a production environment. In this approach, two identical environments are maintained: one environment running the current version of the software (blue), and the other environment hosting the updated version (green).

At any given time, only one environment is active and serving production traffic, while the other remains idle. The blue environment serves as the current production environment, whereas the green environment is used to test and validate the updated version of the software. Once the new version has been thoroughly tested and verified in the green environment, traffic is redirected from the blue environment to the green environment. Consequently, the green environment becomes the new production environment, and the blue environment is kept idle as a backup.

This deployment strategy eliminates downtime and enables quick rollback to the previous version if any issues occur with the updated release. It also allows developers to test and validate the new version comprehensively before making it available to end users, ensuring that it functions correctly and reliably.

Implementing Blue-Green Deployment requires a robust and well-designed infrastructure, including load balancers and automated deployment tools to facilitate traffic switching between the environments. Additionally, the blue and green environments must be identical to ensure a seamless deployment process [DMG17].

2.1.4 Anti-pattern

An *anti-pattern* describes a common but ultimately counter-productive solution to a recurring problem in software systems. It highlights a practice that initially appears valid,

yet leads to negative consequences such as excessive cost, poor maintainability, or system fragility. For each anti-pattern, the authors provide identification clues (symptoms), underlying causes (forces), and a *refactored solution* — a corrective path to restore design integrity and prevent further deterioration [BMMM98]. Documentation styles for anti-patterns vary based on the required depth of analysis. The most comprehensive format, popularized by Brown et al. [BMMM98], focuses on the technical and organizational “forces” that lead to failure (Table 2.9).

Table 2.9: AntiPattern Specification Template

Field	Description
AntiPattern Name	A unique, pejorative noun phrase used to identify the principles and facilitate organizational terminology.
Also Known As	Additional popular or descriptive names and phrases for the AntiPattern.
Most Frequent Scale	The design level where the pattern fits: idiom, micro-architecture, framework, application, system, enterprise, or global/industry.
Refactored Solution Name	The identifying name of the refactored solution pattern.
Refactored Solution Type	The type of action required: Software (new code), Technology(acquisition/adoption), Process (activities), or Role (responsibilities).
Root Causes	General human/organizational causes: haste, apathy, narrow-mindedness, sloth, avarice, ignorance, pride, or responsibility.
Unbalanced Forces	Primal forces ignored or misused: management of functionality, performance, complexity, change, IT resources, or technology transfer.
Anecdotal Evidence	(Optional) Often-heard phrases or comedic material associated with the pattern.
Background	(Optional) Further examples or general interesting information regarding where the problem occurs.
General Form	Generic version of the AntiPattern (often including a diagram and prose) that the solution must resolve.
Symptoms and Consequences	A bulleted list of observable signs and the resulting negative impacts on the system.
Typical Causes	Unique technical or situational causes identified in addition to the universal root causes.
Known Exceptions	Specific situations or contexts where the AntiPattern behavior is actually acceptable or necessary.
Refactored Solutions	A structured, step-by-step explanation of how to resolve the forces identified in the General Form.
Variations	(Optional) Known major variations of the AntiPattern or alternative solution design points.
Example	Practical application of the solution to the problem, including problem/solution diagrams and descriptions.
Related Solutions	Cross-references to other AntiPatterns or design patterns, explaining differences and how they resolve newly created forces.

In addition to the comprehensive formats used by Brown et al., Laplante and Neill [LN05] advocate for a more pragmatic, less formal structure that prioritizes the immediate identification of dysfunctional behavior and provides actionable remedies for both the organization and the individual.

Table 2.10: Alternative Less Formal AntiPattern Structure

Field	Description
Name	A name that conveys the antipattern’s meaning.
Central Concept	A short synopsis enough to make the antipattern identifiable.
Dysfunction	General problems with current practice and a list of symptoms.
Vignette	A real or prototypical situation providing context and richness.
Explanation	Causes, consequences, and historical or cultural analogies.
Band Aid	Short-term fix or coping strategy for those unable to refactor properly.
Self-Repair	The first step for the perpetrator to improve their own behavior.
Refactoring	Changes needed to remedy the situation and their rationale.
Observations	Optional section for additional comments or items of note.
Identification	Assessment instrument (diagnostic questions) for informal evaluation.

Example – The Blob (or God Object) The *Blob* is one of the most well-known development anti-patterns described in [BMMM98]. It occurs when a single class or module takes on most of a system’s responsibilities, while other classes merely store data. Although such concentration of logic may seem convenient at first, it leads to tightly coupled, procedural-style architectures that are difficult to maintain and extend.

Table 2.11: The Blob Anti-Pattern Specification

Field	Description
AntiPattern Name	The Blob
Also Known As	God Class, Winnebago
Most Frequent Scale	Application / Micro-architecture
Refactored Name	Solution Form Follows Function (Responsibility Redistribution)
Refactored Type	Solution Software / Process
Root Causes	Sloth, Haste, Ignorance
Unbalanced Forces	Management of Complexity, Management of Change
Anecdotal Evidence	"This is the class that actually does everything." "Don't touch the MainController, the whole system breaks if you change a line."
Background	Often occurs when a small prototype is incrementally expanded without a formal architectural review, leading to one class "eating" the logic of its neighbors.
General Form	A large central controller class that contains the majority of the system's logic, surrounded by "Anemic" data objects that only hold attributes with getters and setters.
Symptoms and Consequences	• Poor cohesion and high coupling. • Difficult to unit test due to monolithic size. • Limited reuse of components. • Long compile/build times for the main module.
Typical Causes	Lack of object-oriented design experience; evolution of a wrapper class; "feature creep" into a single successful module.
Known Exceptions	Small utility scripts or simple prototypes where speed is prioritized over long-term maintainability.
Refactored Solutions	Identify cohesive sets of operations within the Blob and migrate them to the passive data classes or new service classes. Apply MVC to separate UI from logic.
Example	A GUI application where one class handles user input, database connections, business rules, and report formatting.
Related Solutions	Poltergeists (classes with no real responsibility), Functional Decomposition (procedural code in OO clothes). 24

2.1.5 (Architectural) Tactics

Architectural tactics represent fundamental design decisions aimed at achieving specific quality attribute responses — such as performance, reliability, or security — by manipulating architectural parameters [BBK03]. In other words, a tactic defines how a particular design choice influences the quality attributes of a system. Tactics serve as the building blocks of architecture patterns; if patterns are comparable to molecules, tactics are the atoms from which they are formed [BCK13]. For instance, the Model–View–Controller (MVC) pattern incorporates tactics such as encapsulation, use of intermediaries, and runtime binding to improve modifiability and maintainability [Pet15]. Each tactic operationalizes the relationship between architectural decisions and system qualities, allowing architects to reason about trade-offs and systematically improve the design [Inf20, KZA⁺20].

Example – Heartbeat Tactic The *Heartbeat* tactic is a reliability tactic that enables a system to detect component failures. One component (the *emitter*) periodically sends a “heartbeat” signal to another (the *monitor*). If the monitor does not receive the signal within a predefined interval, it assumes that the emitter has failed and triggers a fault-handling or recovery mechanism.

Quality Attribute Addressed: Reliability

Design Goal: Detect failures in components or communication links.

Architectural Decision: Introduce periodic message exchange between components.

Trade-offs: Increases communication overhead but improves failure detection speed.

Example: Commonly used in distributed or safety-critical systems to ensure timely fault detection and recovery [BCK13, KZA⁺20].

2.1.6 Best Practice

A *best practice* is a proven and widely accepted method or technique that consistently leads to desired results when applied in a specific context. In software engineering, best practices encapsulate practical knowledge derived from collective experience, aiming to improve quality, maintainability, and efficiency of both development processes and system architectures [BCK13, JBR97, Som16].

Unlike architecture patterns or tactics — which formalize reusable design or decision structures — best practices are often more procedural and prescriptive. They guide developers and architects toward actions that reduce common risks or inefficiencies (for example,

performing regular code reviews, applying continuous integration, or documenting architectural decisions). Best practices can apply across all levels of abstraction — from coding conventions to architectural governance — and evolve as technologies and methods advance.

Example – Separation of Concerns (SoC) The *Separation of Concerns (SoC)* principle is a well-established architectural best practice. It encourages dividing a system into distinct modules, each responsible for a specific functionality or concern. This separation reduces complexity, improves maintainability, and allows independent development and modification of system components [BCK13].

2.1.7 Guideline

Guidelines provide recommended practices, rules, or principles that assist architects, developers, and designers in making consistent decisions throughout a project. Unlike architecture patterns or tactics, which offer structured solutions to specific problems, guidelines serve as advisory aids that promote uniformity, maintainability, and alignment with organizational or domain-specific standards. They capture lessons learned, preferred approaches, and architectural conventions that help ensure quality and reduce design risks [HNS00, BCK13].

Example – Minimize Coupling Between Components A common guideline in software architecture is to minimize coupling between system components. This guideline recommends that components should interact through well-defined interfaces and avoid unnecessary dependencies on internal implementations of other components.

2.1.8 Rule

In software engineering, rules capture architectural intent, design rationale, and domain constraints in a format that can be methodically reused across projects and systems. Rules are explicit, declarative constraints that govern system structure, behavior, and evolution. Unlike implementation mechanisms, rules express what must or must not be done rather than how it is implemented. Architectural decisions including constraints and rules play a fundamental role in shaping system quality attributes and long-term maintainability. Rules therefore serve as a mechanism for preserving architectural integrity and preventing design erosion over time [BCK13].

Rules differ from other reusable software engineering artifacts in both purpose and abstraction level. While design patterns propose general solutions to recurring problems, and architectural styles define high-level system organization, rules impose explicit constraints on design and implementation choices. patterns describe proven solutions, whereas constraints and rules define boundaries within which solutions must exist [GHJV94b]. Similarly, Rozanski and Woods distinguish between architectural structures and architectural rules, noting that rules enforce consistency and guide system evolution [RW12].

Example – Layered Dependency Constraint Business logic components must not directly access data persistence components. All data access must occur through repository interfaces.

2.1.9 Smells

In software engineering, “smells” are indicators of potential problems that can affect code quality, maintainability, or project evolution. They are usually categorized based on the scope they affect. The main categories are: Code Smells, Design Smells, Architectural Smells, Test Smells, Database Smells, Implementation Smells [SS18].

Code smell It is a symptom in source code that could have further conceptual drawbacks with software maintenance and evolution, and overall code quality.. In some cases, such symptoms may originate from activities performed by developers under time pressure, e.g., implementing urgent patches or simply making suboptimal choices. In other cases, smells come from some recurring, poor design solutions, also known as anti-patterns. Studies reveal that code smells deteriorate code readability and are a precursor of changes and faults, therefore early detection is required [Fow99].

Kent Beck and Martin Fowler emphasized that code smells guide refactoring decisions. Experienced developers often intuitively recognize when code has deteriorated in quality and requires improvement. Although code smells are not defined by strict quantitative metrics, certain structural characteristics commonly signal the need for refactoring, such as excessively long methods, overly large classes, or classes that contain an excessive number of instance variables.

Example – Long Method A method (function/procedure) contains too many lines of code, therefore, it difficult to understand, extend and modify. It is very likely that this method has too many responsibilities, hurting one of the principles of a good OO design (SRP: Single Responsibility Principle) [FOVF20].

Why This Happens: Methods grow over time as new logic gets added on top of existing code, while old code is rarely removed. Adding a few lines is easier than creating a new method, so developers extend existing ones. This leads to large, monolithic methods that are hard to read and have tangled responsibilities.

How to Avoid: If part of a method needs a comment to explain it, extract that part into its own method. Even a single line can deserve its own method if it represents a distinct action. Clear, simple method names reduce confusion and make code more readable and maintainable.

Design smell In software architecture, design smells are structural, high-level signs that go against basic design principles (such as SOLID¹ or DRY²), leading to more technical debt, poor maintainability, and stiff, brittle code. Design smells reveal deeper, systemic problems that impact several classes, in contrast to code smells, which are surface-level, local problems.

Example – Blob (or God Class) A God Class is a design smell in which a single class implementing several responsibilities, having a large number of attributes, operations and interacts with many other classes [Fow99].

Why This Happens: This smell occurs when developers keep adding new responsibilities to one class. As a result, the class becomes large, complex, and hard to maintain.

How to Avoid: This smell can be avoided by using abstractions, reducing direct dependencies, and organizing modules in a clear hierarchy.

Test smell A test smell is generally defined as a symptom in test code that indicates a deeper design or implementation problem even if the test still executes correctly. This

¹SOLID is an acronym for five fundamental principles of OO:

Single Responsibility Principle: A class should have only one reason to change, meaning it should focus on a single responsibility or task.

Open/Closed Principle: Software entities should be open for extension but closed for modification, allowing behavior to be added without altering existing code.

Liskov Substitution Principle: Subtypes must be substitutable for their base types without affecting the correctness of the program.

Interface Segregation Principle: Clients should not be forced to depend on interfaces they do not use; prefer many small, specific interfaces over one large, general-purpose interface.

Dependency Inversion Principle: High-level modules should not depend on low-level modules; both should depend on abstractions.

²The DRY (Don't Repeat Yourself) principle states that “every piece of knowledge must have a single, unambiguous, authoritative representation within a system”.

may include poor organization, unclear structure, violations of testing best practices, or patterns that make the tests harder to maintain, understand, or evolve [GK18].

Example – Fragile Test A Fragile Test is a test smell in which a test fails due to changes in the source code that are unrelated to the functionality being tested, often because the test is too tightly coupled to implementation details [GK18].

Why This Happens: This smell occurs when tests verify the internal state or structure of an object (like checking specific variable names or internal paths) rather than the observable behavior or output. As a result, any minor refactoring or cosmetic change to the production code causes the test to break.

How to Avoid: This smell can be avoided by testing the public interface and behavior of the code (Black Box testing), using stubs or mocks to isolate dependencies, and avoiding assertions that rely on the specific implementation logic of the system.

2.1.10 Reference architecture

In today's fast-paced world, many companies need to quickly create and support various systems. Without clear architectural guidance, this process can take a lot of time and lead to complicated, custom-built solutions. Reference architectures address this challenge by providing a standard method to speed up development and make maintenance simpler. By offering a common architectural blueprint, they help developers understand systems better and allow them to reuse knowledge across projects.

Reference architectures meet this need by acting as templates for system development. They provide best practices, guidelines, documentation, and technology suggestions, helping to reduce system complexity and cut down on development and maintenance costs. However, for a reference architecture to be truly useful, good documentation is vital. Since reference architectures are purposely abstract, they usually give limited details on implementation [MFAFM17].

Example – Microsoft Azure Cloud Adoption Framework (CAF) The Microsoft Cloud Adoption Framework for Azure covers the whole life of a cloud solution, from the decision to adopt the cloud to the management of the solution once it is in use, and is addressed to cloud architects, IT professionals, and decision makers. It collects best practices, documentation, and tools covering the phases of cloud adoption, from the definition of the strategy and the preparation of the organization to the governance and the management of the adopted solutions [Azu25]. The framework acts as a template, guiding organizations

on how to structure cloud solutions while leaving implementation details flexible. This reduces complexity, ensures consistency, and accelerates development across projects.

2.1.11 Technical debt

Technical debt in software development is like a shortcut, a quick fix or design choice that helps you move faster in the short term, but may make future changes more difficult, costly, or even impossible. It can be thought of as a debt that affects the ease of maintaining and developing your system over time. Using a financial analogy makes this clearer. Just like taking out a loan, technical debt gives you a short-term advantage, but it comes with additional effort, risk, or cost that increases the longer you ignore it. Many organizations struggle to manage technical debt because they don't fully understand it, and this often leads to approaches that don't actually solve the problem or even make it worse. By identifying and properly managing technical debt, developers and managers can make smarter decisions and maintain the health of their software in the long term [Bro24].

Example – Legacy code Legacy code refers to old code that is no longer in line with current best practices or technologies but is still in use.

Why This Happens: As technology evolves, older code may not efficiently utilize new features or practices. Legacy code is often not regularly updated or refactored, and in many cases the original developers have left the organization, making the codebase difficult to understand and maintain. Additionally, legacy code frequently results from rapid development cycles or insufficient documentation.

How to Avoid: To avoid this issue, the codebase should be refactored and updated on a regular basis. Continuous training should be provided to development teams to ensure familiarity with current best practices, and a culture of continuous improvement should be actively encouraged.

2.1.12 Checklists

When engineers work on a complex system, it is easy to overlook things — not because they are careless, but because human memory is limited and attention is finite. A checklist solves this problem in a simple but powerful way: it captures, in a structured list of questions or conditions, everything that must be verified at a specific point in the development process. But in software engineering, a checklist is more than a reminder tool. It is a reusable knowledge artefact — each item on the list encodes a known risk, a common defect, or an

important practice that experienced engineers have learned, often the hard way, must not be skipped.

The idea of using checklists as a formal reusable resource in software engineering goes back to Michael Fagan's work at IBM in the 1970s. Fagan introduced a structured inspection process in which teams would systematically review code and design documents using pre-defined checklists. He showed that significant improvements in both programming quality and productivity could be achieved when inspectors followed a structured process with clearly defined roles and a systematic approach to categorising and analysing the errors they found [Fag76].

2.1.13 Pitfall

A pitfall refers to a common hidden problem or mistake that individuals or teams may encounter during a process, often without recognizing it until it causes negative effects. In software and project management contexts, a variety of pitfalls have been identified by developers and academic research. These pitfalls can hinder the success of projects, reduce quality, and create barriers to achieving goals unless they are properly understood and managed. One research study emphasizes that understanding and addressing pitfalls is crucial to successful implementation of project management practices, suggesting that teams must identify the pitfalls that may arise in order to avoid them and improve outcomes [LSed].

Example – Hidden Pitfalls in Kanban Implementation One clear example of pitfalls in software development comes from research into agile methodologies such as Kanban. According to this study, hidden pitfalls of Kanban in software development include organizational resistance to change and superficial implementations of the method that prevent teams from realizing intended benefits. The research explains that when teams fail to adopt the necessary mindset to support Kanban, this becomes a pitfall affecting the entire project, ultimately causing the implementation to fail. Additionally, many teams claim to use Kanban but implement only a shallow imitation of the real method, which also creates pitfalls that hinder success [Ant23].

Why This Happens: The study on Kanban reveals that these pitfalls arise because organizations often do not fully understand or embrace the values required for effective agile practices. When teams adopt agile terminology without internalizing its principles, the result is often incomplete or incorrect application of the methodology, leading to hidden pitfalls that affect project outcomes.

How to Avoid: To prevent problems from occurring, organizations must go beyond the

superficial use of tools and properly understand and support the underlying principles and values of the methodology. In this regard, providing training and cultural support to align the organizational mindset with the chosen frameworks is essential, as well as utilizing experienced experts to identify and resolve challenges early. Although the focus of this study is on Kanban, the principle of proactive problem management is vital in all areas of project management and software applications.

2.1.14 Failure

Failure in software engineering is when defects (or faults) lead to unwanted, unintended software events (incidents), with significant loss, degradation of service or risk to safety. These failures can be caught internally during development or externally in production; either way, they indicate shortcomings of the engineering process and serve as valuable feedback to facilitate risk identification and mitigation [ASH⁺25].

In the context of reusable conceptual resources, a failure is not treated as a general learning experience but as a specific, documented failure case that encodes reusable knowledge about what went wrong, under what conditions, and how it can be avoided. The value of a failure as a reusable resource lies precisely in its specificity: a well-documented failure case records the observable symptoms, root causes, context, and consequences of a real breakdown in a system, process, or artefact, so that future practitioners can recognise similar conditions and take preventive action.

Software failures carry significant technical and organizational consequences, especially in high-reliability domains such as aerospace, healthcare, and finance, where software malfunctions may lead to financial loss or even threats to human life. Consequently, continuous learning from failures is a core engineering principle aimed at preventing recurrence and improving system reliability.

Example — Ariane 5 Flight 501 (1996) The rocket’s inertial reference system failed because a 64-bit floating-point number was converted to a 16-bit integer without an overflow guard, causing an operand error. This specific failure case has been widely documented and classified as a reusable resource in software dependability literature, encoding the recurring failure pattern of unchecked numeric conversion in safety-critical embedded systems — a pattern practitioners can apply when reviewing any system that reuses software components from a different operational context [Inq96].

2.1.15 Mistake

A mistake in software engineering refers to an incorrect action, decision, or assumption made during the software development process that can negatively affect the quality, cost, or success of a system. Several studies explicitly discuss common mistakes in software engineering, particularly in early phases such as requirements engineering and design. These mistakes are usually well known in the literature and are intended to be avoided, yet they frequently reoccur due to human factors, organizational pressure, or lack of experience. Research emphasizes that mistakes made in early development stages tend to propagate to later phases, making them more expensive and difficult to correct. Identifying common mistakes helps engineers improve decision-making and reduce long-term project risks. [Dav95].

Example – Poor or Incomplete Requirements Specification One widely documented example of a common mistake in software engineering is poor or incomplete requirements specification. Several academic studies explicitly describe this as a common mistake in requirements engineering, noting that unclear, ambiguous, or missing requirements often lead to system rework and project failure. According to the literature, requirements-related mistakes are among the most frequent and costly errors in software projects, as they directly influence design, implementation, and testing activities. [ZC05].

Why This Happens: Research identifies multiple reasons for why these mistakes occur. Common causes include insufficient stakeholder involvement, communication gaps between developers and customers, and time pressure during early project phases. Studies on requirements engineering mistakes show that teams often underestimate the importance of proper requirement elicitation and validation, assuming that issues can be fixed later in development. This assumption itself is identified as a recurring mistake in software engineering practice.

How to Avoid: Scientific sources emphasize the adoption of preventive and structured approaches to prevent common mistakes in software projects. In this regard, the application of systematic requirements engineering methods—including careful extraction, validation, and documentation—is recognized as one of the main pillars of success. In addition, continuous stakeholder participation in all stages of development and conducting early inspections to identify errors before moving to subsequent phases are of great importance. The results of these approaches show that timely detection and handling of challenges not only minimizes development risks, but also leads to significant improvement in the final results of the project.

The resource types presented in this section are the ones that recur most often in software engineering. They differ in what they prescribe, in the problems they address, and in the

fields used to describe them, and they are the starting point of the classification proposed in Chapter 4.

2.2 Machine Learning Overview

Machine Learning (ML) is a subfield of Artificial Intelligence concerned with building systems that learn a model directly from data, rather than following explicitly programmed rules, and use this model to make predictions or decisions on new, previously unseen inputs. ML approaches are commonly grouped into a small number of paradigms: *supervised learning*, in which a model is trained on labeled input-output pairs to perform classification or regression; *unsupervised learning*, in which a model discovers structure in unlabeled data, for example through clustering or dimensionality reduction; and *reinforcement learning*, in which an agent learns a sequence of actions through trial-and-error interaction with an environment, guided by reward signals. *Deep learning*, based on multi-layered artificial neural networks, is not a separate paradigm but a family of techniques applicable within any of the above, and has driven much of the recent progress in ML, particularly for unstructured data such as images, audio, and text.

An ML(AI)-fueled system typically consists of a pipeline of interdependent stages: data collection and preparation, feature engineering, model training, model evaluation, deployment, and monitoring. Unlike traditional software, whose behavior is determined entirely by its source code, the behavior of an ML(AI)-fueled system is jointly determined by its code *and* the data used to train it. This additional dependency, together with the iterative and experimental nature of the ML lifecycle, is what distinguishes the engineering of ML(AI)-fueled systems from conventional software engineering.

The stages of this pipeline address different concerns. Data collection and preparation gather the data and bring it into a usable form, by removing errors, handling missing values, and making formats consistent. Feature engineering derives from the prepared data the variables that the model receives as input. Model training learns the parameters of a model from a training set, while model evaluation measures how well the trained model behaves on data that it has not seen, using metrics such as accuracy for classification or error measures for regression. Deployment integrates the trained model into a running system, and monitoring observes its behaviour in production.

Two properties of this pipeline have a direct effect on the engineering of the system. First, the stages are not executed once: the results of the evaluation usually lead to changes in the data, in the features, or in the model, so the development of a model is experimental and iterative, and the experiments have to be recorded if their results are to be reproduced. Second, the behaviour of a deployed model depends on the data it receives, which changes over time. A model that performs well at deployment time can degrade when the

distribution of the input data moves away from the distribution of the training data, so monitoring and retraining are part of the normal life of the system rather than exceptional activities.

An ML(AI)-fueled system is therefore a software system that contains, besides its conventional components, one or more ML components together with the data and the pipelines that produce them. Its development requires the traditional software engineering activities, which are themselves affected by the presence of the ML components, and, in addition, activities that concern data and models, such as data preparation, feature engineering, model training, and model monitoring. This is the reason why the reusable knowledge needed for these systems cannot be taken from software engineering alone.

Reusable Resources in the Machine Learning Field. The resource types reviewed earlier in this chapter — design patterns, anti-patterns, smells, best practices, technical debt, among others — were originally developed for conventional, deterministic software. However, because ML(AI)-fueled systems introduce data as a new, independent source of complexity and failure, the software engineering and ML communities have begun adapting and extending each of these resource types to the specificities of ML systems.

For example, *code smells* have been extended into *ML-specific code smells* [ZCvD22], capturing recurring implementation pitfalls that occur specifically in data cleaning, feature engineering, model training, and model evaluation code, such as directly accessing internal model attributes or hardcoding hyperparameters that should instead be discovered through experimentation. A parallel and largely novel category, *data smells* [FFR22, RRDNP24], has also emerged, capturing quality issues that reside not in the code but in the dataset itself — for example, columns containing implausible values, or inconsistent representations of missing data — since such issues would have no direct analogue in traditional, code-only software.

Similarly, *anti-patterns* have been extended into the operational domain through *MLOps anti-patterns* [VK24], describing recurring dysfunctional practices in the deployment, monitoring, and governance of ML systems in production, such as the absence of data versioning or of automated retraining pipelines. *Architecture patterns* and *tactics* have likewise been adapted into ML-specific and federated-learning-specific design patterns [NBP24], and *best practices* and *guidelines* have been mined directly from ML practitioner communities [MHBLV⁺23] to capture recommendations spanning the full ML pipeline, from data collection through to deployment.

This extension of established software engineering vocabulary into the ML domain is precisely the space in which the catalog developed in this thesis is positioned. Chapter 3 reviews this body of related work in detail, organized by resource type, before Chapter 4 introduces the catalog itself.

Chapter 3

State of the Art

This chapter presents the work done by other researchers on the topic of this thesis. The first part reviews the studies that propose collections of reusable resources for the development of ML(AI)-fueled systems, such as code and data smells, operational antipatterns, best practices, and architectural and design solutions. The second part reviews the methods that these and similar studies used to evaluate their proposals. Together, the two parts show which resources are already available, and which evaluation methods are suitable for a catalog at an early stage of development, such as the one presented in this thesis.

3.1 Reusable Resources for ML(AI)-Fueled Systems

Code and data smells. Zhang, Cruz, and van Deursen [ZCvD22] introduced a catalog of machine-learning-specific code smells aimed at improving code quality in ML applications. To construct and validate the catalog, they conducted a multivocal literature review combining academic and non-academic sources — mining 1,750 academic papers, 2,170 grey literature entries, 491 Stack Overflow posts, and 87 GitHub commits — followed by triangulation with official ML library documentation. This process reduced an initial set of 31 candidate smells to a final catalog of 22 ML-specific code smells, classified by their impact on maintainability, reproducibility, efficiency, robustness, and error proneness. Bhatia et al. [BLR⁺24] complement this with a taxonomy of machine-learning-specific data quality antipatterns, built by analysing existing data cleaning, profiling, and validation tools such as Data Linter and TensorFlow Data Validation to identify recurring antipatterns in dataset preparation.

Operational antipatterns. Virmani and Kuppam [VK24] followed a qualitative, illustrative strategy to identify common MLOps antipatterns across the machine learning lifecycle, validating their relevance through practical deployment scenarios, implementation examples, and mitigation strategies rather than formal empirical experimentation. Their contribution establishes a shared vocabulary for recognizing weaknesses in enterprise ML systems and offers practical guidance to improve reproducibility, scalability, monitoring, governance, and operational reliability in MLOps environments.

Best practices. Mojica-Hanke et al. [MHBLV⁺23] present an empirical study mining 242 posts from 14 Stack Exchange communities to identify and catalog ML best practices reported by practitioners, resulting in a taxonomy of 127 validated practices organized across 10 categories aligned with the ML pipeline stages defined by Amershi et al. All practices are supported by explanations, examples, and references to the existing literature. Serban et al. [SvdBV23] take a complementary, more quantitative approach: after mining 29 practices from academic and grey literature and structuring them across six categories (data, training, deployment, coding, team, and governance), they evaluate real-world relevance through a large-scale practitioner survey of 313 respondents, using descriptive statistics, regression, and predictive models to show that adoption of these practices correlates with perceived software quality and team effectiveness.

Architecture and design. Nazir et al. [NBP24] present the first mixed-method investigation of software architecture design challenges, best practices, and design decisions for ML(AI)-fueled systems, identifying 435 design challenges across six categories, 42 best practices across seven categories, and 27 design decisions across seven categories, derived from a systematic literature review of 41 primary studies and 12 semi-structured expert interviews. Lo et al. [Til21] instead focus on a specific ML setting, presenting the first comprehensive collection of 14 architectural patterns for the design of federated learning systems, organized into four groups — client management, model management, model training, and model aggregation patterns — each documented using a structured template covering context, problem, forces, and solution.

3.2 Evaluation Methods

We draw substantial inspiration from the taxonomy evaluation framework proposed by Szopinski et al. [SSK19]. This framework provides a structured lens to analyse how related studies evaluate catalogs, patterns, smell collections, and other reusable conceptual resources in the machine learning and software engineering literature. Evaluation approaches in this area can be categorised into four major types: *illustrative scenario*

evaluation, experimental evaluation, expert evaluation, and logical/descriptive evaluation. In addition, many contributions rely primarily on systematic or multivocal literature reviews, which serve as a foundational approach. In addition, the studies differ in how the catalogs or collections themselves were constructed. While some contributions derive their artifacts primarily through systematic or multivocal literature reviews, others combine literature-based synthesis with empirical sources such as practitioner input, industrial cases, or observational analyses. Several of the works presented in Section 3.1 do not only propose resources: they also evaluate their value, for example by measuring the effect of the reported problems on real datasets or by collecting the judgement of experts. The following subsections review the approaches used for this purpose.

3.2.1 Literature Reviews

Literature reviews are considered here separately from the other evaluation approaches. In several of the works presented above, a systematic or multivocal literature review is the method used to build the collection of resources, and not the method used to evaluate it. Zhang, Cruz, and van Deursen [ZCvD22], for example, built their catalog of ML-specific code smells through a multivocal literature review of academic and non-academic sources, while Recupito et al. [RRDNP24] combined a systematic literature review with an empirical analysis of real datasets. A literature review supports the construction of a collection and shows that its resources are reported in the literature, but it does not by itself show that they are useful in practice.

3.2.2 Illustrative Scenario Evaluation

Illustrative scenario evaluation (also referred to as *scenario-based evaluation*) is a common approach used to demonstrate the practical value of newly proposed artifacts in software engineering and machine learning research. In this type of evaluation, the proposed taxonomy, catalog, framework, or detection approach is applied to real-world datasets, research articles, or practical examples in order to demonstrate applicability and usefulness [SSK19].

- Foidl, Felderer, and Ramler [FFR22] formally defined data smells as latent data quality issues in AI-based systems. They compiled a comprehensive catalog of 36 data smells, grouped into three categories: believability, understandability, and consistency. In addition, they developed automated detection tools based on both rule-based and ML-based approaches. They evaluated the practical relevance of the proposed catalog by analysing 246 randomly selected Kaggle datasets licensed under Creative Commons CC0. Their analysis showed that most datasets contained several smelly attributes, demonstrating that data smells identified in the catalog are

highly prevalent in real-world machine learning datasets.

- Shome, Cruz, and van Deursen [SCvD22] introduce the concept of data smells in machine learning datasets and contribute a structured catalog of 14 recurring data quality anti-patterns, organized into categories such as redundant values, categorical issues, missing values, and string-related issues, each illustrated with real dataset examples and accompanied by mitigation suggestions to support data-centric machine learning practices. Their evaluation is tightly coupled with the construction of this catalog and is based on an empirical study of 25 structured public Kaggle datasets, where they perform a manual two-pass inspection process to support smell identification: the first pass is exploratory and aimed at detecting potential data quality issues, while the second pass refines, validates, and extends the initially identified smells as more datasets are analyzed. This iterative process leads to the final set of smells and their grouping into categories based on conceptual similarity, although the categorization process itself is not formally specified and is instead driven by qualitative interpretation. The evaluation is further supported by a quantitative analysis of smell prevalence across datasets (frequency counts and distribution patterns) and a refinement step using exclusion criteria and internal author validation to ensure generalizability.
- Haiyin Zhang, Luís Cruz, and Arie van Deursen [ZCvD22] introduced a catalog of machine-learning-specific code smells aimed at improving code quality in ML applications. To construct and validate the catalog, they conducted a multivocal literature review combining academic and non-academic sources, including Stack Overflow posts, GitHub commits, grey literature, and peer-reviewed publications. Their methodology involved mining 1,750 academic papers, 2,170 grey literature entries, 491 Stack Overflow posts, and 87 GitHub commits, followed by triangulation with official machine learning library documentation. The authors then iteratively analysed and refined the identified smells through manual inspection and validation discussions. This process reduced an initial set of 31 candidate smells to a final catalog of 22 ML-specific code smells. The resulting catalog captures recurring implementation pitfalls across stages of the machine learning pipeline, including data cleaning, feature engineering, model training, and model evaluation. It also classifies their impacts on maintainability, reproducibility, efficiency, robustness, and error proneness.
- Ankit Virmani and Manoj Kuppam [VK24] also followed a qualitative illustrative strategy by identifying common MLOps antipatterns across the machine learning lifecycle. They validated their relevance using practical deployment scenarios, implementation examples, and mitigation strategies rather than formal empirical experimentation. Their study introduced a taxonomy of recurring operational and engineering issues. These include disconnected development and operations, manual experimentation and tracking, lack of security and governance, ad-hoc deployment and

monitoring, insufficient ethical considerations, and absence of data versioning. For each antipattern, the authors provided representative code snippets demonstrating problematic practices. They also provided corresponding mitigation approaches and best-practice recommendations using technologies such as Docker, MLflow, CI/CD pipelines, SHAP explainability, secrets management, and DVC-based data versioning. The resulting contribution establishes a shared vocabulary for recognizing weaknesses in enterprise ML systems. It also offers practical guidance to improve reproducibility, scalability, monitoring, governance, and operational reliability in MLOps environments.

- Zhu, Yang, and Ren [ZMZ23] analysed 148 highly cited environmental machine learning research articles to identify recurring methodological weaknesses in the use of machine learning within environmental science. They developed a systematic literature mining pipeline that initially collected approximately 180,000 publications from 10 high-impact environmental journals. These were reduced to 2,900 supervised ML-related papers through keyword- and NLP-based filtering, and then narrowed to 148 highly cited articles using citation-based screening and manual curation.

The selected studies were examined through structured qualitative analysis to identify recurring methodological issues across the ML workflow. The authors then used domain expertise in machine learning and environmental modelling to interpret, validate, and group these issues into coherent categories. This process resulted in more than 30 common pitfalls and corresponding best-practice recommendations. The identified issues covered topics such as data leakage, improper train-test splitting, feature selection bias, model optimisation, evaluation misuse, reproducibility, and explainability. The study further organised these challenges across multiple stages of the machine learning pipeline, including data preprocessing, model development, validation, and interpretation, with the aim of improving robustness, transparency, and scientific validity in ML-based environmental research.

- Matthews [Mat20] The paper contributes a structured discussion of accountability and transparency in AI systems. It organizes the discussion around seven principles derived from the ACM’s Statement on Algorithmic Transparency and Accountability. These include awareness, access and redress, accountability, explanation, data provenance, auditability, and validation and testing. The paper also identifies ten common antipatterns observed in deployed AI systems. These include learning from biased historical data, producing spurious correlations, reusing systems in unintended contexts, and failing to measure social impact. In addition, it provides practical recommendations for researchers and developers. These aim to support the development of more accountable and transparent AI systems.

The paper evaluates its arguments using illustrative real-world case studies and documented incidents. Examples include Amazon’s biased recruiting tool, Microsoft’s Tay

chatbot, demographic disparities in facial recognition systems, and errors in forensic DNA software. These examples are used to demonstrate the identified antipatterns. They also ground the proposed principles and recommendations in documented real-world harms.

- Rucco, Saad, and Longo [RSL25] present a systematic formalization of two hybrid data pipeline patterns: ETLT (Extract–Transform–Load–Transform) and ELTL (Extract–Load–Transform–Load). They also introduce enhanced variants, ETLT++ and ELTL++, positioning all four as formal, reusable design patterns for modern data engineering. The work makes three primary contributions. First, it provides the first academic formalization of ETLT and ELTL as distinct design patterns, explicitly defining their structure, trade-offs, and applicable use cases. Second, it introduces ETLT++ and ELTL++ as enhanced variants that treat governance, quality assurance, and observability as mandatory architectural obligations rather than optional add-ons. This is achieved by embedding data contracts, append-only versioned storage, semantic curation layers, and continuous Service Level Indicator (SLI) monitoring directly into the pipeline design. Third, the paper supplies practical implementation artifacts, including structural diagrams, code templates, and cost-comparison models, to facilitate adoption across multi-cloud environments. Regarding evaluation, the study is primarily conceptual. Validation is limited to illustrative examples using synthetic data and a theoretical cost analysis. This analysis shows that tiered storage in ELTL++ can reduce operational costs by over 90% compared to baseline ELTL. No empirical benchmarking, real-world deployment, or experimental comparison against existing pipelines is conducted. The authors acknowledge this limitation and propose a future benchmark plan covering five dimensions: latency, error containment, reproducibility, cost efficiency, and operational resilience. They suggest using open datasets such as NYC Taxi Trips alongside tools such as Great Expectations or Deequ for future validation. This gap between conceptual formalization and empirical validation is a notable limitation, and one that the present thesis aims to address through practical implementation and experimental evaluation.

3.2.3 Experimental Evaluation

Several studies employed experimental or technical evaluation¹ approaches, referring to controlled empirical methods used to assess the performance of design science artifacts using real or synthetic data.

¹A performance evaluation of an algorithm implementation using real-world data, synthetic data, or no data, designed to evaluate the technical performance, rather than its performance in relation to the real world [PRTV10]

- Recupito et al. [RRDNP24] extended the catalog proposed by Foidl et al. [FFR22] by introducing 12 additional data smells. Their evaluation combined a systematic literature review with empirical analysis on 19 real-world datasets, where they statistically investigated the relationship between detected smells and structural data quality metrics such as completeness, uniqueness, consistency, and readability. They further identified Extreme Value Smell and Missing Value Smell as the most prevalent issues in real-world datasets and demonstrated that high concentrations of data smells can significantly degrade data quality, particularly affecting completeness, consistency, and readability.
- Bhatia et al. [BLR⁺24] propose a comprehensive taxonomy of machine-learning-specific data quality antipatterns. To develop this taxonomy, the authors followed a systematic multi-step approach. They began by analysing existing data cleaning, profiling, and validation tools (such as Data Linter and TensorFlow Data Validation) to identify antipatterns these tools could detect and mitigate. They then performed snowballing from academic publications accompanying these tools. Finally, they conducted a general academic search on Google Scholar using terms related to data quality in machine learning. After collecting candidate antipatterns from tools and literature, the first two authors organised them into a taxonomy using a card sorting method. They held four collaborative sessions (approximately one hour each) to iteratively group and refine the antipatterns based on their characteristics. This process resulted in eight top-level categories and fourteen sub-types, covering structured data quality issues such as Packaging Antipatterns, Schema Violations, Data Miscoding, Feature-Row Imbalance, Data Distribution Antipatterns, Inconsistent Representation, Correlation & Redundancy, and Label Antipatterns.

To evaluate the practical impact of these antipatterns, the authors conducted a large-scale empirical study using software defect prediction (SDP) datasets derived from nine open-source projects. The evaluation addresses four main aspects: (1) the prevalence of antipatterns in real SDP data, (2) the effects of different antipattern-cleaning orders on model performance, (3) the influence of individual antipatterns when all others have been removed, and (4) the consistency of model interpretation results.

- Hahn and Sales [HS25] adopted a controlled simulation-based evaluation approach using synthetically generated datasets with intentionally injected data smells. Their study investigated how data smells behave across different stages of data transformation pipelines and evaluated the accuracy of rule-based smell detection mechanisms by comparing detected smells against known ground-truth injections.
- Giordano et al. [GALP23] conduct a preliminary empirical investigation into the diffusion of code smells in Python-based AI-enabled systems. The study addresses a gap in the literature, as previous research mainly focused on Java and traditional

software systems. The authors analyze 200 AI-enabled projects from the NICHE dataset, covering more than 10,600 releases. They examine the frequency, density, and temporal variation of code smells, as well as the developer activities associated with their introduction. The study identifies Complex List Comprehension as the most prevalent and longest-lived smell, appearing 1,465 times. In contrast, object-oriented smells such as Large Class are not observed. This finding suggests that Python developers tend to prefer non-object-oriented reuse mechanisms. The results further show that 70% of code smell introductions occur during evolutionary activities, particularly merge operations. This observation indicates a broader lack of awareness among practitioners regarding the importance of monitoring code quality throughout software evolution.

The paper evaluates its contributions through a repository mining study. The authors use PyDriller for release extraction and PySmell for code smell detection across 200 open-source AI-enabled Python projects. The evaluation quantitatively measures code smell frequency, density per release, and temporal variation. In addition, a manual commit message analysis categorizes smell-introducing activities into four labels: Bug Fixing, Evolutionary Activity, Refactoring, and Other. To ensure reliability, two authors independently label each commit and resolve disagreements through discussion until consensus is reached. The paper also discusses threats to validity related to dataset selection, tool limitations, and generalizability. Furthermore, all scripts, raw data, and replication materials are made publicly available.

- Nicolás Cardozo, Ivana Dusparic, and Christian Cabrera [CDC23] contributes the first dedicated software engineering analysis of code quality issues in Reinforcement Learning (RL) projects through the systematic study of code smells in RL-based Python repositories. The authors analyze 24 RL projects using eight established software quality metrics related to maintainability, modularity, abstraction, and program structure. Their study identifies a high prevalence of code smells—particularly Long Method, Large Class, Multiply-Nested Container, and Long Parameter List—revealing recurring maintainability and design problems in RL implementations. Beyond reporting quantitative findings, the paper argues that many of these issues stem not only from developer practices but also from the intrinsic structural complexity of RL systems and the limitations of current programming abstraction. The paper evaluates its contribution through a preliminary empirical repository analysis of 24 Python-based RL projects, including 20 popular GitHub repositories and 4 professionally engineered examples from DeepMind’s Acme framework. The evaluation applies eight predefined code smell metrics to assess maintainability and software quality characteristics across the selected repositories. The authors quantitatively compare the frequency and distribution of code smells between community-developed projects and professionally developed RL systems, identifying a total of 1090 code smells and showing that GitHub projects exhibit substantially higher smell density. The eval-

uation further analyzes the most recurrent smell categories and discusses how they reflect structural and abstraction-related challenges inherent to RL implementations. To strengthen validity, the paper also includes a discussion of threats to validity concerning dataset selection, metric thresholds, and generalizability of the results.

- Rucco et al. [RLS26] propose MIND (Metadata-driven INgestion Design pattern), a cloud-agnostic and metadata-driven design pattern for scalable and flexible data ingestion in large-scale data engineering environments. The approach centralizes ingestion configurations within a Unified Metadata Table (UMT), which stores information such as ingestion type, schema definitions, partitioning strategies, transformation rules, and validation parameters. A metadata-driven orchestration mechanism dynamically reads these configurations and adapts pipeline behavior accordingly, enabling support for full, incremental, and hybrid ingestion without requiring manual pipeline modifications. The authors validated the approach on Microsoft Azure using Azure Data Factory, Azure Databricks, and Azure Data Lake, evaluating its performance on both simple and complex datasets scaled up to one billion records. Their results demonstrate that the metadata-driven approach improves scalability, reduces operational complexity, simplifies schema evolution handling, and decreases maintenance effort by enabling reusable and adaptable ingestion pipelines.
- Rucco, Longo, and Saad [RLS24] propose a cloud-agnostic data ingestion design pattern for big data in cloud architectures. Their work targets a persistent gap in existing Data Engineering Design Patterns (DEDPs). The authors observe that while patterns such as Lambda Architecture, Kappa Architecture, and Data Mesh have been well studied, the challenge of high-volume data ingestion from heterogeneous sources remains inadequately addressed. The primary contribution is a flexible, metadata-driven framework. It combines both full refresh and incremental ingestion techniques within a single, parameterized pipeline. A central mapping table, stored in a SQL database, governs each source’s ingestion strategy dynamically. This allows data engineers to modify ingestion types, schema configurations, and data sources with minimal manual intervention. Two incremental techniques are introduced. The first is date column-based ingestion, which filters records using watermark timestamps. The second is hash-based ingestion, which detects changes through record-level hash comparison. The work is evaluated through a controlled experimental approach conducted on a real Azure cloud infrastructure. The testbed includes Azure Data Factory, Azure Databricks with Apache Spark, Delta Lake, and Azure Data Lake Gen2. A synthetic bank transactions dataset ranging from 1K to 1 billion rows is used for testing. Three ingestion methods are compared: full ingestion, incremental ingestion, and hybrid ingestion. The sole evaluation metric is execution time in minutes. Results show that incremental ingestion significantly outperforms full ingestion at scale, completing in 6 minutes compared to 26 minutes for a 1-billion-row dataset. The hybrid mode achieves a balanced result of 9.3 minutes, proving most practical

when data sources have mixed ingestion requirements. However, the evaluation has notable limitations. Only one metric is measured, with no assessment of cost, resource consumption, or data quality. Testing is restricted to a single cloud platform, limiting generalizability. The dataset is synthetic rather than real-world, and no statistical analysis or repeated trials are reported. These limitations suggest that further validation across diverse environments and real-world datasets is needed to confirm the broader applicability of the proposed pattern.

3.2.4 Expert Evaluation

Another important category is expert evaluation, where researchers involve practitioners, developers, or domain experts to assess the validity and relevance of the proposed artifacts.

- Mojica-Hanke et al. [MHBLV⁺23] present an empirical study mining 242 posts from 14 Stack Exchange communities to identify and catalog ML best practices reported by practitioners, resulting in a taxonomy of 127 validated practices organized across 10 categories aligned with the ML pipeline stages defined by Amershi et al., covering areas from model requirements and data collection through to model deployment, monitoring, and implementation, with all practices supported by explanations, examples, and references to existing literature.

The study evaluates its contribution through a multi-phase validation process. Four independent ML experts with research and industry experience reviewed 157 candidate practices. Each practice was labeled as valid, conditionally valid, invalid, or unclear. A practice was included in the final catalog only if at least three experts agreed on its validity. As a result, 127 practices were retained. The study also discusses threats to validity, including selection subjectivity, tagging bias, and limited generalizability. These threats were mitigated through group-based tagging, conflict resolution, and score-based post filtering.

Expert Interview Expert interviews entail evaluating an artefact through experts who have proven expertise and experience in a (for the catalog relevant) domain.

- Nazir et al. [NBP24] The study presents the first mixed-method investigation of software architecture design challenges, best practices, and design decisions for ML(AI)-fueled systems. It identifies 435 design challenges organised into six categories. It also reports 42 best practices across seven categories. In addition, it extracts 27 design decisions spanning seven categories. These results are derived from both academic literature and practitioner experience. The study further highlights corre-

lations among challenges, best practices, and design decisions to support researchers and practitioners in ML system architecture.

The evaluation is based on a combined systematic literature review and expert interviews. The literature review includes 41 primary studies selected from an initial pool of 3,038 peer-reviewed publications across four databases. The study is complemented by 12 semi-structured interviews with ML researchers and practitioners from nine countries. Data extraction and analysis are conducted using content analysis and narrative synthesis. Threats to validity, including search completeness, construct validity, and generalisability, are mitigated through a validated search string, recursive snowballing, a structured extraction form, consensus-based author review, and a publicly available replication package.

Survey Surveys are an easy and cost-effective way (in terms of money, time and effort) of collecting large data sets through standardized questionnaires.

- Washizaki, Khomh, and Guéhéneuc [WUKG19] make a combined contribution toward systematizing software-engineering knowledge for machine learning systems by identifying, classifying, and evaluating architecture and design patterns across both academic and gray literature. They first conducted a systematic literature review to extract machine learning-related architectural patterns, design patterns, and anti-patterns. This resulted in a structured taxonomy of 33 SE patterns mapped onto both the machine learning pipeline (e.g., data processing, training, deployment, monitoring) and standard software engineering lifecycle phases (e.g., architectural and detailed design). Their findings highlight recurring reusable solutions such as microservice architectures, data lakes, Lambda architectures, and closed-loop ML systems. They also expose the prevalence of ad-hoc design practices in real-world ML development. To evaluate the practical relevance of these patterns, they further validated their results through a practitioner survey conducted with software engineers and ML developers during a workshop. Participants reported which patterns they actively use in practice.
- The paper Software Engineering Design Patterns for Machine Learning Applications by Hironori Washizaki and colleagues[WKY⁺22] contributes a structured body of knowledge for designing machine learning software systems through reusable software engineering patterns. Using a multivocal literature review covering both academic and gray literature, the authors identified 15 ML design patterns addressing architectural, programming, and model operation concerns in ML systems. They further classified these patterns into topology patterns, programming patterns, and model operation patterns, while mapping them to software quality attributes such as maintainability, reliability, efficiency, security, explainability, and prediction accuracy. The work also standardizes the documentation of these patterns using a

consistent pattern format including intent, problem, solution, applicability, and consequences, making them easier for practitioners to understand and reuse in real ML system development.

The evaluation combines literature-based analysis with practitioner-oriented empirical validation. First, the candidate patterns extracted from the literature were reviewed and refined with the help of industrial ML developers to ensure practical relevance and remove weak or unclear patterns. Then, the authors conducted a questionnaire survey with 118 ML developers to investigate awareness, adoption, and future intention to use the identified patterns. The results showed that although many developers were unfamiliar with most patterns, several patterns such as ML Versioning and Microservice Architecture for ML were already widely adopted, especially for improving maintainability. The study also found that teams with more systematic reuse practices tended to use more design patterns, suggesting that increasing awareness and structured reuse practices could improve ML software engineering maturity.

- Serban et al. [SKvdBHV20] contribute a comprehensive synthesis and empirical validation of software engineering best practices for machine learning systems. They first mine 29 practices from academic and grey literature and structure them across six categories (data, training, deployment, coding, team, and governance). They then evaluate their real-world relevance through a large-scale practitioner survey. The evaluation is based on 313 valid responses collected via a cross-sectional questionnaire targeting ML practitioners. Participants reported the degree of adoption of each practice using a Likert scale and assessed perceived effects such as agility, software quality, team effectiveness, and traceability. The authors further analyzed the data using descriptive statistics, correlation analysis, and linear regression (with F-tests and R^2 to test significance and explanatory power). They also applied non-linear predictive models including random forests and AutoML approaches. This multi-level evaluation shows both adoption patterns across demographic groups (e.g., team size, region, and organization type) and statistically significant relationships between practice adoption and perceived outcomes. The results ultimately demonstrate that sets of SE practices can meaningfully predict quality-related effects, while also revealing mismatches between practice importance and actual adoption.
- Ximenes and Ramalho [XGLR21] propose a developer-centered ethical Machine Learning framework consisting of 18 concrete guidelines and 24 actionable best practices. These are designed to translate high-level ethical AI principles into operational steps within the ML development lifecycle. The contribution is grounded in an interdisciplinary focus group that integrates perspectives from ML practitioners and experts in law, sociology, economics, and related fields. This focus group is used to systematically derive and structure these recommendations. The proposed framework is then empirically evaluated through a large-scale survey of 134 machine learning

practitioners. The survey assesses both the perceived responsibility of developers to apply each guideline and the actual self-reported adoption of these practices in real-world projects. This mixed-method evaluation enables a comparative analysis between normative acceptance and practical implementation. The results reveal gaps between ethical awareness and operational behavior in industrial and academic ML workflows.

Focus Group Focus groups involve a group of usually between six to ten participants who jointly discuss the evaluation of a catalog. By enabling the exchange of perspectives among participants, focus groups support collective reflection on the catalog and may generate new insights or interpretations.

- J. Heli et al. [HLB⁺24] address the lack of concrete, actionable design guidance for environmentally sustainable ML(AI)-fueled systems by synthesizing a catalog of 30 green architectural tactics organized into six categories — data-centric, algorithm design, model optimization, model training, deployment, and management — derived from 51 peer-reviewed Green AI publications and structured using the Archive of Awesome and Dark Tactics (AADT) template, with the full catalog made publicly available online to support both researchers and practitioners.

The study evaluates its contribution through a two-step approach: first, a literature-based synthesis using qualitative document and content analysis on solution-oriented papers from an existing systematic Green AI review, and second, a *focus group* with three SA4ML experts who validated the soundness and relevance of the tactics, suggested additional literature that led to four new tactics, and refined the descriptions and categorization of the overall collection, with threats to validity related to data source coverage, focus group generalizability, and subjective bias openly discussed and mitigated through iterative peer review within the research team.

- Leuthe et al. [LMHPS24] present the Sustainable Machine Learning Design Pattern Matrix (SML-DPM). It is a framework of 35 design patterns. These patterns are structured across four ML development phases: ML demand specification, data collection and preparation, modeling and training, and deployment and monitoring. They are also mapped to three ESG dimensions: environmental, social, and governance. Each pattern is assigned to one or more of five ML stakeholder groups.

The study addresses the fragmented state of sustainability-of-AI research. It provides an artifact that covers all ML development phases across all ESG dimensions. It also offers actionable, practitioner-validated guidance.

The evaluation follows a four-step Design Science Research process. It combines ex ante and ex post evaluation stages. The first step is problem justification grounded in the literature. The second step is a focus group of 22 researchers to validate the

design against four key requirements. The third step includes three iterative rounds of naturalistic evaluation. These involve two additional focus groups and 12 semi-structured expert interviews totaling 572 minutes. These rounds refine the patterns until saturation is reached. The final step is a case study demonstration across three real-world ML projects using a web-based prototype. The results are assessed using ease of use, usefulness, and behavioral intention criteria. These are measured through a structured Likert-scale survey based on the Technology Acceptance Model.

- Indykov et al. [ISW25] present a systematic literature review aimed at providing a comprehensive picture of quality attributes and architectural tactics for ML(AI)-fueled systems, contributing a common quality model consisting of 11 frequently reported quality attributes, a catalog of 16 architectural tactics mapped to those attributes, and a trade-off impact matrix of 85 potential quality consequences, all derived from 206 primary sources and compared against relevant ISO standards including ISO/IEC 25010 and ISO/IEC 25059.

The study evaluates its contributions through a multi-layered verification approach: the quality model was validated in a focus-group setting with six experts at the Swedish Requirements Engineering Meeting (SiREN 2023) and cross-checked by four ML engineers from Swedish AI companies against real industrial design practices; the list of architectural tactics was similarly reviewed by the same practitioners for applicability and relevance; the trade-off matrix was verified through continuous internal peer-review by all co-authors; and data extraction consistency was ensured through a test-retest process on random paper samples across all three research questions, with all artifacts made publicly available to support reproducibility.

3.2.5 Logical Argument Evaluation

Finally, some studies relied primarily on logical argumentation or descriptive synthesis rather than direct empirical validation.

- Bogner, Verdecchia, and Gerostathopoulos [BVG21] employs a Systematic Mapping Study (SMS) to characterize technical debt and antipatterns in AI-based systems. Following established guidelines for secondary studies in software engineering, the evaluation begins with an automated Google Scholar search, followed by a structured selection process using predefined inclusion and exclusion criteria. The initial set is expanded through iterative forward and backward snowballing until saturation, resulting in 21 primary studies that form the final analysis corpus.

The extracted data is analyzed using qualitative synthesis techniques, including open coding to identify recurring concepts and axial coding to group them into higher-level categories. Findings are consolidated through harmonization and frequency

analysis across studies, covering technical debt types, antipatterns, affected quality attributes, and proposed solutions. This evaluation is literature-driven and interpretive in nature, aiming to construct an evidence-based conceptual overview rather than empirically testing hypotheses.

- Heiland et al. [HHB23] conduct a multivocal literature review (MLR) to collect, categorize, and document design patterns for AI-based systems, covering both new AI-specific patterns and adaptations of traditional ones, resulting in a collection of 70 unique patterns extracted from 51 resources across white and gray literature, organized into seven categories including architecture, deployment, security and safety, and process, and made accessible through a publicly available web-based pattern repository.
- Mailach and Siegmund [MS23] present the largest qualitative empirical study of socio-technical challenges in building ML(AI)-fueled software, identifying 17 anti-patterns grouped into three main organizational areas — organizational silos, communication failures, and leadership vacuum — along with their root causes and practitioner recommendations, drawing on over 66 hours of talks from 73 videos published by the MLOps community, a practitioner group of over 11,000 members, and offering a novel management-level perspective largely absent from prior work.

The study evaluates its contribution through reflexive thematic analysis conducted in two phases — an inductive open-coding phase followed by a deductive refinement phase — with regular researcher meetings to resolve disagreements and iteratively refine themes, and further validates its findings by systematically cross-referencing the identified anti-patterns against five related empirical studies, demonstrating substantial overlap and thereby strengthening the generalizability and trustworthiness of the results, while threats to validity related to subjective analysis, single-community data source, and keyword-based filtering are openly acknowledged and partially mitigated through the two-phase design and breadth of the speaker demographics.

- Lo et al. [LLZ⁺22] present the first comprehensive and systematic collection of 14 architectural patterns for the design of federated learning systems, organized into four groups — three client management patterns, four model management patterns, three model training patterns, and four model aggregation patterns — each described using a structured template covering context, problem, forces, solution, consequences, related patterns, and known uses, and associated with specific state transitions in a defined federated learning model lifecycle to serve as practical architectural guidance for practitioners building federated learning systems. The study derives its patterns through a systematic literature review of 231 papers conducted following Kitchenham’s SLR guidelines, supplemented by a review of 22 additional machine learning and federated learning papers published after the search cutoff and 22 real-world

application cases drawn from the authors’ implementation experience, with 12 patterns sourced from the literature and 2 identified through real-world applications, and each pattern further validated through known industrial uses from systems such as Google’s FedAvg, IBM Federated Learning, and Microsoft Azure Machine Learning.

- Lu et al. [LZX⁺24] The study presents a Responsible AI Pattern Catalog derived from a Multivocal Literature Review (MLR). The dataset includes 205 academic sources and 69 grey literature sources. The catalog contributes a structured collection of responsible AI patterns. These patterns are organized into three main groups. The first group consists of multi-level governance patterns. These cover industry, organization, and team levels. The second group includes trustworthy process patterns. These span the full AI development lifecycle, from requirements engineering to system operation. The third group contains responsible-AI-by-design product patterns. These target supply chain, system, and operational infrastructure layers. The study provides practitioners with systematic, actionable, and consequence-aware guidance. It supports the operationalization of responsible AI beyond high-level principles and purely algorithm-level techniques.

The evaluation is conducted through the MLR methodology itself. The study follows Kitchenham’s SLR guidelines for academic literature. It also follows Garousi et al.’s guidelines for grey literature. A rigorous keyword-based search is performed across five major databases. Snowballing is applied from a seed set covering multiple AI ethics principles. Studies are screened using defined inclusion and exclusion criteria. A pilot study on 20 items is used to calibrate the extraction process. A consistency check is performed in which a second researcher evaluates a random sample of selected studies. Threats to validity are addressed, particularly those related to terminological ambiguity and incomplete coverage. These are mitigated through broad synonym-inclusive search strings and the combination of academic and grey literature sources.

3.2.6 Additional Evaluation Approaches

Although the reviewed literature mainly relies on a limited set of evaluation approaches, the taxonomy evaluation framework proposed by [SSK19] also includes several additional methods that are available for evaluating taxonomies in Information Systems research.

Delphi technique. The Delphi technique is a structured consensus-building approach involving multiple iterative rounds of anonymous expert feedback. Domain experts independently evaluate a catalog or framework; their aggregated responses are then shared anonymously among participants, who are invited to revise their assessments in subsequent rounds until a stable consensus is reached. Compared to synchronous focus groups,

this method minimises direct confrontation and reduces the influence of dominant participants, making it particularly appropriate for complex or interdisciplinary catalogs where agreement across specialists from multiple domains is required to reduce conceptual ambiguity and ensure construct validity. From a logistical standpoint, the Delphi technique also offers notable advantages over synchronous methods such as workshops or expert panels: it eliminates the need to coordinate schedules and allows participants to contribute asynchronously and independently, without temporal or geographical constraints. This renders it especially well-suited to early-stage or emerging catalogs whose categories and dimensions are still evolving and require conceptual stabilisation before further empirical validation is feasible.

Action research evaluation. In action research evaluation, researchers actively intervene within an organisational environment while simultaneously observing and evaluating the effects of the proposed artefact. Applied to catalog evaluation, this approach may involve introducing a taxonomy or pattern collection into an industrial machine learning workflow and continuously studying how practitioners engage with it during real-world problem-solving activities. Unlike one-time assessments, action research enables the evaluation of not only immediate usefulness and applicability, but also longer-term dimensions such as organisational adoption, collaboration dynamics, and operational impact. This renders it particularly appropriate when the research goal extends beyond conceptual validation toward understanding how the catalog transforms practice over time.

Case study evaluation. A case study evaluation involves the in-depth investigation of the catalog applied within one or more organisations over a defined period. For reusable conceptual resources in machine learning, a case study can examine whether a catalog or taxonomy improves understanding, communication, decision-making, or classification consistency among practitioners. Compared to illustrative scenarios — the most common approach in the literature reviewed above — case studies provide richer contextual insights because they observe the artefact in realistic operational environments rather than through isolated demonstrations. This method is best suited to mature catalogs whose structural integrity is well established and where the evaluation focus shifts from conceptual validity to practical application.

Log diary evaluation. The log diary method is a longitudinal approach in which participants continuously document observations, interactions, and incidents related to the evaluated artefact during their day-to-day work activities. For example, machine learning practitioners may maintain structured logs describing encountered antipatterns, data smells, or ethical issues while using a taxonomy-based support tool. Such longitudinal observations yield valuable insights into the frequency, usability, and practical relevance of

Reference	Expert Evaluation			Illustrative Scenario	Experimental	Logical
	Expert Interview	Focus Group	Survey			
Foidl et al. [FFR22]	—	—	—	✓	—	—
Shome et al. [SCvD22]	—	—	—	✓	—	—
Zhang et al. [ZCvD22]	—	—	—	✓	—	—
Virmani and Kuppam [VK24]	—	—	—	✓	—	—
Zhu et al. [ZM23]	—	—	—	✓	—	—
Matthews [Mat20]	—	—	—	✓	—	—
Recupito et al. [RRDNP24]	—	—	—	—	✓	—
Bhatia et al. [BLR+24]	—	—	—	—	✓	—
Hahn and Sales [HS25]	—	—	—	—	✓	—
Giordano et al. [GALP23]	—	—	—	—	✓	—
Cardozo et al. [CDC23]	—	—	—	—	✓	—
Mojica-Hanke et al. [MHBLV+23]	✓	—	—	—	—	—
Nazir et al. [NBP24]	✓	—	—	—	—	—
Washizaki et al. [WUKG19]	—	—	✓	—	—	—
Washizaki et al. [WKY+22]	—	—	✓	—	—	—
Serban et al. [SKvdBHV20]	—	—	✓	—	—	—
Ximenes and Ramalho [XGLR21]	—	—	✓	—	—	—
Heli et al. [HLB+24]	—	✓	—	—	—	—
Leuthe et al. [LMHPS24]	—	✓	—	—	—	—
Indykov et al. [ISW25]	—	✓	—	—	—	—
Bogner et al. [BVG21]	—	—	—	—	—	✓
Heiland et al. [HHB23]	—	—	—	—	—	✓
Mailach and Siegmund [MS23]	—	—	—	—	—	✓
Lo et al. [LLZ+22]	—	—	—	—	—	✓
Lu et al. [LZX+24]	—	—	—	—	—	✓

Table 3.1: Overview of evaluation approaches used in related works

catalog categories in real operational settings — dimensions that single-session evaluations are typically unable to capture. This method is particularly valuable when the research objective is to understand how a catalog is used over time rather than whether it is merely applicable in principle.

Sorting evaluation. Sorting-based evaluation methods, including card sorting and Q-sorting, require domain experts to classify concepts, smells, patterns, or best practices into predefined catalog dimensions. Researchers then measure inter-rater agreement to assess the consistency, clarity, and construct validity of the taxonomic structure. These methods are particularly effective for catalogs containing a moderate to large number of items where the goal is to verify that classification boundaries align with expert mental models and that dimensions are intuitive and unambiguous to users. Sorting evaluations also produce quantitative agreement indices — such as Krippendorff’s α or Fleiss’ κ — which can serve as objective measures of construct validity and inter-rater reliability.

Chapter 4

The Catalog of Reusable Conceptual Resources

This chapter presents the Catalog of reusable conceptual resources. It first describes how the resources were collected and what their presentation in the sources looked like. It then introduces the five kinds of resources used to classify them, the template associated with each kind, and the tags that connect each resource to the activities it supports and to the qualities it affects. It also shows how the traditional types map onto the new kinds. The chapter ends with the organization of the Catalog and with the data engineering part, which was compiled in this thesis and is reported in [Appendix A](#).

4.1 Selection of Resources

The Catalog was constructed through a combination of literature-based research and Large Language Model (LLM)-assisted searches. The purpose of this process was to identify relevant resources, such as methods, frameworks, and recurring solutions, that could be considered for inclusion in the Catalog, while maintaining traceability to the sources through which each entry was identified. This traceability was maintained to support potential future analyses, such as identifying the proponents of the resources and assessing their popularity.

4.1.1 Literature and Repository-Based Search

The first source of Catalog entries was a structured search of the relevant literature and repositories. This search proceeded in two phases, distinguished by the search keys used.

In the first phase, the most relevant scientific and grey literature repositories were selected — namely:

- **Scientific literature:** Google Scholar, Scopus, IEEE Xplore, ACM Digital Library, SpringerLink, arXiv, DBLP, and ResearchGate;
- **Grey literature:** Google, Medium, LinkedIn, and Stack Overflow.

These repositories were searched using two families of search keys: (i) the names of the classical reusable resource types introduced in Chapter 2, such as *design pattern*, *architecture pattern*, *anti-pattern*, *tactic*, *best practice*, *guideline*, *pitfall*, and *technical debt*, and (ii) a checklist of candidate names of resource types, prepared before the search and grouped by similarity, which was used so that also the types not named in Chapter 2 could be reached. Each search key was used together with the term “machine learning”, and the name of the resource type was enclosed in quotation marks, as in the search string “data smell” “machine learning”. The search keys are listed in Table 4.1.

Phase	Search keys
First phase	design pattern, architecture pattern, anti-pattern, tactic, best practice, guideline, pitfall, technical debt, and the candidate names of the checklist
Second phase	ML pattern, pattern, data smell

Table 4.1: Search keys used in the two phases. Each of them was combined with the term “machine learning”.

In a second phase, additional, more specific search keys were introduced based on terminology encountered in the documents collected during the first phase. These included terms such as *ML pattern*, *pattern*, and *data smell*, among others identified while examining first-phase results.

In addition, further sources were identified through snowballing, that is, by following the references of the documents already collected.

The documents found in this way were then examined one by one to decide which resources they contributed to the Catalog. A resource was selected through the following steps:

1. Read the document and list the resources that it describes.
2. Decide whether each of them falls within the scope of the Catalog, that is, whether it is a piece of reusable knowledge for the development of ML(AI)-fueled systems.
3. Extract from the document the information needed to fill in the fields of the template.

4. Record the document from which the resource was taken.
5. Where the same resource was found in more than one document, keep a single entry and record all its sources.

The documents identified through this process were then used directly to write the corresponding Catalog entries, with each entry retaining a reference back to the source document from which it was derived.

4.1.2 LLM-Assisted Search

In addition to the literature-based search, LLMs were used as a complementary discovery mechanism. The objective of the LLM-based search was to identify potentially relevant Catalog entries that might not have been identified through the conventional literature search.

Exactly four LLMs were used during the discovery process: ChatGPT, Gemini, DeepSeek, and Claude. For each search key identified during the literature-based search (Section 4.1.1), a prompt of the following form was submitted to each model:

“I want a list of [search key] in the context of machine learning.”

where [search key] was substituted with each of the terms listed in Table 4.1. A systematic comparison of the results obtained from the LLM-assisted search against those obtained from the conventional literature search — for example, in terms of overlap, precision, or the relative contribution of each LLM — falls outside the scope of this thesis; here, LLMs are used solely as an additional discovery channel.

The LLM-based search followed a process similar to the literature search:

1. Define the topic or category for which additional resources were required.
2. Formulate a specific prompt describing the information to be identified.
3. Submit the prompt to the selected LLM.
4. Record the resources suggested by the LLM.
5. Examine the suggested resources and their descriptions.
6. Where applicable, trace the suggested resource to an external source or reference.
7. Evaluate its relevance to the Catalog.

8. Add relevant resources to the Catalog while recording their LLM provenance.

The output of an LLM was therefore treated as a discovery mechanism rather than as authoritative evidence by itself. LLM-generated suggestions were examined and, where possible, associated with supporting references or other source information before being incorporated into the Catalog.

4.1.3 Identification and Consolidation of Catalog Entries

At the current stage, all candidate resources identified through the process described above have been added to the Catalog, with two exceptions. A candidate resource was omitted when either of the following applied: (i) the resource was only cited in passing in a source document, without a definition sufficient to describe it as a standalone Catalog entry; or (ii) the resource was specific to a particular technology, library, or programming language (e.g., a PyTorch-specific mechanism, or a Python-language construct) rather than representing a generalizable conceptual resource applicable across tools and languages.

For each Catalog entry, the references to the documents in which the resource was found were recorded.

4.1.4 Final Selection

The final catalog was obtained after reviewing the candidate resources identified through both literature/repository searches and LLM-assisted searches. Candidate resources were retained when they were relevant to the scope of the catalog and sufficient information was available to describe and trace the resource.

The resulting catalog therefore combines resources identified through conventional literature research with resources discovered through LLM-assisted exploration. The use of multiple discovery mechanisms was intended to broaden the search while maintaining traceability of the origin of each catalog entry.

4.1.5 Classification of the Found Resources

The analysis of the collected sources showed that the classification of reusable resources is not stable. The same resource was often presented with different types in different sources: more than one third of the Catalog resources for which the type used in the source was recorded received at least two different types. For example, *data versioning* was presented as a best practice, an anti-pattern, and an architectural tactic, while *canary deployment*

appeared as a design pattern, a deployment pattern, a guideline, a check, an ML pattern, and an ML design pattern.

Moreover, several sources used new types specific to machine learning, such as *ML pattern*, *ML design pattern*, *data smell*, and *responsible AI pattern*, which have no classical equivalent.

These observations motivated the definition of a new classification of reusable resources based on their role. There are three reasons for it. First, the type used in the sources is not a reliable way of classifying a resource, since the same resource receives different types in different sources. Second, the classical terminology of software engineering, such as pattern, smell, tactic, and technical debt, is not very popular with data and machine learning engineers, who are the intended readers of the Catalog, and it is therefore of little help to them in finding what they need. Third, the types specific to machine learning keep growing in number, so a classification based on the types would have to be extended every time a new one appears. A classification based on the role that a resource plays avoids these three problems, because the role can be decided for every resource and does not depend on the terminology of the source. It is presented in the next section.

4.2 Kinds of Reusable Conceptual Resources

There exist many different kinds of reusable conceptual resources, as discussed in Chapter 2. Organizing these resources according to well-defined dimensions, developers and researchers can quickly identify those relevant to specific tasks or scenarios, reducing the effort required to navigate large repositories.

The kinds presented here follow from the reasons given in Section 4.1.5. They are named with simple, self-explanatory words that say directly what the resource does for the reader, rather than with the terminology of software engineering, such as pattern, smell, or technical debt.

Recognizing that a well-structured classification enhances the reusability of conceptual resources, and given the emergence of new types such as training patterns, we propose a classification approach that effectively accommodates both software engineering and machine learning contexts, as shown in Figure 4.1.

The proposed conceptual structure provides a systematic framework for organizing and reusing conceptual resources in software engineering and machine learning. The goal is to create a coherent and extensible scheme for documenting, categorizing, and reusing conceptual resources. This conceptual structure is created to ensure that a conceptual resource is uniformly described, categorized along different axes, and related to different qualitative contexts and concerns. As a result, it facilitates structured reuse and analytical

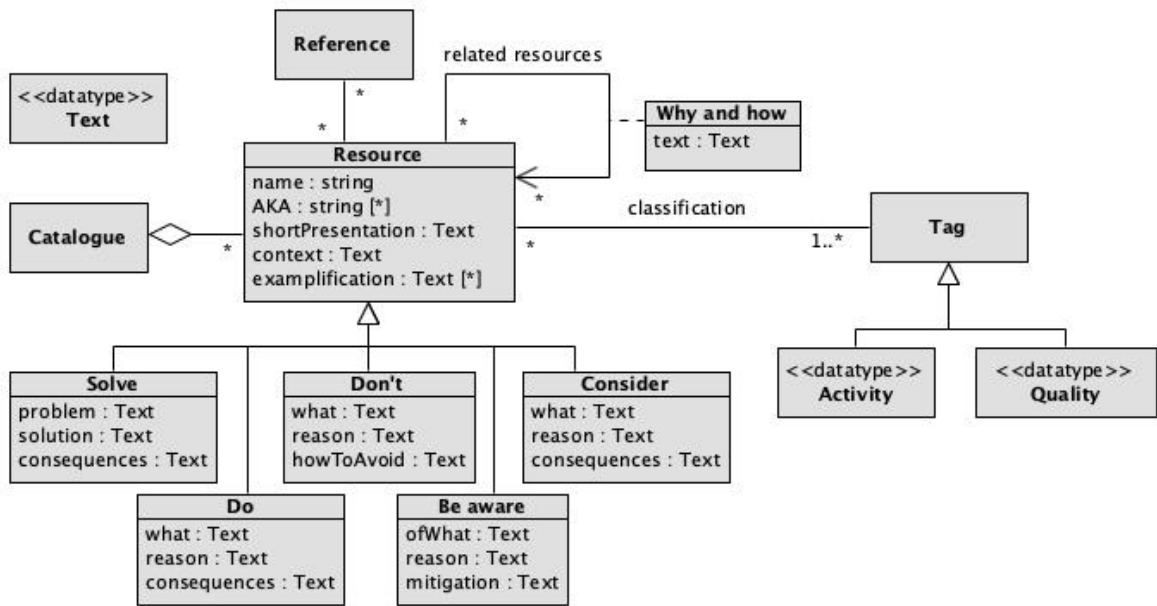


Figure 4.1: Structure of the Catalog: the resources, their kinds and templates, and the tags.

thinking about each resource.

Reusable conceptual resources vary according to the guidance roles they play in design and decision-making processes. A design pattern provides a solution, whereas a code smell provides a warning; a best practice prescribes an action, whereas an anti-pattern prohibits one. To capture this functional diversity and support consistent retrieval, this section introduces a role-based classification scheme that organizes each resource according to the guidance role it fulfills in a given context rather than its nominal type. Each role is referred to as a category, and in the current version of the Catalog every resource is assigned exactly one category. The scheme comprises five categories: Solve, Do, Don't, Consider, and Be aware. Table 4.2 provides a concise summary of all five categories before each is discussed in detail.

Assigning a single category is a choice that has still to be discussed. The same piece of knowledge can often be presented as a resource of different kinds: a practice that should be followed can be described as a Do resource, the corresponding mistake can be described as a Don't resource, and the risk that it creates can be described as a Be aware resource. When entries coming from different sources were consolidated, one kind was chosen for each resource, and the other formulations were left out. Many resources can therefore be classified in more than one way with this scheme, and the evaluation reported in Chapter 5 shows that the experts do not always agree with the kind that was chosen. Future work will investigate whether it is useful to offer the same piece of knowledge as several resources of different kinds, for example as a best practice and as the corresponding anti-pattern.

Table 4.2: Reusable conceptual resources

Category	Role
Solve	Provides a reusable, proven solution to a named recurring problem.
Do	Prescribes a practice or rule that must consistently be followed to ensure quality, not only of the produced system but also of the process and of the data used to build it.
Don't	Identifies a harmful approach that may be mistakenly adopted and must be avoided or reversed, also to avoid failures and troubles.
Consider	Suggests an available technique or strategy whose adoption depends on context and trade-offs.
Be aware	Alerts practitioners to a symptom or condition that signals existing or potential risk.

4.2.1 Solve Resources

The unique feature of a Solve resource is that it has both a clear problem and a matching full solution. Resources of this type must describe a solution that is concrete enough to be implemented without too much need for re-interpretation.

Solve resources are reactive in nature: they are triggered by the recognition of a specific problem rather than applied continuously throughout a lifecycle phase. This distinguishes them from Do resources, which prescribe actions that should be performed regardless of whether a problem has been identified. The Solve category accordingly encompasses resource types whose primary purpose is problem resolution: design patterns, architecture patterns, and deployment patterns. The template shown in table 4.3 governs the documentation of every Solve resource in the catalog. All sections of this template are mandatory. For instance, a solution with no recorded exemplification either reflects incomplete documentation or an implausible design, and neither can support reliable decision-making.

Field	Description
Name	A unique name identifying the resource
Also known as	Alternative names that improve discoverability
Short presentation	A concise description of the resource
Context	The context in which the resource is applicable
Problem	A precise statement of the recurring problem the resource resolves.
Solution	A description of the structural or procedural arrangement that resolves the problem. Should be concrete enough to applied without significant reinterpretation.
Consequences	Summarises the benefits and evaluates the net effect on the system's quality attributes.
Exemplification	One or more examples demonstrating practical usage

Table 4.3: Solve resource template

Short presentation It provides a concise, one- or two-sentence summary of the resource that immediately conveys its purpose and the problem it addresses. Unlike classical design patterns, which often begin directly with Context and Problem, we include this field to allow practitioners to quickly assess relevance in a large catalog before reading the full details.

Context It explains the circumstances in which the resource can be used. These circumstances, which capture the environment in which the resource provides its intended value, can be organizational, architectural, environmental, process-related or concerning the ML model building. Before continuing, it enables practitioners to assess whether their current circumstances are within the resource’s intended scope. Misuse is prevented by a clearly defined context. When a resource is used outside of its intended scope, it may not provide the anticipated benefits or may have unforeseen effects.

Problem It describes the recurring difficulty in terms of observable behavior and constraints — not in terms of the solution. A problem statement expressed in solution terms (“use a factory when you need to create objects”) does not provide a criterion by which a practitioner can judge whether their situation matches; a problem statement expressed in context terms (“situations where the creation of families of related objects risks inconsistency and tight coupling”) does. The Context field complements Problem by specifying the architectural or environmental conditions under which the problem arises.

Exemplification Provides one or more concrete examples that illustrate how the solution is applied in practice. The examples should be specific enough for practitioners to understand the structure and usage of the resource, typically including relevant code snippets, architectural diagrams, or real-world scenarios from software or ML systems. This field helps bridge the gap between the abstract description and its practical implementation.

Example of a Solve resource: Table 4.4 shows the Abstract Factory pattern documented using the Solve resource template, strictly following the roles and collaborations defined in the original design pattern literature.

Field	Description
Name	Abstract Factory
Also Known As	Kit
Short Presentation	Provides an interface for creating families of related or dependent objects without specifying their concrete classes, decoupling clients from product implementations.
Context	Cross-platform UI toolkits or any system where multiple families of related objects (e.g., look-and-feel themes) must coexist and remain interchangeable.
Problem	Systems that require creation of families of related objects risk inconsistency and tight coupling if clients construct concrete components directly.
Solution	Provide an interface for creating families of related or dependent objects without specifying their concrete classes. Define an abstract factory with creation methods for each type of product. Concrete factories implement these methods to produce concrete product objects from one family. Clients use only the abstract factory and abstract product interfaces.
Consequences	Decoupled, consistent product families with improved configurability and portability across environments; at the cost of a larger class hierarchy and harder extensibility for new product types.
Exemplification	In the InterViews toolkit, the <i>WidgetFactory</i> interface defines methods like <i>CreateButton()</i> . The <i>MotifFactory</i> (ConcreteFactory) implements these to return <i>MotifButton</i> objects, whereas <i>OpenLookFactory</i> returns <i>OpenLookButton</i> objects. The client code interacts only with <i>WidgetFactory</i> and the abstract <i>Button</i> interface.

Table 4.4: Abstract Factory as a Solve Resource

Consequences Describes the measurable impact of applying the resource on the system's quality attributes. This field explicitly details the effects of the solution, identifying both the improvements (e.g., enhanced maintainability) and the liabilities (e.g., increased performance overhead). To support reliable decision-making, this field is mandatory; every resource must record both its benefits and its costs, as an absence of recorded trade-offs suggests incomplete documentation.

4.2.2 Do Resources

A Do resource prescribes an action, practice, or rule that must be consistently followed within its applicable context. Do resources are proactive: they specify what practitioners must do to maintain or achieve a defined level of quality, reliability, or correctness, and their applicability is not contingent on the recognition of a specific problem. Their omission increases risk even in the absence of a manifest failure.

A Do resource does not respond to a specific problem; it governs the ongoing behavior of practitioners throughout a life cycle phase. Its purpose is preventive rather than corrective; by embedding sound practices into the routine of a life cycle phase, it reduces the likelihood that problems arise in the first place. Table 4.5 presents the template for the DO resources

Field	Description
Name	A unique name identifying the resource
Also known as	Alternative names that improve discoverability
Short presentation	A concise description of the resource
Context	The context in which the resource is applicable
What	A precise statement of the action, practice, or constraint being prescribed. Should be stated in imperative or declarative form: <i>"Always..."</i> , <i>"All components must..."</i> .
Reason	The justification for why this action is obligatory. The reason should reference a documented quality risk, a known failure mode, or an empirically validated benefit.
Exemplification	One or more examples demonstrating practical usage

Table 4.5: Do resource template

What It specifies the concrete action, practice, or constraint that the resource prescribes. It is the operational core of a Do resource, translating its intent into a direct, executable instruction. A practitioner should be able to read this field and know precisely what is expected of them, without needing to consult additional sources or exercise significant judgment. The specificity of this field is what distinguishes a well-documented Do resource from a general recommendation.

Reason It explains the necessity of the prescribed action. Knowing what to do is not always enough; practitioners who comprehend the reasoning behind an instruction are better able to apply it appropriately and identify circumstances in which it is particularly crucial. As a result, the Reason field provides evidence to support the instruction, such as

a verified benefit, a known failure mode, or a documented quality risk. It must provide an explanation that is independent of the What field; merely rewording the instruction does not qualify as an explanation.

Example of a Do resource: Table 4.6 shows the Separation of Concerns (SoC) principle described using the Do resource template, highlighting its role as a proactive and mandatory architectural guideline.

Field	Description
Name	Separation of Concerns (SoC)
Also Known As	SoC Principle; Single Responsibility at Module Level
Short Presentation	Divide a system into distinct modules, each responsible for one specific functionality or concern, to reduce complexity and allow independent development and modification.
Context	All software systems, across all abstraction levels — from class design to system architecture — especially those expected to evolve or be maintained by multiple developers.
What	Always decompose systems so that each module, component, or class encapsulates a single well-defined concern. No module should carry responsibilities that belong to another module.
Reason	Mixed concerns increase coupling, making changes in one area inadvertently break another. SoC reduces this coupling, lowers cognitive load for developers, and enables independent testing, deployment, and replacement of components. Empirical and theoretical literature in software engineering consistently identifies it as a foundation of maintainable system design.
Exemplification	Separating HTTP request handling, business logic, and database access into distinct layers in a web application so that changing the database technology does not affect the request handling layer.

Table 4.6: Separation of Concerns as a Do resource

4.2.3 Don't Resources

A Don't resource documents a practice, design approach, or structural pattern that should never be intentionally applied because it reliably produces negative consequences, including technical debt, poor maintainability, performance degradation, system fragility and ML model biasedness or drifting. The defining characteristic of a Don't resource is that harmful practice was adopted as an apparent solution, it is not a random error, but a recognizable, recurrent response to a problem that proved, in practice, to be counter-productive. Table 4.7 presents the template for the Don't resources.

Field	Description
Name	A unique name identifying the resource
Also known as	Alternative names that improve discoverability
Short presentation	A concise description of the resource
Context	The context in which the resource is applicable
What	A precise statement of the harmful action, practice, or structural arrangement that should be avoided.
Reason	The justification for why this practice must be avoided. The Reason field must identify the specific negative consequences (e.g., technical debt, degraded maintainability, reduced testability, performance failures) that the practice reliably produces, with reference to documented evidence. A Don't resource whose Reason cannot cite recurring documented harm is not a true reusable resource; it is a stylistic preference and should not be included in the catalog.
How to avoid	Provides a clear, actionable alternative or a refactoring path to correct the harmful practice.
Exemplification	One or more examples demonstrating practical usage

Table 4.7: Don't resources template

What It identifies the harmful practice, approach, or structural arrangement that must be avoided. It is the operational core of a Don't resource: it gives the practitioner a precise and recognizable description of what should never be intentionally applied, so that it can be identified before or during application. The field should be stated in concrete, observable terms so that a practitioner can recognize the practice in their own work without ambiguity. A vague or overly abstract What field undermines the purpose of the resource: if the harmful practice cannot be clearly identified, it cannot be reliably avoided.

Reason It explains why the described practice should never be used. The specific detrimental effects that the practice consistently results in, such as technical debt, diminished maintainability, decreased testability, performance failures, ML model drift or biasedness, serve as documented evidence for the prohibition. This rationale is what distinguishes a Don't resource from a stylistic preference: the Reason field must show that the harm is documented, recurrent, and caused by the practice itself rather than by coincidental events. A practitioner who is aware of the repercussions is better able to identify early warning indicators, withstand pressure to adopt the practice under time constraints, and inform others about the risk.

How to avoid It provides the "corrective" logic for the resource. While the *What* field identifies the problem, *How to avoid* outlines the refactored solution or the alternative practice that restores quality and technical integrity. This field ensures the Don't resource is not just a warning, but a constructive tool for system improvement.

Example of a Don't resource: Table 4.8 shows the Blob anti-pattern described using the Don't resource template, illustrating how a common but counter-productive design choice is documented to facilitate its avoidance and reversal.

Field	Description
Name	The Blob
Also Known As	God Class; Winnebago
Short Presentation	A single class or module accumulates the vast majority of a system's logic and control, while surrounding classes act as passive data containers with little behaviour of their own.
Context	Object-oriented systems of any size, particularly those that evolved incrementally from small prototypes without formal architectural review.
What	Never design or allow to evolve a class that concentrates most business logic, control flow, and responsibility while other classes serve only as data holders.
Reason	• Poor cohesion and excessive coupling — changes in one area break unrelated parts. • Extreme difficulty in unit testing due to monolithic size and tangled dependencies. • Very limited component reuse, as logic cannot be extracted without ripple effects. • Long build and compile times for the central module. • Rapidly growing technical debt that becomes exponentially more expensive to address over time.
How to avoid	Identify cohesive sets of operations within the Blob and migrate them to the passive data classes or new service classes. Apply the MVC pattern to separate UI logic from business logic.
Exemplification	A GUI application where one MainController class handles user input, database connections, business validation rules, and report formatting simultaneously.

Table 4.8: The Blob as Don't resource

4.2.4 Consider Resources

A Consider resource suggests a technique, strategy, or architectural decision that may be beneficial in a given context but whose adoption is neither mandatory nor problem-triggered. Consider resources are advisory and situational; they present a technique, articulate the conditions under which it is likely to be beneficial, and make explicit the trade-offs its adoption entails. Practitioners are expected to evaluate these trade-offs against their specific quality goals and system constraints before deciding whether to apply the resource. Table 4.9 presents the template for the Consider resources.

Field	Description
Name	A unique name identifying the resource
Also known as	Alternative names that improve discoverability
Short presentation	A concise description of what should be considered
Context	The context in which the resource is applicable
What	A precise description of the technique, strategy, or approach being presented, for informed evaluation and application
Reason	The justification for why this technique is worth considering, grounded in the conditions under which it has demonstrated value
Consequences	Summarises the benefits and evaluates the net effect on the system's quality attributes.
Exemplification	One or more examples demonstrating practical usage

Table 4.9: Consider resource template

What It describes the technique, strategy, or approach being presented for consideration. Unlike prescriptive resources, it does not mandate adoption: it gives the practitioner a precise and concrete understanding of what is being proposed, so that they can evaluate its fit within their own context.

Reason It explains why the described technique is worth considering. It should identify the conditions under which the technique has demonstrated value, the benefits it may deliver, and any relevant trade-offs the practitioner should be aware of before making a decision. The Reason field does not argue for unconditional adoption: its purpose is to provide the practitioner with sufficient evidence to make an informed judgment.

Example of a Consider resource: Table 4.10 shows the Heartbeat Tactic described using the Consider resource template, illustrating how a situational reliability mechanism

is documented for trade-off analysis.

Field		Description
Name		Heartbeat Tactic
Also Known As	Known	Ping/Echo; Keep alive Signal; Live-ness Check
Short Presentation	Presen-	A health-monitoring mechanism in which one component (the emitter) periodically sends a signal to a monitor; the absence of the signal within a defined interval triggers a recovery process.
Context		Distributed systems, safety-critical systems, or any system where undetected component failure is unacceptable and a monitoring infrastructure is feasible.
What		Introduce a periodic message exchange between components: the emitter sends a heartbeat signal at a defined interval; a monitor component times out and triggers recovery if no signal is received within the expected window.
Reason		It improves fault detection speed and system reliability in environments where components might stop functioning without signaling a failure.
Consequences		Improves system availability and reliability, but introduces communication overhead and network load.
Exemplification		Kubernetes uses liveness probes (a heartbeat-style mechanism) to detect and restart unresponsive application containers in a cluster.

Table 4.10: Heartbeat Tactic as a Consider resource

4.2.5 Be aware Resources

A Be aware resource identifies a symptom, indicator, or condition that signals existing or potential risk in a system, codebase, process, dataset or ML model. Be aware resources are diagnostic in function, their primary purpose is to alert practitioners to the presence of a problematic condition so that its cause can be investigated and, where necessary, addressed. They do not prescribe a structural solution; they prescribe investigation. Table 4.11 presents the template for the Don't resources.

Field	Description
Name	A unique name identifying the resource
Also known as	Alternative names that improve discoverability
Short presentation	A concise description of the concept
Context	The context in which the resource is applicable
Of What	A precise description of the symptom, indicator, or condition that signals the presence of a risk
Reason	The justification for why the identified symptom warrants attention, grounded in known quality risks or failure modes.
Mitigation	Suggests initial steps for investigating the root cause and points toward potential strategies or Solve resources to address the issue.
Exemplification	One or more examples demonstrating practical usage

Table 4.11: Be Aware resource template

Of What it describes the symptom, indicator, or condition that signals the presence of an existing or potential risk. It should be stated in concrete, observable terms so that a practitioner can recognize the symptom in their own system, codebase, process, dataset or ML model without ambiguity. The Of What field does not diagnose the underlying cause: it directs the practitioner’s attention to a condition that warrants further investigation.

Reason It explains why the identified symptom warrants attention and investigation. It should reference the known quality risks, empirical observations, or documented failure modes that the symptom commonly indicates. A well-written Reason field gives the practitioner sufficient grounds to prioritise investigation — not by confirming that a problem exists, but by establishing that the symptom is a credible signal that one may.

Mitigation This field outlines the transition from diagnosis to resolution. It provides practitioners with immediate ”next steps” to verify the symptom’s severity and suggests potential strategies for overcoming the issue. Crucially, this field often acts as a link within the catalog, pointing the practitioner toward specific Solve resources (patterns) or Do resources (practices) that can effectively eliminate the identified risk.

Example of a Be aware resource: Table 4.12 shows the Long Method design smell described using the Be aware resource template, highlighting its role as a diagnostic signal for code quality.

Field	Description
Name	Long Method
Also Known As	Bloated Method; God Method; Oversized Function
Short Presentation	A method or function that has grown so large that it is difficult to understand, extend, and modify — a symptom that it has accumulated too many responsibilities.
Context	All object-oriented and procedural codebases, especially those that have undergone iterative development with frequent feature additions.
Of What	A method that is excessively long — typically requiring a reader to scroll to understand its full logic — and that likely handles more than one distinct responsibility. The presence of comments explaining different blocks within the method is a particularly strong indicator.
Reason	Long methods correlate with reduced readability, higher defect density, and increased resistance to change. They violate the Single Responsibility Principle at the method level, making it difficult to understand, test, or reuse individual parts. Fowler and Beck[FS01] documented that long methods are among the most reliable indicators of code that needs refactoring. The symptom does not guarantee a problem exists — it raises a credible signal that investigation is warranted.
Mitigation	Perform a decomposition of the method using the <i>Extract Method</i> refactoring technique. Identify distinct responsibilities within the method and move them into smaller, well-named helper functions.
Exemplification	A processOrder() method spanning 300 lines that handles input validation, inventory checking, payment processing, email notification, and logging within a single body.

Table 4.12: Long Method as Be aware resource

4.3 Mapping Traditional Kinds of Resources to the New Taxonomy

The transition from traditional conceptual reusable resources classification to the proposed five-kind taxonomy requires a mapping that reinterprets standard terminology through the lens of intent. As shown in Table 4.13, traditional Design Patterns and Deployment Patterns are categorized as Solve resources because they offer specific, complete structural resolutions to named problems. In contrast, Best Practices and Checklists map to Do resources, as they represent mandatory, continuously applicable guidelines rather than one-time fixes. Concepts that offer situational benefits, such as Tactics and Reference Architectures, fall under Consider resources, emphasizing that their adoption is optional and dependent on trade-off analysis. Furthermore, the taxonomy clarifies the distinction between prohibited actions and warning signals; Anti-Patterns and Mistakes are classified as Don't resources (intentional bad choices), whereas Code Smells, Technical Debt, and Pitfalls are categorized as Be aware resources. This distinction is critical because "Be aware" items are symptoms or risks that emerge from valid design decisions, whereas "Don't" items represent choices that should never be made. This mapping provides a comprehensive bridge between classical architectural literature and a more operational, decision-oriented framework.

Table 4.13: Mapping of Traditional Architectural Kinds to the 5-Kind Taxonomy

Traditional Kind	New Kind	Rationale
Design Pattern	Solve	Provides a specific, complete structural answer to a named design problem.
Architectural Pattern	Solve	Resolves fundamental structural problems (e.g., scalability) completely.
Deployment Pattern	Solve	Provides a specific structural solution to deployment problems (e.g., Blue/Green).
Best Practice	Do	Considered standard/mandatory in a context (e.g., Versioning).
Checklist	Do	Represents a set of mandatory, continuously applicable rules.
Rule	Do	Represents a strict, continuously applicable requirement (e.g., "Always validate input").
Guideline	Do	In a mature governance context, guidelines are treated as mandatory "Do" resources to ensure consistency.
Tactic	Consider	Improves quality but does not resolve the whole problem; requires trade-off analysis.
Reference Architecture	Consider	A starting point that accelerates development but is not obligatory.
Anti-Pattern	Don't	Represents an intentional design choice that should never be made.
Mistake	Don't	An action that should have been avoided; maps to a prohibited action.
Smells	Be aware	A symptom or warning signal that triggers investigation.
Technical Debt	Be aware	A state or condition of the system that accumulates risk over time.
Pitfall	Be aware	A context-specific risk condition to look out for.
Failure	Be aware	An event or symptom indicating a breakdown.

4.4 Classification and Tagging

To ensure efficient retrieval and multi-axis querying, every resource in the catalog is characterized by a set of **Tags**. A Tag is a metadata descriptor that allows the catalog to be filtered by role, by life cycle phase, and by quality concern simultaneously. Table 4.14 summarizes the two primary dimensions of the tagging system.

Dimension	Description
Activity	Identifies the engineering domain and lifecycle stage where the resource is applicable (Support).
Quality	Identifies the software, model, or data characteristics influenced by the resource (Affect).

Table 4.14: The dimensions of the tagging system.

In the current version of the Catalog a third dimension is also used, which records the techniques a resource refers to, such as deep learning or large language models. The tagging system follows a hierarchical structure; applying a specialized tag (e.g., *Data Labelling*) implicitly associates the resource with its parent domain (e.g., *Data Engineering*). This allows for both broad and granular searches within the repository.

4.4.1 Activity Tags (Support)

The Activity dimension identifies the engineering activities to which the reusable conceptual resource applies.

In the catalog, Activity is modeled as a classification dimension composed of multiple engineering domains. These domains include general software engineering activities (such as architecture, coding, deployment, development processes, and testing), as well as specialized activities related to data engineering, machine learning engineering, and management practices such as governance and project management.

The purpose of Activity is to enable context-oriented reuse. By associating resources with specific engineering tasks or phases, the catalog allows users to identify which conceptual resources are relevant to their current development activity or decision-making context.

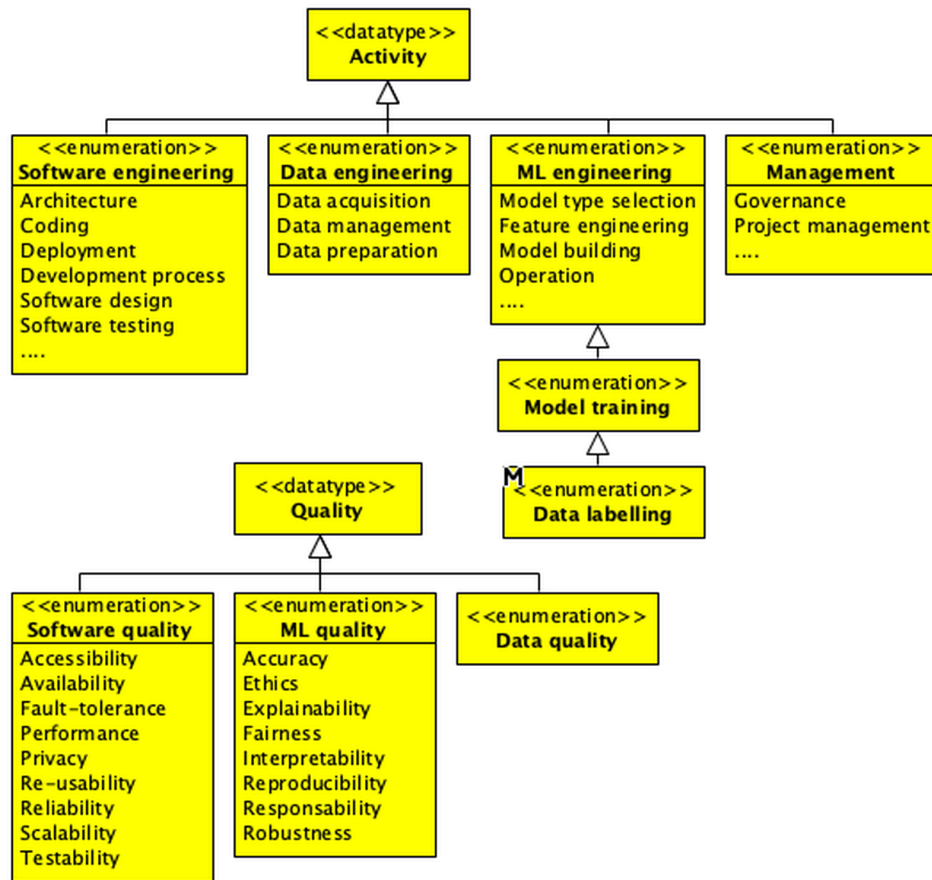


Figure 4.2: Current structure of the tags. It is still incomplete for the parts of the Catalog other than data engineering.

Table 4.15 lists the activity tags, grouped by area, following the structure shown in Figure 4.2.

Software engineering	
Architecture	Defining the high-level structure, components, and interactions of the system.
Coding	Implementing system functionality through programming.
Deployment	Releasing and configuring the system in a production environment.
Development process	Defining and managing the methodologies, workflows, and lifecycle stages of the system development project.
Maintenance	Updating, fixing, and improving the system after deployment.
Software design	Designing modules, interfaces, and system components to meet requirements.
Software testing	Verifying and validating that the system meets requirements and behaves correctly.
Data engineering	
Data acquisition	Gathering, importing, and integrating raw data from various internal or external sources into storage systems or pipelines.
Data management	Organizing, storing, and tracking different versions of datasets over time to ensure reliability and reproducibility.
Data preparation	Detecting and correcting errors, transforming formats, combining sources, and ensuring data correctness and consistency for analysis and modelling.
ML engineering	
Data labelling	Assigning labels to the data used to train and evaluate a model.
Feature engineering	Creating and selecting meaningful input features for model training.
Model building	Defining and tuning the structure of the model, including its hyper-parameters.
Model training	Learning model parameters from training data.
Model type selection	Choosing appropriate algorithms or model types for a given task.
Operation	Managing the end-to-end lifecycle of ML models in production, including deployment, scaling, versioning, and continuous monitoring.
Management	
Governance	Defining policies, standards, and oversight mechanisms for the development and the operation of the system and for the data that it uses.
Project management	Planning, coordinating, and controlling project activities.

Table 4.15: Activity tags, grouped by area.

4.4.2 Quality Tags (Affect)

The Quality Tag identifies the quality dimensions that a reusable conceptual resource influences. A quality dimension represents a quality characteristic of a system, model, or dataset that can be affected by the reuse of the conceptual resource.

In the catalog, Affect is modeled as a classification dimension comprising three main categories: software quality, model quality, and data quality. Each category captures the relevant quality aspects within its domain—for example, software quality may include maintainability and reliability; model quality may include explainability, interpretability, and reproducibility; and data quality may include completeness and consistency.

The purpose of Quality is to support quality-driven evaluation and decision-making. It allows users to analyze how a resource contributes to achieving desired quality goals or to assess potential trade-offs among competing quality attributes.

Table [4.16](#) lists the quality tags, grouped by domain.

ML quality	
Accuracy	Degree to which model predictions correctly reflect true outcomes.
Ethics	Adherence to moral principles such as fairness, accountability, and respect for user rights.
Explainability	Degree to which model mechanisms can be understood by a human expert.
Fairness	Extent to which a system produces unbiased outcomes across groups.
Interpretability	Degree to which a human can understand the cause of a model decision.
Reproducibility	Ability to obtain consistent, matching results when an experiment, model, or analysis is repeated, either by the original researchers or by others, using the same, or clearly specified, code, data, and experimental conditions.
Responsibility	Ensures accountability, fairness, and ethical compliance in system design and operation.
Robustness	Ability of a trained model to maintain reliable predictive performance when facing noisy, corrupted, or out-of-distribution input data at inference time.
Trustworthiness	Degree to which a system is reliable, safe, fair, and aligned with human values.
Software quality	
Accessibility	Degree to which a system can be used by people with diverse abilities and needs.
Availability	Proportion of time the system is functional and accessible.
Fault-tolerance	Capability of a system to continue operating correctly in the presence of failures or faults.
Maintainability	Ease of modifying the system to fix defects, improve performance, or adapt to change.
Performance	Efficiency of a system in terms of response time, throughput, and resource utilization.
Privacy	Protection of personal or sensitive data from unauthorized access or misuse.
Re-usability	Extent to which components, data, or models can be used across different systems with minimal modification.
Reliability	Ability to perform required functions under stated conditions for a specified period.
Robustness	Ability of a system to handle unexpected, malformed, or erroneous data without crashing or producing incorrect results.
Scalability	Ability of a system to handle increasing workloads, such as more users, requests, or data, by adding resources while maintaining acceptable performance.
Testability	Ease of establishing test criteria and executing tests.
Data quality	

4.5 Catalog Structure

The Catalog presents all the collected resources using the kinds, the templates, and the tags introduced in the previous sections. Its current version is organized in parts, one for each development activity. The first ten parts cover general software engineering activities: development process, requirement engineering, design, system architecture, coding, testing, deployment, project management, operation, and governance. They are followed by a part on data engineering and by seven parts on activities specific to machine learning: model type selection, feature engineering, model training, model testing, data labelling, model architecture, and model building. Before these parts, a foreword defines the qualities used as quality tags.

Each part starts with a summary table that lists its resources, together with the activities they are related to and the qualities they affect, that is, their activity and quality tags. After the table, each resource is described using the template of its kind (Solve, Do, Don't, Consider, or Be aware). Besides the fields of its template, a description can include the other names of the resource (*aka*), exemplifications, the related resources, and the references to the sources in which the resource was found. Some descriptions also include a figure, for example to show the structure of an architecture.

The data engineering part, which is the focus of this thesis, is presented in the next section.

4.6 The Data Engineering Part of the Catalog

The data engineering part of the Catalog was compiled in this thesis. It contains 47 resources concerning the quality, the preparation, and the management of the data used by ML(AI)-fueled systems. All of them are tagged with the data engineering activity, and some of them also with governance or model building. Most of them affect data quality.

The resources belong to all five kinds: 26 Be aware, 12 Do, 5 Don't, 2 Solve, and 2 Consider resources. The Be aware resources describe data smells, that is, problems in the values or in the structure of a dataset, such as *ambiguous date/time format*, *dummy value*, and *synonyms in data*. They are mainly based on studies of data smells [RRDNP24, FFR22, SCvD22]. The Do resources present good practices for data management, such as *data profiling*, *data versioning*, and *document the dataset*, while the Don't resources describe practices to avoid, such as *ETL monolith* and *over-aggressive filtering*. Finally, the Solve resources present solutions to recurring problems, namely *data pipeline* and *NoSQL*, and the Consider resources present options to evaluate, namely *incremental data processing* and *data retention*.

The complete data engineering part is reported in Appendix A, where Table A.1 lists its

resources together with their activity and quality tags.

Chapter 5

Catalog Evaluation

This thesis focuses exclusively on the data-related portion of the catalog. This scope was established at the outset of the research, and the results of the evaluation will inform decisions about how to proceed with the remaining parts of the catalog in future work.

5.1 Evaluation Dimensions

The evaluation of each catalog resource is conducted across multiple dimensions, each targeting a distinct aspect of its quality and relevance. The first three dimensions concern the resource itself, while the last two concern its presentation in the Catalog.

Usefulness refers to the practical value of the resource for software developers and data practitioners working on ML(AI)-fueled systems. A resource is considered useful if it supports practitioners in identifying, avoiding, or resolving recurring issues in real-world projects.

Popularity captures how widely the resource is already known among experienced professionals. It is assessed by asking the experts whether they were already familiar with the resource. A resource that is already well known is not for this reason irrelevant, since in classical software engineering widely known ideas are also recorded as patterns or as best practices, but knowing how widely a resource is diffused helps to understand which parts of the Catalog carry knowledge that is less common.

AI deprecation and enforceability examines whether the resource has been rendered obsolete by recent advances in AI, or conversely, whether the issue it addresses can now be readily detected or resolved through AI-based tooling.

Clarity refers to the degree to which the description of the resource is understandable, so that a practitioner can grasp what the resource is about without needing further explanation.

Level of detail concerns the amount of information given in the description of the resource, which should be neither too concise, and therefore incomplete, nor too detailed, and therefore verbose.

5.2 Catalog Evaluation Method

What is evaluated here is the content of the data engineering part of the Catalog: the resources that it collects and the descriptions that present them. The classification itself, that is, the definition of the five kinds and of the tags, is not the object of this evaluation. The question is whether the resources are worth collecting and whether their descriptions allow a practitioner to use them.

This cannot be decided from the literature, because the resources were taken from the literature in the first place. It has to be decided by people who develop ML(AI)-fueled systems and who can say whether a resource matches what they meet in their work. The evaluation is therefore based on the judgement of domain experts, asked about every resource of the part, one resource at a time.

The method designed for this purpose is an expert-based Delphi study. Experts answer a questionnaire covering all the resources; the resources are then revised according to their answers, and the revised version is submitted again, until no substantial new comment is raised. The Delphi technique was chosen because it collects the judgement of several experts without requiring them to meet: each of them answers independently, at the time that suits them, which matters here because the experts work in different organizations and countries. Since the resources are short and can be judged one by one, no sampling was applied and all the resources of the data engineering part were submitted.

Only the first round of this design could be carried out within the time available for the thesis. The revision of the resources and the further rounds are left to future work, so what is reported in Section 5.3 are the answers of a single round.

The experts were chosen among people working on data management and on the devel-

opment of ML(AI)-fueled systems at the University of Genova. Five of them were invited and four took part. One is a professor whose research concerns data management, and in particular the integration of data sources, query processing, and the heterogeneity and the evolution of data. The other three are PhD students of the same department, working on databases and on data-intensive systems; one of them worked as a data engineer in a company before starting the doctorate. The panel therefore combines a senior researcher of the area covered by this part of the Catalog, three researchers at the beginning of their career, and one direct experience of data engineering in industry.

All the members of the panel belong to the same university, and only one of them has worked in industry. This has to be kept in mind when reading the answers, because the first question of the questionnaire asks whether a resource is useful for people developing ML(AI)-fueled systems, and a panel of this composition judges that question mainly from an academic position. The labels Expert 1 to Expert 4, used in Section 5.3 and in Appendix B, are assigned in an arbitrary order.

The questionnaire is organized in two parts. The first part presents each resource through its kind and its short presentation, and asks three questions: whether the resource is useful for people developing ML(AI)-fueled systems, whether the expert was already familiar with it, and whether the advent of AI will affect its usefulness. The second part presents the complete description of each resource, written with the template of its kind, and asks whether the presentation is clear and whether its level of detail is appropriate, leaving space for an optional comment.

The questionnaire was built from the version of the Catalog available at the time of the evaluation, which contains, besides the resources described in Appendix A, the data-related architectures and a few further resources added later.

The questionnaire was distributed as a fillable PDF form, so that the experts could answer it on a computer or on a phone and return the completed file. Four of the five invited experts returned it. Their answers are reported in full in Appendix B.

5.3 Results

Four of the five invited experts returned the questionnaire. All four answered the first part, which concerns the resources themselves. Three of them also answered the second part, which concerns the complete descriptions of the resources; the fourth reported that he could not complete it within the time he had available. A few questions were left unanswered, in most cases because the expert preferred to write a comment instead of choosing one of the options.

The results are reported as proportions of the answers that were actually given. With four

experts they indicate tendencies and point at the entries that have to be revised; they are not statistically significant values, and no measure of agreement between the experts is computed. Tables 5.1 and 5.2 give the overall answers, and the findings are presented below in decreasing order of relevance for the Catalog.

Is the resource useful for people developing ML(AI)-fueled systems?	
yes	97%
no	3%
Were you already familiar with this resource?	
yes	94%
no	6%
Will the advent of AI affect the usefulness of this resource?	
it will be easier to use	62%
it will be as before	35%
it will become useless	3%

Table 5.1: Answers of the four experts on the resources.

Is the presentation of this resource clear?	
yes	85%
no	15%
The presentation of this resource	
has the right level of detail	83%
is too concise	14%
is too detailed	4%

Table 5.2: Answers of the three experts on the complete descriptions.

5.3.1 The obligation expressed by the Do and Don't resources and the way they are named

This is the finding with the largest consequences for the Catalog, because it concerns the kinds and not single entries.

The answers of the first part are almost the same for all the kinds: whichever kind a resource belongs to, it was judged useful in between 94% and 100% of the answers and easier to use with the support of AI in between 61% and 63%. The answers of the second part are not uniform in the same way. Figure 5.1 reports them by kind and shows that

the descriptions the experts found less satisfactory are concentrated in two kinds, *Do* and *Don't*. The descriptions of the *Do* resources were judged clear in 25 answers out of 35 and appropriate in their level of detail in 22 out of 35, and those of the *Don't* resources were judged clear in 15 answers out of 23. The descriptions of the *Be aware* resources, which are the most numerous, were judged clear in 74 answers out of 79.

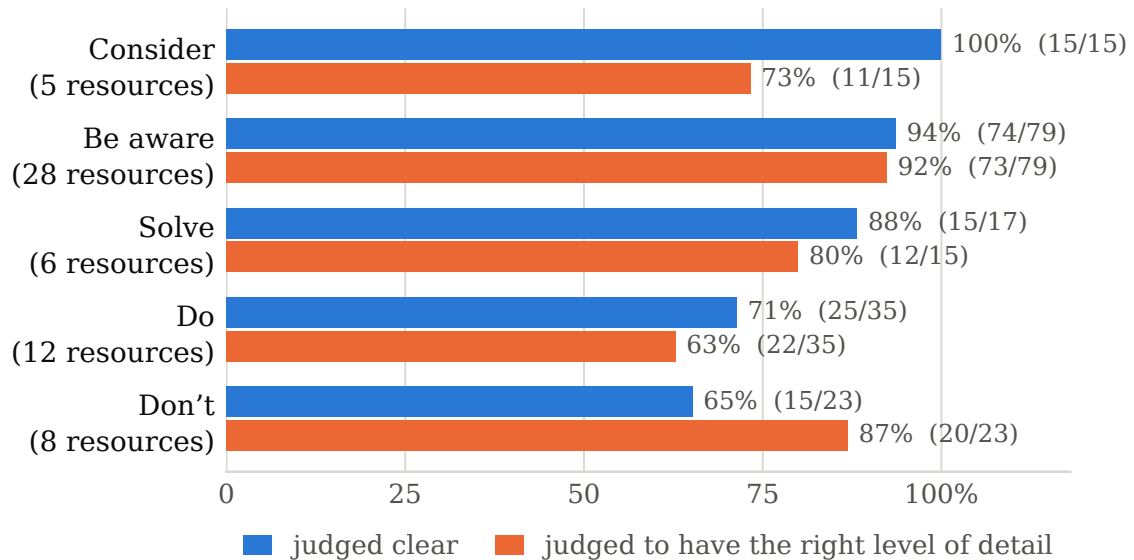


Figure 5.1: Answers of the three experts on the complete descriptions, by kind of resource. The number of resources of each kind is given next to its name.

The written comments show that these two kinds were criticized in two opposite ways. Some entries were found too prescriptive: *process raw data only once* and *remove outliers* were reported as not clear and as too concise by two experts, and one comment observes that outliers “are not always a problem to be removed”, while another observes that schema on read “is a different mode, not a don’t”. These entries state an obligation that the experts consider dependent on the context. Other entries were found not prescriptive at all: a comment on the *Do* resources observes that they “seem more definitions than DO patterns”, and *NoSQL* and *data cleaning* were described as definitions rather than as reusable resources.

The two criticisms have a common origin in the way the entries are named. Of the twelve *Do* resources, six are named with a verb phrase that says what to do, such as *check columns names* and *ensure data quality*, while the other six are named with a noun phrase that only names a subject, such as *data cleaning*, *data profiling*, and *data governance*. The *Don't* resources are divided in the same way: some name the practice to avoid, such as *ETL monolith* and *schema on read*, and others are expressed as a prohibition, such as *no*

duplicate data. A resource named with a noun reads as the definition of a subject and not as something to be done, which is what the first comment reports; a resource named with a bare imperative reads as a rule without exceptions, which is what the second comment reports.

The Catalog should therefore name the *Do* and the *Don't* resources uniformly with verb phrases, so that the name itself says what the reader is expected to do or to avoid, and the degree of obligation should be stated in the description of the entries that hold only under certain conditions, or such entries should be moved to the *Consider* kind, which expresses a weaker obligation.

5.3.2 The relations between resources are not recorded

Two experts asked how *no duplicate data* differs from *no data duplication*. The two resources are distinct: the first concerns identical records inside one dataset, the second concerns the same data held in several systems or locations. Neither of them, however, refers to the other, so a reader who meets one of them cannot see where its boundary lies. The same applies more generally: several comments ask for the relations between resources to be made explicit, because a resource is often understood only together with the ones that overlap with it.

The template of the resources already provides a *Related resources* field, but at the time of the evaluation it was filled in only in two of the entries of the data engineering part, and the relations it recorded pointed in one direction only. What the experts report is therefore not an error in the resources but the consequence of a field that had been left empty. Following this finding the field has been filled in for all the entries that overlap, and every relation is now recorded in both of the resources that it connects. In particular *no duplicate data* and *no data duplication* now refer to each other and state how they differ.

5.3.3 The resources are confirmed, their relation to AI is not

The resources themselves were confirmed. They were judged useful almost without exception, and only five received at least one negative answer: *long data value*, *meaningless value*, *process raw data only once*, *remove outliers*, and *schema on read*. Most of them were also already known to the experts; the thirteen unknown to at least one expert are mainly the data-related architectures, such as *deep lake architecture*, *data mesh architecture*, and *medallion architecture*, which are the most recent additions to the Catalog.

The answers on the advent of AI are less settled. In most answers the resource was judged easier to apply with the support of AI, and in about one third of them unaffected by it; the few answers expecting a resource to become useless concern six resources describing

data quality problems that automated tools can detect, such as *special characters in data* and *strings in human-friendly formats*. Figure 5.2 shows how often the four experts gave the same answer on the same resource. On usefulness they agree on 53 resources and differ on 5, and on familiarity they differ on 13, which is expected since they come from different backgrounds. On the advent of AI they agree on 11 resources only and differ on the remaining 47, and in 41 of these the difference is between “it will be easier to use” and “it will be as before”. The disagreement therefore concerns how much AI helps and not whether the resource survives, but this dimension is the least consolidated of the three and the answers collected here are not sufficient to draw a conclusion on it.

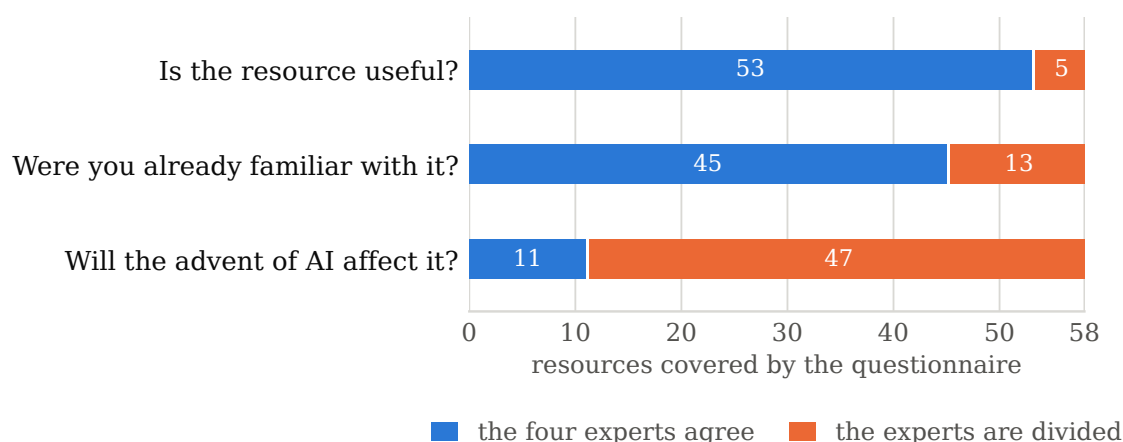


Figure 5.2: Number of resources on which the four experts gave the same answer, for each question of the first part of the questionnaire.

5.3.4 A fault in the questionnaire document

Two experts independently reported that *ETL monolith* and *special characters in data* appear twice in the material they received. Each of these resources occurs once in the Catalog, so the repetition was introduced when the questionnaire document was produced and does not affect the Catalog itself. It is recorded here because it costs the experts part of their effort, and the generation of the questionnaire has to be checked before a further round.

5.3.5 Changes suggested by the evaluation

The answers suggest four changes for the next version of the Catalog, in the order in which they should be applied: to name the *Do* and the *Don't* resources uniformly with verb

phrases and to state the degree of obligation of the entries that depend on the context, moving to *Consider* those that are not obligations; to record the relations between the entries that overlap, which has already been done for this part; to make the level of detail of the descriptions more uniform, since the answers reporting a description as too concise are almost four times as many as those reporting it as too detailed; and to correct the production of the questionnaire before submitting it again. The evaluation dimensions and the questionnaire proved usable in practice, so the same instrument can be applied to the other parts of the Catalog.

Chapter 6

Conclusion

6.1 Summary of Contributions

This thesis addressed the heterogeneity of the reusable conceptual resources that support the development of ML(AI)-fueled systems. Its main contributions are the following.

First, the resources collected from the literature and from the LLM-assisted search were analysed with respect to the types used to present them. This analysis showed that the classification adopted in the sources is not stable: more than one third of the resources for which the type used in the source was recorded received at least two different types, and several sources introduced new types specific to machine learning, such as *ML pattern* and *data smell*.

Second, the thesis contributed to the definition of the Catalog: a classification of the resources based on the role they play, with five kinds (Solve, Do, Don't, Consider, and Be aware) and a template for each kind, together with a tagging system based on the supported activities and on the affected qualities.

Third, the data engineering part of the Catalog was compiled. Its 47 resources, which concern the quality, the preparation, and the management of the data used by ML(AI)-fueled systems, were described with the template of their kind, tagged, and linked to the sources in which they were found. The remaining parts of the Catalog were revised as well.

Finally, the evaluation of the data engineering part was designed and a first round of it was carried out: the evaluation dimensions were defined, a questionnaire covering all the resources of this part was prepared and submitted to domain experts, and the answers of the four experts who took part were analysed. The resources were judged useful and, for the most part, already known, and the advent of AI was not expected to make them obsolete. The analysis also identified what has to be revised: the obligation expressed

by the Do and Don't kinds and the way their resources are named, the relations between overlapping resources, which the template provides for but the entries had left empty, and a group of descriptions considered too concise.

6.2 Limitations and Future Work

The work presented in this thesis has some limitations. The Catalog covers the development of ML(AI)-fueled systems as a whole, but only its data engineering part was considered in this thesis.

The classification of a resource into one of the five kinds, as well as the choice of its tags, requires interpretation. This is particularly evident when the sources present the same resource with different types, so different people may classify some resources differently.

The resources were collected from scientific and grey literature and through an LLM-assisted search. A systematic comparison between the results of the two searches was outside the scope of this thesis, so the contribution of the LLM-assisted search to the content of the Catalog was not measured.

Finally, four experts took part in the evaluation. This number was sufficient to identify the entries that need to be revised, as the convergence of the answers on a small group of resources shows, but it is too small to draw general conclusions on the usefulness of the Catalog, and it does not allow any statistical analysis of the answers. The evaluation also covered a single round: the resources were not revised and submitted again, so the convergence that a Delphi study aims at was not reached. A further limitation concerns the composition of the panel, whose members are all drawn from a single university and only one of whom has worked in industry. The Catalog is addressed to people who develop ML(AI)-fueled systems in industry, and their judgement on the usefulness of the resources may differ from the one collected here.

Future work follows from these limitations. The changes suggested by this round will be applied to the resources and, where needed, to their classification, and a further round will be run on the revised version, with a larger panel that includes practitioners working in industry. The same evaluation can then be applied to the other parts of the Catalog, which can also be extended with further resources. A further direction is the systematic comparison between the resources found through the literature and those suggested by the LLMs.

Appendix A

Data engineering

Asset name	Related to	Affected qualities
Ambiguous date/time format	Data engineering	Data quality
Ambiguous value	Data engineering	Data quality
Check columns names	Data engineering	Data quality
Column header containing value	Data engineering	Data quality
Data cleaning	Data engineering	Data quality
Data governance	Data engineering Governance	
Data inconsistencies	Data engineering	Data quality
Data miscoding	Data engineering	Data quality
Data pipeline	Data engineering	Reproducibility Reusability Maintainability Scalability Fault-Tolerance
Data profiling	Data engineering	Data quality
Data unit measure unknown	Data engineering	
Data retention	Data engineering Governance	
Data usage consent	Data engineering Governance	
Data versioning	Data engineering	Data quality
Document the dataset	Data engineering Governance	Reproducibility Explainability Maintainability
Dummy value	Data engineering	Data quality
Ensure data quality (Garbage in, Garbage out)	Data engineering	Data quality
ETL monolith	Data engineering	
Extraneous value	Data engineering	Data quality
Extreme data value	Data engineering	Data quality
Incorrect column header	Data engineering	Data quality
Intermingled data type	Data engineering	Data quality
Incremental data processing	Data engineering	Performance
Long data value	Data engineering	Data quality
Make data sets available on shared infrastructure (private or public)	Data engineering	
Meaningless value	Data engineering	Data quality
Missing values (and meta-data)	Data engineering	Data quality
Multi value data	Data engineering	
No data duplication	Data engineering Governance	
No duplicate data	Data engineering	
NoSQL	Data engineering	Performance
Over-aggressive filtering	Data engineering	Data quality
Process raw data only once	Data engineering Model building	
Remove outliers	Data engineering Model building	
Responsible data management	Data engineering	Responsibility Ethics
Schema on read	Data engineering	
Small number	Data engineering	Data quality
Special characters in data	Data engineering	Data quality
Split value data	Data engineering	Data quality

BE AWARE – Ambiguous date/time format

A date or time value is stored or recorded in a format that is ambiguous, inconsistent, or non-standardized across the dataset, making it unclear how the value should be interpreted.

Of what: Date and/or time attribute columns in a dataset, where the ambiguity stems from the absence of a declared format standard, making the same value interpretable in multiple ways (e.g., `01/02/2020` could represent MM/DD/YYYY or DD/MM/YYYY formats).

Reason: Obvious

Mitigation: Standardize all date/time values to a single, unambiguous format (e.g., ISO 8601:

YYYY-MM-DDThh:mm:ss) during data preprocessing; apply format validation rules at ingestion time; use dedicated date/time data types rather than free-text strings.

Exemplifications: A column `record_date` contains values such as `01/02/03`, `2003-02-01`, and `Feb 1, 2003` all intended to represent the same date, but are parseable differently depending on the assumed convention.

Related resources:

- **Data cleaning:** because resolving inconsistent or ambiguous date and time formats is one of the formatting corrections carried out during data cleaning.
- **Ensure data quality:** because ambiguous date and time formats make temporal ordering and arithmetic unreliable across records, and therefore lower the quality of the data.

References: [[RRDNP24](#), [FFR22](#)]

BE AWARE – Ambiguous value

A data value is recorded in a way that can be interpreted in multiple ways, making its actual meaning unclear without additional context.

Of what: Any categorical, numerical, or coded attribute in a dataset where the semantic meaning of a value cannot be unambiguously determined from the data itself or its accompanying metadata.

Reason: Ambiguous values undermine the reliability of data-driven systems because they introduce semantic noise that is invisible to automated tools, causing models to learn from misinterpreted features and leading to incorrect or misleading predictions.

Mitigation: Provide and maintain a data dictionary documenting all value encodings and their meanings; replace cryptic or abbreviated values with self-explanatory ones; enforce controlled vocabularies or enumerated types at data collection time; perform data audits when integrating data from multiple sources.

Exemplifications: A column `status` contains values `0`, `1`, and `2` with no accompanying documentation, leaving it unclear whether they encode {inactive, active, pending}, {low, medium, high}, or some other domain-specific mapping.

Related resources:

- **Data cleaning:** because a value that cannot be interpreted unambiguously is a direct target of data cleaning, which must resolve it.
- **Ensure data quality:** because ambiguous values lead to incorrect analysis or model training if they are not resolved, and therefore threaten data quality.

References: [[RRDNP24](#), [FFR22](#)]

DO – Check columns names

Ensure that column names in datasets are consistent, descriptive, and standardized to prevent ambiguity and errors in data processing and machine learning workflows.

Context: Data management and preprocessing in machine learning and data-driven systems, where datasets are collected, integrated, and prepared from multiple sources. This is particularly relevant in scenarios involving data cleaning, feature engineering, and dataset merging, where inconsistent or poorly defined column names can lead to misinterpretation and processing errors.

What: Systematically verify and standardize column names across datasets by enforcing naming conventions, removing ambiguities, resolving duplicates, and ensuring consistency in formatting (e.g., casing, separators, and abbreviations). This includes aligning semantically equivalent columns and ensuring that names accurately reflect the underlying data.

Reason: Inconsistent or unclear column names are a common form of data quality issue that can lead to incorrect data interpretation, faulty feature engineering, and errors in data integration. Clear and standardized naming improves data understandability, reduces the risk of misalignment during joins or transformations, and enhances Maintainability and collaboration.

Consequences: Applying column name checks improves data consistency, interpretability, and integration across datasets, leading to more reliable preprocessing and model development. It reduces errors in data transformations and feature selection, and facilitates collaboration among practitioners.

However, enforcing naming standards may require additional preprocessing effort and coordination, especially when integrating heterogeneous data sources.

Exemplifications: For example, a dataset may contain columns such as `UserID`, `user_id`, and `uid` referring to the same concept. Without standardization, these inconsistencies can lead to incorrect joins or duplicated features. By enforcing a consistent naming convention (e.g., `user_id`), such issues can be avoided, improving data quality and downstream model performance.

References: [Til21, RRDNP24]

BE AWARE – Column header containing value

Column headers that contain data values instead of attribute names represent a structural data quality issue that hinders proper data interpretation and processing.

Of what: Be aware of datasets in which column names encode actual data values (e.g., years, categories, or measurements) instead of representing stable attributes.

Reason: Column headers containing values should be carefully monitored because they obscure the true structure of the data and lead to incorrect interpretation and processing. This issue complicates data transformation, aggregation, and analysis, and may introduce errors during feature engineering or model training. It also reduces data consistency and interoperability, making integration with other datasets more error-prone.

Mitigation: Restructure the dataset to follow a normalized, tidy format by transforming column headers containing values into row-level entries. This can be achieved through operations such as unpivoting or melting, in which variable names are separated from their values, and appropriate attribute columns are introduced. Enforcing schema design standards and automated validation checks can further prevent this issue.

Exemplifications: For example, a dataset may include columns such as **Sales_2019**, **Sales_2020**, and **Sales_2021**. In this case, the year is encoded in the column header rather than represented as a separate attribute. A corrected structure would include columns such as **Year** and **Sales**, where each row corresponds to a specific year-value pair.

Related resources:

- **Ensure data quality:** because headers that accidentally contain data corrupt the structural integrity of the dataset and make it unreliable for automated processing.

References: [\[RRDNP24\]](#)

DO – Data cleaning

Data cleaning is the process of detecting, correcting, or removing errors, inconsistencies, and inaccuracies in datasets to improve data quality and ensure reliable machine learning outcomes.

Context: Data preprocessing and data management stages in machine learning and data-driven systems, where raw data from diverse sources must be prepared before analysis or model training. This is particularly critical in real-world datasets that are noisy, incomplete, or inconsistent.

What: Identify and address data quality issues by handling missing values, correcting inconsistencies, removing duplicates, standardizing formats, and filtering outliers or erroneous records. This may involve both automated procedures and domain-informed decisions to ensure the dataset accurately represents the underlying phenomenon.

Reason: Data cleaning is essential because machine learning models are highly sensitive to data quality. Poor-quality data can lead to misleading patterns, biased models, and unreliable predictions. Ensuring clean data improves the validity of insights and reduces the risk of errors propagating through the pipeline.

Consequences: Applying data cleaning improves data consistency, model performance, and robustness, and reduces the likelihood of overfitting or underfitting caused by noise and errors in the data. It also enhances interpretability and trust in model outputs. However, it may require significant effort and domain expertise, and excessive or improper cleaning can remove valuable information or introduce bias.

Exemplifications: For example, a dataset used for training a predictive model may contain missing values, duplicate records, and inconsistent formats (e.g., different date formats or units). By imputing missing values, removing duplicates, and standardizing formats, the dataset becomes more reliable, leading to improved model accuracy and stability.

Related resources:

- **Ambiguous value:** because ambiguous values represent a direct target of data cleaning. When a value cannot be interpreted unambiguously data cleaning must resolve it.

- **Ambiguous date/time format:** because inconsistent or ambiguous date and time formats are among the most common formatting errors addressed during data cleaning.
- **Data miscoding:** because values recorded using the wrong type or encoding scheme must be detected and converted to their correct representation during data cleaning.
- **Dummy value:** because dummy values masquerade as valid data but represent missing or unknown information. Data cleaning must identify all such sentinel values and fix them.
- **Missing values (and metadata):** because handling missing values is one of the core tasks of data cleaning. Decisions about whether to impute or drop missing entries directly affect dataset completeness and the validity of any model trained on it.
- **Multi value data:** because fields containing multiple values packed into a single cell violate atomicity and must be normalized during data cleaning.
- **Ensure data quality:** because data cleaning is the central mechanism through which the data quality principles operationalized.
- **Meaningless value:** because values that do not have meaning must be identified and either removed or replaced during data cleaning.
- **Data inconsistencies:** because data cleaning is the primary operational response to inconsistencies detected in a dataset. While **Data inconsistencies** describes the problem data cleaning provides the concrete procedures to resolve them.
- **Remove outliers:** because outlier removal is a key step within the data cleaning process. Extreme or erroneous values are identified and either corrected or excluded to prevent them from distorting learned patterns.
- **Document the dataset** because every cleaning decision should be documented alongside the dataset. Without this record, the cleaned dataset cannot be correctly interpreted, reproduced, or audited by others.

References: [[Hal19](#), [Til21](#), [Hal20](#), [Gup20](#), [ZMZ23](#), [OBI+22](#), [AG24](#)]

DO – Data governance

Data governance in ML(AI)-fueled systems addresses the need of structured control over data collection, storage, usage, coordination, and management.

Context: Machine learning and MLOps environments where data is produced, shared, and consumed across multiple teams and system components. In such settings, the absence of clear organizational control over data handling and operational processes can lead to inconsistencies and unreliable system behavior.

What: Define and enforce organizational practices that ensure structured handling of data across machine learning systems, including clear responsibilities for data usage, coordination between stakeholders, and controlled management of data-related processes in ML workflows and production systems. Data governance practices include maintaining a data catalog, implementing validation checks (for missing values/duplicates), metadata tagging and ensuring data privacy.

Reason: Data governance is required to reduce operational failures and coordination issues in machine learning systems, where unclear responsibilities and unmanaged data usage can lead to inconsistent outcomes, unreliable system behavior, and difficulties in maintaining ML systems in practice.

Consequences: Applying data governance improves coordination across teams and reduces operational risks in machine learning systems by introducing structured control over data usage and responsibilities. However, it introduces trade-offs such as increased organizational and operational overhead, the need for continuous coordination between stakeholders, and reduced flexibility due to stricter control over data usage. In contrast, insufficient governance increases the likelihood of inconsistent system behavior, misalignment between teams, and operational difficulties in maintaining ML systems.

References: [Zdr24, Cra24]

[Zdr24, Cra24]

BE AWARE – Data inconsistencies

Conflicting or mismatched information within a dataset, often caused by joining disparate sources or errors in the ETL process.

aka: Data Discrepancies; Conflicting Data

Context: Data engineering pipelines and datasets that aggregate information from multiple upstream databases, third-party APIs, or legacy systems where naming conventions and measurement units are not standardized.

Of what: The following types of data inconsistencies should be monitored:

- **Syntax inconsistency** – the same numeric value is written using different syntactic conventions across records, making parsing and comparison ambiguous.
- **Date/Time Format Inconsistency** – dates and timestamps follow different formats across records, making temporal ordering and arithmetic unreliable without prior normalization.
- **Inconsistent representation** – the same real-world entity is expressed using different encoding across different rows, preventing correct grouping or joining (e.g., "USA", "U.S.A.", and "United States" referring to the same country).
- **Missing value inconsistency** – the method used to represent absent or unknown data is not standardized across the dataset, causing missing data to go undetected by generic checks and leading to incorrect imputation or filtering.
- **Transposition inconsistency** – row and column labels are accidentally swapped or mixed in parts of the dataset, causing values to be assigned to the wrong attribute entirely.
- **Special character inconsistency** – the same string value is encoded using different character representations or encodings, causing string matching and deduplication to fail silently.
- **Spacing inconsistency** – a value appears in multiple forms due to variation in the number or type of whitespace characters, so what should be treated as the same value is interpreted as distinct.

- **Separating inconsistency** – values that should logically be stored in separate fields are sometimes combined into a single field and sometimes correctly split, breaking relational operations and feature extraction.
- **Precision inconsistency** – the same numeric attribute is stored with varying numbers of decimal places across records, introducing artificial differences that can distort distance-based or statistical analyses.
- **Value length inconsistency** – fields that encode fixed-length identifiers or codes appear with varying lengths across records due to missing leading zeros or truncation, making lookups and joins fail.
- **Abbreviation inconsistency** – the same concept is abbreviated differently across records, fragmenting what should be a single category.
- **Unit inconsistency** – the same attribute is recorded using different units of measurement across rows or sources.
- **Casing inconsistency** – the same string value appears with different letter casing across records, causing case-sensitive operations such as grouping, filtering, or joining to produce incorrect results.

Reason: Inconsistencies act as “noise” that confuses the model’s learning process. If one system records temperature in Celsius and another in Fahrenheit without conversion, the model learns incorrect relationships. This is a primary cause of model instability and poor generalization, as the system cannot establish a “Ground Truth.”

Mitigation: Recommended mitigation strategies include:

- Define and enforce a **canonical schema and data dictionary**¹ before data collection begins, specifying allowed units, formats, and encoding conventions.
- Implement routines to systematically surface suspicious patterns (e.g., mixed units, ambiguous encodings) during data preparation.
- Maintain **data provenance records** so that the origin of each field is traceable, facilitating the identification of inconsistencies introduced at specific collection stages.

¹A formal specification of each attribute, including its type, format, allowed values, unit of measurement, and semantic meaning.

Exemplifications:

- A defect prediction dataset in which some projects label bugs as **"bug"**, others as **"defect"**, and others as **1**—a classifier trained on this data will learn from noise rather than signal (representation inconsistency).
- A dataset of pull request records where **closed_at** timestamps use ISO 8601 in some rows and Unix epoch in others, causing date arithmetic to produce nonsensical durations (date/time format inconsistency).
- A merged dataset of issue trackers where the **priority** field uses a 1–5 numeric scale in one source and a **low/medium/high** string scale in another, leading to incomparable priority distributions unless reconciled (inconsistent representation / unit inconsistency).
- A people dataset where the **name** field contains **"John Smith"**, **" John Smith"** (leading space), and **"john smith"**—three records that refer to the same individual but are treated as distinct entities (spacing and casing inconsistency).
- A dataset where missing salary values are represented as **NULL** in some rows, **-1** in others, and an empty string in yet others—a naïve imputation step will fail to identify all missing cases (missing value inconsistency).

Related resources:

- **Data cleaning:** because data cleaning provides the concrete procedures that resolve the inconsistencies described here.
- **Ensure data quality:** because conflicting or mismatched information within a dataset is a primary data quality issue that has to be detected and resolved.

References: [BLR⁺24, FFR22, RRDNP24]

CONSIDER – Incremental data processing

Processes data in small, continuous increments rather than reprocessing the entire dataset, enabling efficient handling of large-scale or streaming data.

Context: Data engineering and ML pipelines dealing with large, continuously growing, or streaming datasets where full reprocessing is costly or impractical, especially in real-time or near real-time systems.

What: Process only newly arrived or updated data in small batches or streams, updating outputs, features, or models incrementally instead of recomputing results from the entire dataset.

Reason: It significantly reduces computational cost and latency, enables real-time or near real-time insights, and scales efficiently with growing data volumes compared to full batch reprocessing.

Consequences: Applying incremental processing improves scalability, reduces processing time and resource consumption, and supports timely updates; however, it introduces additional system complexity (state management, consistency handling), requires careful handling of data dependencies, and may increase the risk of error propagation if updates are not properly managed.

Exemplifications: In a real-time recommendation system, instead of retraining the model on the entire dataset daily, new user interactions are processed incrementally to update user profiles and recommendations continuously as new data arrives.

References: [\[Bar23\]](#)

BE AWARE – Data miscoding

Data miscoding occurs when some information is recorded using the wrong type of representation or encoding scheme.

Context: Data preprocessing and data management stages in machine learning and data-driven systems, where data is collected, transformed, and prepared from heterogeneous sources. In such contexts, inconsistent or incorrect data representations frequently arise due to lack of standardization or weak validation practices.

Of what: Be aware of incorrect or inconsistent data type and encoding representations in datasets, where the same information may be stored using different or inappropriate formats. This includes cases such as:

- Integer stored as floating-point
- Floating-point stored as string
- Integer stored as string
- Time stored as string
- Date stored as string
- Date stored as date/time format
- Date/time stored as string
- Inconsistent categorical encodings for the same concept (e.g., multiple coding schemes for the same labels such as A/B/C vs. A/D/N)

Reason: Data miscoding is a data quality issue frequently observed in datasets and data-driven systems due to heterogeneous data sources, weak schema enforcement, or inconsistent preprocessing practices. It can introduce hidden inconsistencies and increase the risk of incorrect interpretation and processing errors in downstream machine learning tasks.

Mitigation: Enforcing strict schema definitions during data ingestion and preprocessing, standardizing data types and encoding schemes, and validating consistency across datasets. Data preprocessing checklists and data quality issue detection practices can be used to systematically identify and correct mismatched or inconsistent data representations before model training.

Exemplifications: For example, a dataset may store numerical values such as age or salary as strings in some records and as numeric types in others, preventing correct aggregation or statistical computation. Similarly, categorical labels may be encoded inconsistently across sources (e.g., different coding schemes for the same class labels), leading to incorrect grouping or model training errors.

Related resources:

- **Data cleaning:** because values recorded with the wrong type or encoding scheme have to be detected and converted to their correct representation during data cleaning.
- **Ensure data quality:** because miscoded values introduce hidden inconsistencies that reduce data quality and can cause incorrect computation or training errors.

References: [[BVG21](#), [Til21](#), [BLR⁺24](#), [HS25](#), [FFR22](#), [SCvD22](#), [RRDNP24](#)]

[[BVG21](#), [Til21](#), [BLR⁺24](#), [HS25](#), [FFR22](#), [SCvD22](#), [RRDNP24](#)]

SOLVE – Data pipeline

A series of automated data processing steps where the output of one element is the input of the next, ensuring data flows reliably from source to destination.

aka: ETL Pipeline; Automated Data Workflow; Data Ingestion Pipeline

Context: Software and ML systems requiring continuous or batch processing of raw data into a format suitable for model training, evaluation, or production serving.

Problem: Manual data processing is error-prone, non-reproducible, and fails to scale. Without a pipeline, data transformations often become “black boxes” where intermediate states are lost, making it difficult to debug, audit, or recreate a specific dataset version.

Solution: Decompose data processing into discrete, modular stages (e.g., ingestion, cleaning, normalization). Connect these stages via a directed acyclic graph (DAG) managed by an orchestrator to ensure systematic execution, error handling, and repeatable results.

Consequences: Implementing an automated pipeline ensures that the data processing lifecycle is modular and repeatable, which significantly improves the reliability of the ML model’s input and enables clear auditing of data lineage.

However, this architecture introduces additional infrastructure that must be managed and monitored, which can lead to increased operational overhead and potential latency in the availability of fresh data.

Exemplifications: An Apache Airflow DAG that triggers a daily extraction of logs from an S3 bucket, applies normalization via a Spark job, and loads the final feature set into a Feature Store for model training.

References: [[HGE25](#)]

DO – Data profiling

Data profiling is the systematic analysis of datasets to understand their structure, content, distribution, and quality characteristics before their use in ML(AI)-fueled systems.

Context: Data profiling is applied in data preparation stages of ML(AI)-fueled systems where data is collected from heterogeneous sources and must be assessed before being used in training, validation, or deployment pipelines. It is relevant in contexts where data quality directly impacts system-level quality attributes such as reliability and Maintainability.

What: Perform structured analysis of datasets to identify statistical properties, schema characteristics, data types, value distributions, missing values, and potential anomalies. This includes examining column-level and dataset-level properties to assess whether the data is suitable for downstream machine learning tasks.

Reason: Data profiling is necessary because machine learning systems are highly dependent on the quality and structure of input data. Without prior understanding of dataset characteristics, hidden quality issues may propagate into model training and system behavior, negatively affecting reliability, correctness, and other system quality attributes.

Consequences: Applying data profiling improves data understanding, supports early detection of quality issues, and reduces risks of incorrect assumptions during preprocessing and model development. It contributes to improved system quality attributes such as reliability and Maintainability.

However, it introduces additional upfront effort and may increase preprocessing time in data-intensive workflows.

Exemplifications: For example, profiling a dataset may reveal unexpected missing values in critical features, inconsistent data types within a column, or highly skewed distributions. Identifying these issues before model training allows corrective preprocessing steps such as imputation, normalization, or schema correction.

References: [\[ISW25\]](#)

CONSIDER – Data retention

Data retention is a critical aspect of data lifecycle management that ensures organizations keep the necessary information for appropriate periods, comply with legal requirements, and optimize storage resources.

Context: Data management in machine learning, and data-driven systems, where decisions must be made about how long data is stored, maintained, and made accessible throughout its lifecycle. This includes considerations of data retention, archival, and deletion across all stages of data handling, particularly in contexts requiring regulatory compliance, auditability, reproducibility, and ethical data governance.

What: Establish and enforce explicit data retention policies that define what data should be stored, for how long, in what form, and for what purpose. These policies apply at the organizational level across the entire data lifecycle in machine learning systems, including artifacts such as raw data, processed datasets, training splits, model artifacts, and inference logs.

Reason: Data retention enables reproducibility, supports monitoring and drift detection with historical data, ensures regulatory compliance, and facilitates debugging and auditing. It also helps control storage costs; without it, systems may either lose critical information or face unnecessary storage and compliance risks.

Consequences: A data retention policy improves reproducibility and traceability by ensuring past decisions and outputs can be reconstructed and audited. It strengthens regulatory compliance by keeping required records accessible throughout their mandated window, and enhances privacy protection by enforcing systematic deletion of personal data once it is no longer needed. Purging obsolete data also reduces storage costs over time. On the other hand, enforcing retention rules introduces operational complexity, requiring clear governance, automated expiration mechanisms, and consistent application across data stores. The benefits are contingent on correct implementation, as policy failures can lead to legal liability or privacy breaches.

Exemplifications: Data retention under the Health Insurance Portability and Accountability Act [U.S03] mandates that healthcare organizations maintain protected health information for at least six years. The retention period starts from the creation date or when the data was last used.

Related resources:

- Ensure data quality: because retaining data beyond its valid period, or discarding it prematurely, affects the quality and the relevance of the dataset.

References: [[ICIA24](#), [Sah24](#)]

BE AWARE – Data unit measure unknown

Data unit measure unknown occurs when the unit of measurement for a numerical attribute is missing, implicit, or inconsistently documented, making the data difficult to interpret correctly.

Of what: Be aware of numerical attributes in datasets where the unit of measurement is not explicitly specified, inconsistently documented, or implicitly assumed. This includes cases where values represent quantities such as length, weight, or time without clear indication of their unit.

Reason: Missing or unclear unit information leads to ambiguity in data interpretation, as the same numeric value may represent different real-world magnitudes depending on the assumed unit. This increases the risk of incorrect assumptions during analysis and can introduce errors in downstream processing of the data.

Mitigation: Explicitly documenting units for all quantitative attributes, ensuring consistent unit usage across the dataset, and validating unit completeness during data collection and preprocessing. Standardization of units should be enforced as part of data quality checks.

Exemplifications: For example, a dataset may contain a feature such as **temperature** with values like 25, without specifying whether the unit is Celsius or Fahrenheit. Without explicit unit information, the same value may be interpreted differently, leading to incorrect analysis results.

References: [[SCvD22](#)]

DO – Data usage consent

Obtain informed consent when collecting, buying or using personal data. Erase or provide ways to erase data and its traces if consent is revoked. Put in action a mechanism through which an individual can request their personal information be removed.

Context: Machine learning and data-driven systems that process personal or sensitive data, where ethical and responsible data handling practices are required throughout the data lifecycle, including collection, storage, and usage phases.

What: Ensure that personal data is collected and used only after obtaining informed consent from individuals. Implement mechanisms that allow users to revoke consent and request the deletion of their personal data, including derived or stored representations where applicable. This includes providing accessible processes for data removal requests.

Reason: Data usage consent is necessary to ensure ethical handling of personal data, protect individual privacy rights, and comply with responsible data practices. Without explicit consent and deletion mechanisms, individuals may lose control over their personal information, leading to ethical, governance and legal regulation issues in data-driven systems.

Consequences: Applying data usage consent practices improves transparency, user trust, and ethical compliance in machine learning systems. It ensures that individuals retain control over their personal data and reduces risks related to misuse or unauthorized retention. However, it introduces operational overhead due to the need for consent management systems and data deletion mechanisms across distributed storage and processing components.

References: [[Sah24](#), [RIT23](#), [XGLR21](#)]

[[Sah24](#), [RIT23](#), [XGLR21](#), [XGLR21](#)]

Obtain informed consent when collecting, buying or using personal data. Erase or provide ways to erase data and its traces if consent is revoked. Put in action a mechanism through which an individual can request their personal information be removed.

DO – Data versioning

The practice of tracking and managing changes to datasets over time, ensuring that every version of a model can be linked to the exact state of the data used to train it.

Context: Machine learning systems where data is non-static and undergoes frequent updates, re-labeling, or cleaning. It is particularly critical for systems requiring reproducibility and for teams collaborating on shared data assets.

What: Implement a system that creates immutable snapshots or unique identifiers (hashes) for every version of a dataset. This involves using specialized tools—such as DVC [DVC], LakeFS [Lak], or Git LFS [Git] to store metadata about data changes in version control alongside the code. This ensures that any specific version of a model is explicitly linked to a specific, immutable version of the data.

Reason: ML models are the product of both code and data. While code versioning is standard, failing to version data creates a “reproducibility crisis” where a model’s performance cannot be replicated because the original training data has been overwritten or lost. Without a history of data changes, it is impossible to perform “time-travel” debugging or to safely roll back to a previous system state if a new data ingestion introduces errors.

Consequences: Data versioning significantly enhances the reproducibility and auditability of the ML lifecycle, allowing researchers to experiment with confidence and meet regulatory requirements for data lineage. While it introduces additional complexity in the storage layer and requires disciplined team workflows, it prevents the loss of historical knowledge and simplifies the process of identifying when and why a model’s performance changed.

Exemplifications: A medical imaging team uses the tool DVC to version their 500GB dataset. When a new batch of labeled X-rays is added, they commit a new metadata file to Git. This allows any researcher to run a single command to “checkout” the exact dataset version used for a specific clinical trial result from six months ago.

References: [Gat22, McA20, VK24, ISW25]

DO – Document the dataset

The systematic recording of a dataset’s origins, composition, collection process, processing steps, and intended use to ensure long-term usability and ethical transparency.

Context: Any project where datasets are collected, curated, or shared across teams. This is essential for preventing the misuse of data and for identifying potential biases before they are learned by a model.

What: Create a standardized “Datasheet” or “Data Card” for every dataset used, collected, or shared within the organization. This document should answer specific questions about why the dataset was created, what it contains, how it was pre-processed, and any legal or ethical constraints on its use. This documentation should be stored in a central registry and linked directly to the data versioning system.

Concretely, the documentation should cover dataset provenance, collection methodology, description of instances and features, applied preprocessing and cleaning steps, known limitations, biases, and missing data and legal, privacy, and ethical constraints on use, such as compliance with privacy regulations. Established frameworks such as *Datasheets*[\[GMV⁺21\]](#) for Datasets and *Data Cards*[\[PZK22\]](#) provide concrete templates for structuring this information. This documentation should be stored in a central registry and linked directly to the data versioning system.

Reason: Datasets are often treated as static artifacts, but without documentation, their limitations and biases are invisible. Future users may not know if a dataset contains sensitive information, how it was sampled, or if it has been “cleaned” in a way that removes important signals. Proper documentation acts as a safety manual, preventing engineers from making false assumptions about the data’s quality or representativeness.

Consequences: Comprehensive dataset documentation fosters trust and accountability, making it easier to audit models for fairness and compliance. It streamlines the onboarding of new team members and reduces the risk of technical debt caused by “tribal knowledge.” Although it requires a significant initial time investment from data engineers and domain experts, it serves as a critical defense against the propagation of hidden biases.

Exemplifications: A financial services company adopts the “Datasheets for Datasets” framework. For their credit scoring dataset, the datasheet explicitly notes that the data was

collected between 2018 and 2022 and that it intentionally excludes certain demographic features to comply with anti-discrimination laws.

Related resources:

- **Data cleaning:** because every cleaning decision should be documented alongside the dataset; without that record the cleaned dataset cannot be correctly interpreted, reproduced, or audited.

References: [[SHHJ22](#), [Via22](#), [WF24](#)]

BE AWARE – Dummy value

It refers to the presence of substitute values in a dataset to represent a condition in which the actual data is unknown, inapplicable, or an error occurred during capture or transformation. It signals a potential data quality issue, often arising from poor data management practices or as a workaround for system constraints.

Of what: Be aware of placeholder or substitute values used to represent missing, unknown, or invalid data, such as default codes that do not reflect real observations but are embedded in the dataset as valid values.

Reason: Dummy values introduce ambiguity in data interpretation, as they may be indistinguishable from valid data without proper documentation. This can lead to incorrect assumptions during analysis and negatively affects data quality and reliability.

Mitigation: Clearly distinguishing missing or invalid data from valid values by using standardized representations for missing data, documenting their meaning, and applying data validation and cleaning steps to identify and handle such placeholders appropriately.

Exemplifications: For example, a dataset may use the value -1 or 999 to represent missing age information.

Related resources:

- **Data cleaning:** because dummy values masquerade as valid data while representing missing or unknown information, and data cleaning must identify and fix all such sentinel values.
- **Ensure data quality:** because dummy values silently degrade data quality by introducing observations that do not represent actual information.

References: [[RRDNP24](#), [FFR22](#), [SCvD22](#), [HS25](#)]

DO – Ensure data quality

Ensure that data used in machine learning systems is accurate, consistent, and suitable for its intended purpose in order to support reliable model development and operation.

aka: Garbage in, Garbage out

Context: Machine learning and data-driven systems, where data is collected, processed, and used for training and deployment. Data quality is critical in both development and production environments, particularly when data originates from multiple sources or evolves over time.

What: Establish and apply practices to assess and improve the quality of data used in machine learning systems. This includes identifying and addressing issues such as missing values, inconsistencies, noise, and incorrect data, and ensuring that the data is suitable for model training and evaluation.

Reason: Ensuring data quality is necessary because poor-quality data can lead to incorrect model behavior, reduced performance, and unreliable outcomes. Data issues are frequently identified as a major cause of failures in machine learning projects.

Consequences: Applying data quality practices improves model performance, reliability, and robustness, and reduces risks associated with incorrect or misleading data. It supports more stable and maintainable machine learning systems. However, ensuring high data quality may require additional effort in data preparation and ongoing monitoring in production environments.

Exemplifications: For example, a machine learning model trained on data containing missing values, inconsistent formats, or incorrect labels may produce inaccurate predictions. By identifying and correcting these issues during preprocessing, the resulting model becomes more reliable and performs better in practice.

Related resources:

- **Ambiguous date/time format** because inconsistent or ambiguous date and time formats undermine data quality by making temporal ordering and arithmetic unreliable across records.

- **Ambiguous value** because ambiguous values directly threaten data quality by introducing records that cannot be interpreted unambiguously, leading to incorrect analysis or model training if not resolved.
- **Data cleaning** Data cleaning is a core activity that operationalizes data quality assurance. It consists of preprocessing techniques applied to detect and correct issues such as missing values, inconsistencies, noise, and incorrect labels. While data quality defines the desired properties of data, data cleaning provides the concrete methods to achieve those properties during the machine learning lifecycle.
- **Data inconsistencies** because conflicting or mismatched information within a dataset is a primary data quality issue that must be detected and resolved to ensure that the data accurately and consistently represents the underlying phenomenon.
- **Data miscoding** because values recorded using the wrong type or encoding scheme introduce hidden inconsistencies that reduce data quality and can cause incorrect computation or model training errors if not corrected.
- **Data retention** because retaining data beyond its valid period or discarding data prematurely affects the quality and relevance of the dataset, as outdated or incomplete historical records can mislead model training and evaluation.
- **Missing values (and metadata)** because missing values are one of the most direct threats to data quality, as incomplete records can bias model training and produce unreliable predictions if not handled appropriately.
- **Dummy value** because dummy values that masquerade as real data silently degrade data quality by introducing observations that do not represent actual information.
- **Meaningless value** because values do have meaning reduce data quality by introducing noise that can mislead analysis and model training.
- **Extraneous value** because fields containing more information than they should violate the expected structure of the data, reducing its reliability and interpretability.
- **No duplicate data** because duplicate records inflate dataset size and distort statistical summaries, directly violating the accuracy and consistency dimensions of data quality.
- **No data duplication** because redundant data across sources introduces inconsistency and increases the risk of conflicting information entering the dataset.
- **Incorrect column header** because mislabeled columns cause values to be misinterpreted throughout the entire dataset, representing a structural threat to data quality.

- Column header containing value because headers that accidentally contain data corrupt the structural integrity of the dataset, making it unreliable for automated processing.

[Mol23, Dez18, Til21, Kao21, Rom24, Zdr24, She25, Wij24b, Sra24, Wij24a, WF24, Lon24, SKvdBHV20, SvdBV23]

DON'T – ETL monolith

When a single, tightly coupled, centralized process handles all Extract, Transform, and Load (ETL) operations for the entire organization or a vast majority of its data. It is characterized by a single, massive codebase or platform responsible for connecting to numerous disparate source systems, applying complex business logic and transformations, and loading everything into a single, massive data warehouse or data lake. This approach, while simple at a small scale, becomes a major bottleneck and source of technical debt as the organization and its data volumes grow.

Context: Data engineering and data processing environments where ETL pipelines are used to integrate, transform, and load data from multiple heterogeneous sources into centralized storage systems. This is especially relevant in growing systems where data volume, variety, and processing needs increase over time.

What: Avoid designing ETL processes as a single monolithic system that combines all data ingestion, transformation, and loading logic into one centralized and tightly coupled pipeline. Such systems typically aggregate multiple responsibilities and dependencies into a single implementation.

Reason: Monolithic ETL pipelines reduce modularity and flexibility, making the system difficult to maintain, extend, and scale. Changes in one part of the pipeline may impact unrelated components, increasing complexity and the risk of failures.

How to avoid: Adopt modular and decoupled data processing approaches by separating ETL components into smaller, independent pipelines or stages. Encourage clear separation of concerns, incremental processing, and independent evolution of data workflows to improve Maintainability and scalability.

Exemplifications: For example, a single ETL pipeline that processes data from multiple sources and applies all transformations in one centralized workflow may become difficult to modify when new data sources or transformations are introduced, leading to delays and increased risk of errors.

References: [\[STR24\]](#)

BE AWARE – Extraneous value

Extraneous value refers to a situation where a single attribute contains more information than intended by its semantic role, meaning the value encodes multiple pieces of information or exceeds the column’s defined purpose.

Of what: Be aware of columns where a single cell contains multiple distinct values instead of a single atomic value per field

Reason: Extraneous values violate the principle of atomic and well-defined attributes, leading to ambiguity in interpretation and difficulties in data processing. This can reduce data quality by complicating parsing, feature extraction, and consistent understanding of attribute semantics.

Mitigation: Ensure that each attribute represents a single, well-defined concept by normalizing data structures, splitting overloaded fields into separate attributes, and enforcing clear schema definitions during data collection and preprocessing.

Exemplifications: For example, a single field such as "UserInfo" may contain both a user ID and a timestamp, which should instead be stored as separate attributes. If not separated, downstream processing may misinterpret or incorrectly extract features from the data.

Related resources:

- **Ensure data quality:** because fields containing more information than they should violate the expected structure of the data, reducing its reliability and interpretability.

References: [[RRDNP24](#), [FFR22](#)]

BE AWARE – Extreme data value

It refers to data points that are unusually high or low compared to the rest of the dataset and may indicate anomalies or data quality issues.

Of what: Be aware of values in datasets that significantly deviate from the typical range of observed data. These extreme values may represent rare events, but they can also indicate data errors, inconsistencies, or measurement issues that affect data quality.

Reason: Extreme values can distort statistical analysis and negatively affect data quality by skewing distributions and influencing downstream machine learning models. literature identifies such anomalies as indicators of potential data quality problems that require further investigation.

Mitigation: Detecting extreme values during data preprocessing and assessing whether they represent valid rare events or erroneous data. Depending on the context, they can be corrected, transformed, or removed to reduce their impact on analysis and modeling.

Exemplifications: For example, in a dataset containing human age, values such as 120 may appear due to data entry errors. Such extreme values can significantly distort statistical summaries and negatively impact machine learning model training if not handled properly.

References: [[RRDNP24](#), [HS25](#)]

DO – Make data sets available on shared infrastructure (private or public)

Make datasets accessible through shared infrastructure to enable reuse, collaboration, and consistent usage across machine learning development processes.

Context: Machine learning development environments where multiple practitioners or teams need access to datasets for training, evaluation, and experimentation. This is particularly relevant in collaborative settings and organizations where data is reused across projects.

Provide datasets through shared storage systems, either private or public, to ensure that data is accessible to relevant stakeholders. This includes organizing datasets in a way that supports consistent access and reuse across different stages of machine learning workflows.

What: Provide datasets through shared storage systems, either private or public, to ensure that data is accessible to relevant stakeholders. This includes organizing datasets in a way that supports consistent access and reuse across different stages of machine learning workflows.

Reason: Making datasets available on shared infrastructure supports collaboration and reuse, which are identified as important engineering practices in machine learning. Limited access to data can hinder experimentation, reduce reproducibility, and create inefficiencies in development processes.

Consequences: Applying this practice improves collaboration, enables dataset reuse, and supports more consistent experimentation and development in machine learning systems. It contributes to better coordination across teams and more efficient workflows. However, it may require additional effort to manage access and maintain the shared infrastructure.

Exemplifications: For example, storing datasets in a shared repository allows multiple teams to access the same data for training and evaluation, reducing duplication and ensuring that experiments are conducted on consistent data.

References: [[SvdBV23](#), [SKvdBHV20](#)]

BE AWARE – Incorrect column header

Incorrect column header refers to misleading, ambiguous, or inaccurate naming of dataset attributes, which can lead to misinterpretation of the data.

Of what: Be aware of column names that do not correctly represent the underlying data, including ambiguous labels, misleading names, inconsistent naming conventions, or headers that encode incorrect or incomplete information about the attribute.

Reason: Incorrect column headers reduce data understandability and can lead to incorrect assumptions during analysis and model development. Unclear or misleading metadata undermines data quality and increases the likelihood of errors in data interpretation and use.

Mitigation: Ensuring that column headers are clear, consistent, and accurately describe the data they represent. This includes applying standardized naming conventions, reviewing schema definitions, and validating that attribute names align with their actual content.

Exemplifications: For example, a column labeled **age** may actually contain age groups (e.g., “20-30”, “30-40”) rather than exact numeric values. Such a mismatch between the header and the data can lead to incorrect assumptions and flawed analysis.

Related resources:

- Ensure data quality: because mislabeled columns cause values to be misinterpreted throughout the entire dataset, which is a structural threat to data quality.

References: [[RRDNP24](#)]

BE AWARE – Intermingled data type

When a single column or field contains values of different data types or semantically different kinds of information, violating the fundamental principle of an “atomic” and consistent column. This forces anyone using the data to perform complex parsing and type-checking for nearly every operation, making the data error-prone and difficult to use.

Of what: Be aware of dataset attributes in which different data types are mixed within the same column, such as numeric and textual values, or structured and unstructured representations coexisting in a single field.

Reason: Intermingled data types reduce data consistency and complicate data processing, as operations and analyses typically assume a uniform data type per attribute. Data smell literature identifies such inconsistencies as a source of errors and reduced data quality, increasing the likelihood of incorrect interpretation and processing failures.

Mitigation: Enforcing consistent data types at the attribute level, separating heterogeneous values into appropriate fields, and applying data cleaning and validation steps to ensure uniform representation across the dataset.

Exemplifications: For example, a column representing age may contain numeric values (e.g., 25, 30) along with textual entries such as “unknown” or “N/A”. This mixture of data types can lead to errors in statistical analysis or machine learning preprocessing if not properly handled.

References: [[FFR22](#), [RRDNP24](#)]

BE AWARE – Long data value

When a data value in a column is excessively long compared to the other values in the same column.

Of what: Be aware of attribute values that are excessively long compared to the expected representation, such as overly detailed text, concatenated information, or irregularly extended strings within a single field.

Reason: Long data values may indicate inconsistent data representation or improper structuring of information, where multiple pieces of data are combined into a single field. As identified in the literature, such irregularities reduce data quality, complicate processing, and increase the likelihood of misinterpretation.

Mitigation: Identifying excessively long values during data inspection and restructuring the data by splitting combined information into appropriate attributes, as well as enforcing consistent value formats and length constraints where applicable.

Exemplifications: For example, a column intended to store a product name may contain long concatenated strings that include descriptions, categories, and additional metadata in a single field. This makes it difficult to process or analyze the data correctly.

References: [[RRDNP24](#), [FFR22](#)]

BE AWARE – Meaningless value

When a value is present in a dataset that carries no intrinsic meaning, provides no useful information for analysis or modelling, or violates a known context.

Of what: Be aware of values that do not convey meaningful or interpretable information within the dataset context. This includes:

- Placeholder values or simple default strings that do not represent actual data.
- Contextually invalid values, i.e., syntactically correct entries that violate domain constraints, such as negative ages or impossible quantities.
- Meaningless fallback or default values used in place of missing data, where the assigned value does not accurately represent the absence of information.

Reason: Meaningless values reduce data quality by introducing noise and ambiguity, making it difficult to interpret data correctly. As identified in literature, such values can mislead analysis and negatively impact machine learning models by introducing patterns that have no real-world significance.

Mitigation: Identifying and removing or replacing meaningless values during data cleaning. This includes validating values against expected domains, applying consistency checks, and ensuring that all data entries correspond to meaningful and interpretable information.

Exemplifications: For example, a dataset may contain entries such as "xxx", "—", or random character strings in a field expected to store valid categorical values. These entries do not represent any meaningful category and can lead to incorrect analysis or model behavior if not handled properly.

Related resources:

- **Data cleaning:** because values that do not have meaning must be identified and either removed or replaced during data cleaning.
- **Ensure data quality:** because meaningless values reduce data quality by introducing noise that can mislead analysis and model training.

References: [RRDNP24, FFR22]

BE AWARE – Missing values (and metadata)

Missing values refer to the absence of data for certain attributes in a dataset, which can negatively affect data quality and downstream analysis.

Of what: Be aware of attributes where data entries are absent or not recorded, including explicit missing markers (e.g., **NaN**, **NULL**) as well as implicitly missing values represented by empty fields or inconsistent placeholders.

Reason: Missing values reduce data completeness and can introduce bias or distort analysis results. As highlighted in machine learning practice literature, unhandled missing data can lead to incorrect model behavior, reduced performance, and unreliable outcomes.

Mitigation: Identifying missing values during data preprocessing and applying appropriate handling strategies, such as imputation, removal of incomplete records, or use of models robust to missing data. Consistent representation and documentation of missing values are also required.

Exemplifications: For example, a dataset used for predicting customer behavior may contain missing values in key features such as income or age. If these missing values are not properly handled, the resulting model may produce biased or inaccurate predictions.

Related resources:

- **Data cleaning:** because handling missing values is one of the core tasks of data cleaning, and the choice between imputing and dropping them affects the completeness of the dataset.
- **Ensure data quality:** because incomplete records can bias model training and produce unreliable predictions if they are not handled appropriately.

References: [HS25, Til21, RRDNP24, SCvD22, Hal19, Mak18, CKB+23, Cen24, BVG21, ZCvD22, Sha25, BLR+24, Kao21]

BE AWARE – Multi value data

When multiple, logically distinct pieces of information are improperly stored together as a single value in a single cell or field of a dataset.

Of what: Be aware of dataset fields where a single attribute contains multiple distinct values that should be represented separately, such as concatenated identifiers, combined categories, or multiple records encoded within one cell.

Reason: Multi-value storage violates the assumption of atomicity in data representation, leading to difficulties in parsing, querying, and analyzing data. As identified in literature, this issue reduces data quality and increases the likelihood of incorrect aggregation, feature extraction, and downstream processing errors in machine learning systems.

Mitigation: Restructuring the dataset to ensure atomic representation of attributes, such as splitting multi-value fields into separate records or columns, and enforcing schema constraints during data collection and preprocessing.

Exemplifications: For example, a single field may contain multiple product categories such as "electronics, home-appliances", or multiple identifiers like "ID123; ID456", which should instead be stored as separate entries or in a properly structured relational format.

Related resources:

- **Data cleaning:** because fields that pack several values into a single cell violate atomicity and have to be normalized during data cleaning.

References: [[RRDNP24](#)]

DON'T – No data duplication

When the same or similar data are stored or processed in multiple locations or formats, without a clear and consistent source of truth.

Context: Data management and machine learning pipelines where data is collected from multiple sources, integrated across systems, or repeatedly processed and stored at different stages of the workflow.

What: Do not create or store multiple copies of the same data across datasets or systems through repeated ingestion, improper merging, or lack of coordination between data sources.

Reason: Data duplication introduces redundancy and increases the likelihood of inconsistencies, as multiple copies of the same data may diverge over time. It is identified in data quality literature as a cause of downstream data issues.

How to avoid: Avoid repeated ingestion of the same data, ensure proper data integration practices, and validate data pipelines to prevent unintentional replication of records.

Exemplifications: For example, the same dataset is imported multiple times from different sources without proper checks, resulting in repeated storage of identical records.

Related resources:

- **Ensure data quality:** because redundant data across sources introduces inconsistency and increases the risk of conflicting information entering the dataset.
- **No duplicate data:** because the two resources concern different forms of duplication. No data duplication concerns the same data held in several systems or locations, while No duplicate data concerns identical records inside a single dataset.

References: [[BVG21](#)]

DON'T – No duplicate data

The presence of identical or repeated records within a dataset, indicating a data quality issue.

Context: Datasets used in data analysis and machine learning, especially those constructed from multiple sources or over time.

What: Do not allow identical or semantically equivalent records to appear multiple times within a dataset or across dataset partitions.

Reason: Duplicate data reduces data quality by introducing redundancy and can distort analysis results and machine learning outcomes. It is recognized as a data smell in data quality literature.

How to avoid: Avoid introducing duplicate data by validating data during ingestion, applying consistency checks when merging datasets, and ensuring that dataset splits (e.g., training and test sets) do not contain overlapping or repeated records.

Exemplifications: For example, a customer record appears multiple times in a dataset due to repeated data collection or merging, leading to biased analysis results.

Related resources:

- **Ensure data quality:** because duplicate records inflate the dataset and distort statistical summaries, violating the accuracy and the consistency of the data.
- **No data duplication:** because the two resources concern different forms of duplication. **No duplicate data** concerns identical records inside a single dataset, while **No data duplication** concerns the same data held in several systems or locations.

References: [[HS25](#), [BLR⁺24](#), [SCvD22](#), [KN23](#)]

SOLVE – NoSQL

A typology of database management systems designed to handle unstructured data, and to provide high velocity, and horizontal scalability by diverging from traditional relational structures.

aka: Non-relational database; Schema-less database; NoSQL Storage

Context: Systems that must handle heterogeneous, high-velocity access or large-scale data where the rigid schema and vertical scaling of relational databases become bottlenecks for both storage and real-time access.

Problem: Relational databases cannot efficiently handle the volume, velocity access, and variety of data demanded by modern machine learning workloads. Fixed schemas make it costly and slow to adapt to evolving feature sets, new data types, or changing model requirements, while centralised SQL systems struggle to scale horizontally for distributed AI applications.

Solution: Adopt a NoSQL database as the primary data backend. NoSQL databases provide schema flexibility, horizontal scalability via sharding and replication, and support for polyglot persistence, allowing ML teams to store and retrieve diverse data types with low-latency access patterns suited to training, inference, and real-time feature serving. A unified data platform may further combine operational and analytical NoSQL workloads under one system.

Consequences: Implementing a NoSQL solution provides superior performance for large-scale data ingestion and low-latency access, ensuring that the system can scale easily as data volume grows. It offers developers the flexibility to iterate on data structures without performing expensive schema migrations. However, it requires a shift in how data is queried, as NoSQL lacks complex join operations and “ACID” consistency in some configurations, which can lead to eventual consistency challenges during high-concurrency updates.

Exemplifications: NoSQL databases come in four main types, each suited to different ML system needs: *Document stores* (e.g., MongoDB) store semi-structured data such as user profiles with varying attributes; *Key-value stores* (e.g., Redis) provide low-latency access to simple data such as session states or feature caches; *Column-family stores* (e.g., Apache

Cassandra) handle high-velocity time-series data such as billions of IoT sensor readings per day; and *Graph stores* (e.g., Neo4j) represent and query complex relationships such as those found in knowledge graphs or social networks.

References: [\[SMS⁺22\]](#)

DON'T – Over-aggressive filtering

Over-aggressive filtering refers to the excessive removal of data points during preprocessing, particularly outliers, where valid and informative observations are incorrectly discarded.

Context: Data preprocessing and cleaning stages in machine learning workflows, particularly during outlier detection, noise removal, and data filtering operations.

What: Do not remove large portions of data or eliminate observations solely based on simplistic rules or thresholds without considering their context or validity. This includes blindly discarding outliers or rare cases that may carry important information.

Reason: Over-aggressive filtering can lead to loss of valuable information, reduced data diversity, and biased datasets. According to outlier analysis literature, not all extreme values are erroneous; removing them indiscriminately may distort the underlying data distribution and negatively impact model performance.

How to avoid: Use context-aware filtering strategies, validate whether detected outliers are truly erroneous, apply domain knowledge when defining thresholds, and prefer robust statistical methods that are less sensitive to extreme values instead of removing them blindly.

Exemplifications: For example, removing all high-value transactions in a financial dataset as outliers may eliminate legitimate but rare events, leading to biased models that fail to capture important patterns such as fraud or high-value customer behavior.

References: [[Sam24](#)]

DO – Process raw data only once

Processing raw data only once means designing workflows in which raw data is ingested and transformed a single time into reusable intermediate or final outputs.

Context: Data engineering and machine learning pipelines where raw data is repeatedly accessed, transformed, or recomputed across multiple stages or workflows.

What: Ensure that raw data is processed once and the results are stored and reused, rather than repeatedly reprocessing the same raw inputs in different parts of the system.

Reason: Repeated processing of raw data increases computational cost, introduces potential inconsistencies, and reduces reproducibility. Processing data once and reusing the results improves efficiency, consistency, and reliability of data pipelines.

Consequences: This improves performance and consistency by avoiding redundant computations and ensuring that the same transformations are reused. However, it may require additional storage and careful management of intermediate data artifacts to maintain correctness over time.

Exemplifications: For example, instead of recomputing feature extraction from raw logs for every model training run, the processed features are computed once and stored, allowing consistent reuse across experiments.

References: [[eug23](#)]

DO – Remove outliers

Removing outliers refers to the process of identifying and eliminating data points that significantly deviate from the majority of observations and are considered likely to be erroneous or not representative of the underlying data distribution.

Context: Data preprocessing and feature engineering stages in machine learning pipelines where noisy, corrupted, or anomalous observations may be present in collected datasets.

What: Identify and remove data points that are statistically or contextually extreme and are considered invalid, noisy, or not representative of the target data distribution for the intended analysis or model training.

Reason: Outliers can distort statistical summaries, bias parameter estimation, and negatively affect the performance and stability of machine learning models. As highlighted in data preprocessing and ML best-practice literature, handling outliers appropriately is important to improve data quality and model robustness.

Consequences: Removing outliers can improve model stability, reduce noise, and enhance predictive performance when the outliers are truly erroneous. However, inappropriate removal may lead to loss of meaningful rare cases, reduced data diversity, and biased models that fail to capture important edge behaviors.

Exemplifications: For example, in a sensor-based temperature dataset, values such as -999 or 9999 may appear due to device malfunction or logging errors and should be removed or treated as invalid. However, naturally extreme but valid observations (e.g., unusually high temperatures during a heatwave) should be retained to avoid distorting the model's understanding of real-world variability.

Related resources:

- **Data cleaning:** because the removal of outliers is one of the steps of the data cleaning process, in which extreme or erroneous values are corrected or excluded.

References: [[Til21](#), [Mak18](#), [BLR⁺24](#), [SHHJ22](#), [Pho22](#), [Cen24](#), [Kao21](#), [Zwi22](#), [BLR⁺24](#)]

DO – Responsible data management

Responsible data management refers to the structured and ethical handling of data across its lifecycle, ensuring that data is collected, stored, processed, and used in a controlled, transparent, and accountable manner.

Context: Data-intensive systems and machine learning workflows where data is collected from multiple sources, processed through pipelines, and used for training, evaluation, and deployment of models.

What: Establish and apply practices that ensure data is managed in a structured, transparent, and controlled way across its lifecycle, including clear ownership, access control, documentation, and governed usage of data assets.

Reason: Responsible data management is necessary to ensure data quality, consistency, and trustworthiness in data-driven systems. Without structured management practices, data becomes fragmented, difficult to trace, and prone to misuse or misinterpretation, which negatively impacts analytical and machine learning outcomes.

Consequences: Applying responsible data management improves data reliability, traceability, and accountability, enabling more robust and trustworthy machine learning systems. However, it may introduce additional organizational overhead, require coordination between stakeholders, and increase the need for governance infrastructure and compliance monitoring.

References: [[Lau23](#), [She25](#), [HTK23](#)]

DON'T – Schema on read

When data is loaded into storage in its raw, native format without any upfront validation or transformation, and the structure (schema) is interpreted and applied dynamically at the time the data is read (queried) by the consumer. This contrasts with the traditional Schema-on-Write approach (used by relational databases), where the structure is enforced before the data is written, and any data not conforming to the schema is rejected.

Context: Data lakes, large-scale data storage systems, and machine learning pipelines where heterogeneous data from multiple sources is ingested in raw form and later consumed for analytics, feature engineering, or model training.

What: Do not rely on interpreting or enforcing data structure only at read or query time without prior validation or standardization during data ingestion. Avoid storing unvalidated raw data without defined schema expectations for downstream use.

Reason: Schema-on-read can lead to inconsistent interpretations of the same data, as different consumers may apply different schemas or assumptions at query time. This increases the risk of data quality issues, integration errors, and unreliable machine learning features due to lack of early validation and standardization.

How to avoid: Define and enforce schema constraints during data ingestion, apply validation and standardization before storing data, and maintain consistent schema definitions across data consumers to ensure uniform interpretation of datasets.

Exemplifications: For example, storing raw JSON logs without enforcing field consistency may lead to different applications interpreting missing or renamed fields differently, resulting in inconsistent features used for model training.

References: [\[STR24\]](#)

BE AWARE – Small number

When data values represent numbers in the range (0,1). If several such values are multiplied in subsequent operations, the calculation result can be very small and has many decimal places. As decimal places are often limited manually (e.g., three decimal places) or by the system (i.e., arithmetic underflow), this can lead to incorrect further processing (e.g., $0.02 * 0.02 = 0.0004$).

Context: Numerical data processing and machine learning pipelines where features contain values in the range (0, 1), especially in probabilistic, normalized, or ratio-based representations that undergo repeated arithmetic operations.

Of what: Be aware of numeric attributes containing values in the range (0, 1), especially when such values are used in repeated multiplication or aggregation operations that can produce extremely small results.

Reason: Small values can lead to numerical precision issues in computational systems due to limited floating-point precision or rounding constraints. In iterative or multiplicative computations, this may result in values that underflow or are rounded inconsistently, affecting the correctness of downstream processing.

Mitigation: Apply appropriate numerical scaling, use higher-precision data types when needed, and avoid unnecessary repeated multiplication of small values without normalization or stability checks.

Exemplifications: For example, multiplying several probability-like values such as $0.02 \times 0.02 \times 0.02$ can produce extremely small results (e.g., 0.000008), which may be rounded or truncated by the system, leading to incorrect downstream computations or loss of significance.

References: [RRDNP24, FFR22]

BE AWARE – Special characters in data

When non-alphanumeric characters (like punctuation, symbols, or even control characters) are embedded within data fields in a way that is inconsistent, unintended, or carries implicit meaning. This is not about characters that are part of the legitimate data (such as a hyphen in a phone number or an apostrophe in a name), but about characters used as delimiters, flags, or formatting markers within a single field.

Context: Data ingestion, storage, and preprocessing pipelines where textual or structured data is collected from heterogeneous sources such as user input systems, APIs, logs, or external databases, often leading to unintended inclusion of special characters.

Of what: Be aware of data fields that contain special characters used inconsistently, such as embedded delimiters, hidden formatting symbols, or symbols that unintentionally affect parsing, encoding, or interpretation of values.

Reason: Special characters can introduce ambiguity in data processing by interfering with parsing rules, encoding schemes, or string matching operations. As a result, they may lead to incorrect feature extraction, data corruption during transformation, or inconsistent interpretation across systems.

Mitigation: Standardize input encoding, sanitize data during ingestion, escape or remove unintended special characters, and enforce consistent formatting rules for fields that may contain symbolic or textual noise.

Exemplifications: For example, a field such as “John | Doe” may use the pipe symbol as an unintended delimiter, causing incorrect splitting of a single value into multiple attributes during parsing. Similarly, hidden control characters may disrupt string comparisons or database imports.

References: [[RRDNP24](#), [SCvD22](#), [FFR22](#)]

BE AWARE – Split value data

When a single logical piece of information is unnaturally split across multiple columns in the same table, instead of having one column that holds a complete value, the value is fragmented. This violates the principle that a column should represent a single, atomic attribute of an entity.

Context: Data integration and preprocessing stages in machine learning pipelines where data is collected from multiple sources or legacy systems, often leading to attributes being represented across multiple columns instead of a single atomic field.

Of what: Be aware of cases where a single real-world attribute is distributed across multiple columns, requiring reconstruction or concatenation to obtain the complete value instead of being stored as one atomic field.

Reason: Split values reduce data quality by violating attribute atomicity, increasing complexity in data processing, and making feature extraction and interpretation more error-prone. They can also lead to inconsistent handling of the same concept across different parts of a dataset or pipeline.

Mitigation: Redesign the data schema to ensure each attribute is stored in a single column as a complete atomic value, and merge fragmented fields during preprocessing or data integration stages.

Exemplifications: For example, a full date stored across separate columns such as "Year = 2024", "Month = 01", and "Day = 15" instead of a single standardized date field requires recombination before it can be used for temporal analysis or machine learning models.

References: [[RRDNP24](#)]

BE AWARE – Strings in human-friendly formats

When data that should be stored in a structured, machine-readable format is instead stored as a free-form string intended for easy human reading or input. This often involves using a single string to represent multiple pieces of information with formatting like commas, spaces, or currency symbols.

Context: Data ingestion, integration, and preprocessing pipelines where data is collected from heterogeneous sources such as user input forms, APIs, or legacy systems, often leading to free-form textual representations of structured information instead of properly typed and structured fields.

Of what: Be aware of fields where structured or multi-attribute information is encoded as a single free-form string, often including multiple values combined using delimiters, formatting symbols, or human-readable conventions instead of separate typed attributes.

Reason: Storing structured information as free-form strings reduces data consistency, complicates parsing and feature extraction, and increases the risk of interpretation errors across different systems. It also limits the ability to validate, query, and transform data efficiently in machine learning and analytical workflows.

Mitigation: Enforce structured data schemas at ingestion time, separate distinct attributes into dedicated columns, and apply parsing and validation rules to convert free-form strings into normalized and machine-readable formats during preprocessing.

Exemplifications: For example, storing a value like "12.5 USD, shipped, 2024-01-01" in a single string field instead of separate columns for price, status, and date forces downstream systems to manually parse and interpret the information, increasing the risk of errors and inconsistent feature extraction.

References: [[RRDNP24](#), [SCvD22](#)]

BE AWARE – Suspect data distribution

Suspect data distribution refers to situations where the observed distribution of values in a dataset deviates from expected, domain-consistent, or statistically plausible patterns, potentially indicating data quality issues or collection problems.

Context: Data analysis and machine learning preprocessing stages where datasets are aggregated from multiple sources or collected over time, and statistical properties of features are expected to follow known or stable patterns.

Of what: Be aware of feature distributions that appear inconsistent with domain expectations or historical behavior, such as unexpected skewness, abnormal concentration of values, or distributions that suggest data collection or preprocessing anomalies.

Reason: Unusual or implausible data distributions may indicate underlying data quality problems such as sampling bias, data corruption, measurement errors, or incorrect preprocessing steps. These issues can mislead statistical analysis and degrade machine learning model performance by violating assumptions about the data-generating process.

Mitigation: Validate feature distributions using exploratory data analysis, compare distributions against historical or domain knowledge baselines, and investigate anomalies through data profiling and pipeline auditing before model training.

Exemplifications: For example, a feature representing customer age may show an unrealistic concentration of values around a single age (e.g., all entries being 30), which may indicate data imputation errors or default value filling rather than true observed behavior.

References: [[RRDNP24](#), [FFR22](#), [Til21](#)]

BE AWARE – Suspect data precision

When the numerical precision (the number of decimal places or significant figures) of data values appears inconsistent, excessive, or inappropriately rounded, leading to suspicion about the data’s true accuracy or source.

Context: Numerical datasets in machine learning and data analytics pipelines where values are collected from multiple sources, sensors, or systems with different measurement and storage precision levels.

Of what: Be aware of numerical attributes whose precision is inconsistent across records or unnecessarily high/low relative to the domain, indicating potential measurement, storage, or preprocessing artifacts.

Reason: Inconsistent or suspicious precision may indicate data acquisition errors, incorrect type conversions, or unintended rounding during preprocessing. This can distort statistical analysis and negatively impact model stability and interpretability.

Mitigation: Standardize numerical precision based on domain requirements, validate measurement resolution during data ingestion, and apply consistent rounding or normalization strategies across datasets.

Exemplifications: For example, a dataset may contain values such as 12.3, 12.3000000001, and 12.300, suggesting inconsistent precision caused by floating-point representation or improper data formatting during storage or transformation.

References: [[HS25](#), [FFR22](#), [RRDNP24](#)]

BE AWARE – Suspect date/time interval

When time intervals between consecutive date or timestamp values appear inconsistent, implausible, or violate expected temporal patterns in the data.

Context: Time-series data and event logs in machine learning and data analytics pipelines where temporal ordering and regular sampling are expected or assumed.

Of what: Be aware of irregular or inconsistent time intervals between records that deviate from expected temporal patterns, such as unexpected gaps, duplicates, or unrealistic frequency changes.

Reason: Irregular or suspicious time intervals may indicate missing data, logging errors, clock synchronization issues, or incorrect data aggregation, which can distort temporal analysis and forecasting models.

Mitigation: Validate temporal consistency during preprocessing, enforce chronological ordering, and apply resampling or interpolation techniques when appropriate to ensure consistent time intervals.

Exemplifications: For example, a sensor dataset expected to record measurements every 10 minutes may contain irregular gaps of several hours or duplicate timestamps, indicating data loss or logging failures.

References: [[RRDNP24](#), [FFR22](#), [SCvD22](#)]

BE AWARE – Suspect date value

When individual date values in a dataset appear invalid, inconsistent, or implausible with respect to the expected temporal context or domain constraints.

Context: Datasets containing temporal attributes such as event logs, transactions, or user activity records collected from heterogeneous systems or manual input sources.

Of what: Be aware of date values that fall outside valid ranges, violate logical constraints (e.g., future dates in historical records), or are inconsistent with related temporal attributes.

Reason: Invalid or implausible date values often arise from data entry errors, format misinterpretation, or incorrect system defaults, leading to incorrect temporal analysis and unreliable downstream processing.

Mitigation: Validate date formats and ranges during ingestion, enforce domain-specific temporal constraints, and standardize date parsing across all data sources.

Exemplifications: For example, a transaction dataset may contain a date such as 2099-01-01 for historical records, indicating either a data entry error or incorrect system-generated default values.

References: [[RRDNP24](#), [FFR22](#)]

BE AWARE – Suspect sign

When there is a value with the opposite sign of the majority of the data values.

References: [[HS25](#)]

BE AWARE – Synonyms in data

When the same concept or meaning is represented by multiple, different names, terms, or codes within a dataset or across related datasets (e.g., in a Gender column, you might have values like: M, Male, and Man and in an address dataset, one source uses St. for Street, while another uses Street).

Context: Integrated datasets and data preprocessing pipelines where information is collected from multiple sources with inconsistent naming conventions or coding schemes.

Of what: Be aware of multiple representations of the same semantic concept, such as different labels, abbreviations, or codes used interchangeably for identical values.

Reason: Synonyms in data lead to fragmentation of categorical values, reducing data consistency and causing incorrect aggregation, analysis, or feature encoding in machine learning models.

Mitigation: Standardize categorical values using mapping tables, enforce controlled vocabularies, and harmonize naming conventions across data sources during preprocessing.

References: [[RRDNP24](#), [FFR22](#)]

BE AWARE – Unsound/missing metadata

When metadata describing datasets, features, or data processing steps is incomplete, inconsistent, or unreliable, reducing the interpretability and usability of the data.

Context: Machine learning and data engineering pipelines where datasets are shared, reused, or integrated across teams without sufficient documentation or metadata management practices.

Of what: Be aware of datasets or features that lack proper metadata, contain outdated descriptions, or have inconsistent or missing documentation about structure, meaning, or provenance.

Reason: Missing or unreliable metadata reduces data transparency, makes interpretation difficult, and increases the risk of incorrect assumptions during analysis or model development. This also hinders reproducibility and data governance.

Mitigation: Maintain up-to-date and standardized metadata documentation, enforce metadata requirements during dataset publication, and ensure traceability of data sources and transformations.

Exemplifications: For example, a dataset column labeled "value" without any description may represent different meanings across datasets, making it impossible to correctly interpret or reuse the data without additional context.

References: [[BVG21](#)]

Appendix B

Answers of the Experts

This appendix reports the answers of the four experts who took part in the evaluation described in Chapter 5, exactly as they were returned. The experts are identified as Expert 1 to Expert 4; the order is arbitrary and does not correspond to the order in which they were invited. Table B.1 reports the first part of the questionnaire, which concerns the resources, and Table B.2 the second part, which concerns their complete descriptions. Expert 4 did not answer the second part.

Within each kind the resources are listed in alphabetical order. A dash means that the expert left the question unanswered, in most cases because he preferred to write a comment instead of choosing one of the options. The answers are abbreviated as follows.

Useful, Familiar, Clear **Y** yes, **N** no.

Effect of AI **E** it will be easier to use, **B** it will be as before, **U** it will become useless.

Level of detail **R** right level of detail, **C** too concise, **D** too detailed.

Table B.1: Answers of the experts on the resources (first part of the questionnaire).

Resource	Useful				Familiar				Effect of AI			
	1	2	3	4	1	2	3	4	1	2	3	4
Be aware												
Aggregation mismatch	Y	Y	Y	Y	Y	Y	Y	Y	E	B	B	B
Ambiguous date/time format	Y	Y	Y	Y	Y	Y	Y	Y	E	B	E	B
Ambiguous value	Y	Y	Y	Y	Y	Y	Y	Y	E	B	E	B

Resource	Useful				Familiar				Effect of AI			
	1	2	3	4	1	2	3	4	1	2	3	4
Column header containing value	Y	Y	Y	Y	Y	Y	Y	Y	E	–	E	E
Data inconsistencies	Y	Y	Y	Y	Y	Y	Y	Y	E	B	E	E
Data miscoding	Y	Y	Y	Y	Y	Y	Y	Y	E	–	E	E
Data unit measure unknown	Y	Y	Y	Y	Y	Y	Y	Y	E	E	E	E
Dummy value	Y	Y	Y	Y	Y	Y	Y	Y	E	B	B	E
Extraneous value	Y	Y	Y	Y	Y	Y	Y	Y	E	B	E	E
Extreme data value	Y	Y	Y	Y	Y	Y	Y	Y	E	B	E	E
Incorrect column header	Y	Y	Y	Y	Y	Y	Y	Y	E	E	E	E
Intermingled data type	Y	Y	Y	Y	Y	Y	Y	Y	E	B	B	E
Long data value	Y	N	Y	N	Y	N	Y	Y	E	B	E	U
Meaningless value	Y	Y	Y	N	Y	N	Y	Y	E	U	E	U
Missing values (and meta-data)	Y	Y	Y	Y	Y	Y	Y	Y	E	E	E	E
Multi value data	Y	Y	Y	Y	Y	Y	Y	Y	E	E	E	U
Small number	Y	Y	Y	Y	Y	Y	Y	N	B	B	B	E
Special characters in data	Y	Y	Y	Y	Y	Y	Y	Y	B	E	E	U
Split value data	Y	Y	Y	Y	Y	Y	Y	Y	E	–	E	U
Strings in human-friendly formats	Y	Y	Y	Y	Y	Y	Y	Y	E	E	E	U
Suspect data distribution	Y	Y	Y	Y	Y	Y	Y	Y	E	B	B	B
Suspect data precision	Y	Y	Y	Y	Y	Y	Y	Y	E	E	B	B
Suspect date value	Y	Y	Y	Y	Y	Y	Y	Y	B	B	E	B
Suspect date/time interval	Y	Y	Y	Y	Y	Y	Y	Y	B	B	E	B
Suspect sign	Y	Y	Y	Y	Y	N	Y	Y	E	B	E	B
Synonyms in data	Y	Y	Y	Y	Y	Y	Y	Y	E	E	E	B
Unsound/missing metadata	Y	Y	Y	Y	Y	Y	Y	Y	E	E	E	E
Do												
Check columns names	Y	Y	Y	Y	Y	Y	Y	Y	E	B	E	E
Data cleaning	Y	Y	Y	Y	Y	Y	Y	Y	E	E	E	E
Data governance	Y	Y	Y	Y	Y	Y	Y	Y	E	E	B	B
Data profiling	Y	Y	Y	Y	Y	Y	Y	Y	E	E	E	E
Data usage consent	Y	Y	Y	Y	Y	Y	Y	Y	E	B	B	E
Data versioning	Y	Y	Y	Y	Y	Y	Y	Y	E	B	B	E
Document the dataset	Y	Y	Y	Y	Y	Y	Y	Y	E	E	E	B
Ensure data quality	Y	Y	Y	Y	Y	Y	Y	Y	E	E	B	E

Resource	Useful				Familiar				Effect of AI			
	1	2	3	4	1	2	3	4	1	2	3	4
Make data sets available on shared infrastructure (private or public)	Y	Y	Y	Y	Y	Y	Y	Y	E	B	E	E
Process raw data only once	N	N	Y	Y	Y	N	Y	Y	E	B	B	E
Remove outliers	N	Y	Y	Y	Y	Y	Y	Y	B	E	B	E
Responsible data management	Y	Y	Y	Y	Y	Y	Y	Y	B	B	B	B
Don't												
Class overlap	Y	Y	Y	Y	Y	N	Y	Y	E	–	B	B
ETL monolith	Y	Y	Y	Y	Y	Y	Y	Y	E	–	B	E
ETL monolith	Y	Y	Y	Y	Y	Y	Y	Y	E	B	E	E
No data duplication	Y	Y	Y	Y	Y	Y	Y	Y	E	E	B	B
No Data Silos	Y	Y	Y	Y	Y	Y	Y	Y	E	–	E	B
No duplicate data	Y	Y	Y	Y	Y	Y	Y	Y	E	E	E	E
Over-aggressive filtering	Y	Y	Y	Y	Y	N	Y	Y	E	B	B	B
Schema on read	N	N	Y	Y	Y	Y	Y	Y	E	E	B	E
Consider												
Data fabric architecture	Y	Y	Y	Y	Y	Y	Y	N	E	E	E	E
Data mesh architecture	Y	Y	Y	Y	Y	Y	Y	N	E	–	B	E
Data retention	Y	Y	Y	Y	Y	Y	Y	Y	E	B	B	E
Incremental data processing	Y	Y	Y	Y	Y	Y	Y	Y	E	B	B	E
Logical data warehouse	Y	Y	Y	Y	N	Y	Y	Y	E	B	B	E
Solve												
Data Lake Architecture	Y	Y	Y	Y	Y	Y	Y	Y	E	B	B	E
Data lakehouse architecture	Y	Y	Y	Y	Y	Y	Y	N	E	B	B	E
Data pipeline	Y	Y	Y	Y	Y	Y	Y	Y	E	E	B	E
Deep lake architecture	Y	Y	Y	Y	N	N	Y	N	E	B	E	E
Medallion architecture	Y	Y	Y	Y	Y	Y	Y	N	E	–	B	E
NoSQL	Y	Y	Y	Y	Y	Y	Y	Y	E	B	B	E

Table B.2: Answers of the experts on the complete descriptions (second part of the questionnaire).

Resource	Clear			Level of detail		
	1	2	3	1	2	3
Be aware						
Aggregation mismatch	Y	Y	Y	R	R	R
Ambiguous date/time format	Y	–	Y	R	–	C
Ambiguous value	Y	Y	Y	R	R	R
Column header containing value	Y	Y	Y	R	R	R
Data inconsistencies	Y	Y	Y	R	R	R
Data miscoding	Y	Y	Y	R	R	R
Data unit measure unknown	Y	Y	Y	R	R	R
Dummy value	Y	Y	Y	R	R	R
Extraneous value	Y	Y	Y	R	R	R
Extreme data value	Y	Y	Y	R	R	R
Incorrect column header	Y	Y	Y	R	R	R
Intermingled data type	Y	N	Y	R	–	R
Long data value	Y	Y	Y	R	R	R
Meaningless value	Y	Y	Y	R	R	R
Missing values (and metadata)	Y	Y	Y	R	C	R
Multi value data	Y	Y	Y	R	R	R
Small number	N	Y	Y	C	R	R
Special characters in data	Y	–	Y	R	C	R
Special characters in data	N	–	–	R	–	–
Split value data	Y	–	Y	R	C	R
Strings in human-friendly formats	Y	N	Y	R	–	R
Suspect data distribution	Y	Y	Y	R	R	R
Suspect data precision	Y	Y	Y	R	R	R
Suspect date value	Y	Y	Y	R	R	R
Suspect date/time interval	Y	Y	Y	R	R	R
Suspect sign	Y	Y	Y	R	R	R
Synonyms in data	Y	Y	Y	R	R	R
Unsound/missing metadata	Y	N	Y	R	C	R
Do						
Check columns names	Y	Y	Y	R	R	R
Data cleaning	Y	N	Y	R	C	R
Data governance	Y	N	Y	D	R	R
Data profiling	Y	–	Y	R	D	R
Data usage consent	N	Y	Y	C	R	R

Resource	Clear			Level of detail		
	1	2	3	1	2	3
Data versioning	Y	Y	Y	D	R	R
Document the dataset	Y	Y	Y	R	R	R
Ensure data quality	N	Y	Y	C	–	R
Make data sets available on shared infrastructure (private or public)	N	Y	Y	C	R	R
Process raw data only once	N	N	Y	C	C	R
Remove outliers	N	N	Y	C	C	R
Responsible data management	Y	N	Y	D	C	R
Don't						
Class overlap	N	Y	Y	R	R	R
ETL monolith	Y	Y	Y	R	R	R
ETL monolith	N	–	Y	R	–	R
No data duplication	N	Y	Y	R	R	R
No Data Silos	N	Y	Y	R	R	R
No duplicate data	N	N	Y	R	C	R
Over-aggressive filtering	Y	Y	Y	R	R	R
Schema on read	N	N	Y	C	C	R
Consider						
Data fabric architecture	Y	Y	Y	C	R	R
Data mesh architecture	Y	Y	Y	D	R	R
Data retention	Y	Y	Y	R	R	R
Incremental data processing	Y	Y	Y	R	R	R
Logical data warehouse	Y	Y	Y	D	C	R
Solve						
Data Lake Architecture	Y	Y	Y	R	R	R
Data lakehouse architecture	Y	Y	Y	R	–	R
Data pipeline	Y	Y	Y	R	R	R
Deep lake architecture	N	Y	Y	C	–	R
Medallion architecture	Y	Y	Y	R	C	R
NoSQL	N	–	Y	C	–	R

Bibliography

- [AG24] C. Aliferis and S. Gyorgy. Overfitting, underfitting and general model overconfidence and under-performance pitfalls and best practices in machine learning and ai. In Gyorgy S. and C. Aliferis, editors, *Artificial Intelligence and Machine Learning in Health Care and Medical Sciences: Best Practices and Pitfalls*, pages 477–524. Springer, 2024. doi: [10.1007/978-3-031-39355-6_10](https://doi.org/10.1007/978-3-031-39355-6_10).
- [Alu23] Nagateja Alugunuri. Progressive Delivery in CI/CD Pipelines: Evaluating Canary, Blue-Green, and Feature Flag Strategies. *International Journal of Intelligent Systems and Applications in Engineering*, 11(6s):922–933, 2023.
- [Ant23] S. Anttila. The hidden pitfalls of kanban in software development. Master’s thesis, Aalto University School of Business, 2023.
- [App98] Brad Appleton. Patterns in a nutshell. <http://www.bradapp.net/patterns/>, 1998. Accessed: 2025-11-23.
- [ASH⁺25] Dharun Anandayuvraj, Tanmay Singla, Zain A. H. Hammadeh, Andreas Lund, Alexandra Holloway, and James C. Davis. Learning from software failures: A case study at a national space research center. *arXiv preprint arXiv:2509.06301*, 2025. URL: <https://arxiv.org/abs/2509.06301>.
- [Azu25] Microsoft cloud adoption framework for Azure. Microsoft Learn documentation, 2025. <https://learn.microsoft.com/en-us/azure/cloud-adoption-framework/overview>.
- [Bar23] Doug Barrett. Incremental processing strategies, 2023. URL: <https://coalesce.io/product-technology/incremental-processing-strategies/#:~:text=Incremental%20processing%20strategies%20are%20used,in%20significantly%20lower%20compute%20costs>.

- [BBK03] Felix Bachmann, Len Bass, and Mark Klein. Deriving architectural tactics: A step toward methodical architectural design. Technical Report CMU/SEI-2003-TR-004, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2003. URL: <https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=6423>.
- [BCK13] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley, Boston, MA, 3rd edition, 2013.
- [BLR⁺24] A. Bhatia, D. Lin, G.K. Rajbahadur, B. Adams, and A.E. Hassan. Data quality antipatterns for software analytics, 2024. URL: arxiv.org/abs/2408.12560.
- [BMMM98] William J. Brown, Raphael C. Malveau, Hays W. “Skip” McCormick, and Thomas J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley & Sons, New York, NY, USA, 1998.
- [BMR⁺96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. Wiley Series in Software Design Patterns. John Wiley & Sons, Chichester, UK, 1996.
- [Bro24] Andrew Richard Brown. *Taming Your Dragon: Addressing Your Technical Debt*. Springer Nature Singapore, 2024. Copyright © 2024 by Dr. Andrew Richard Brown. doi:10.1007/979-8-8688-0264-5.
- [BVG21] J. Bogner, R. Verdecchia, and I. Gerostathopoulos. Characterizing technical debt and antipatterns in ai-based systems: A systematic mapping study. In *Proceeding 2021 IEEE/ACM TechDebt*, pages 64–73, 2021. doi:10.1109/TechDebt52882.2021.00016.
- [CDC23] Nicolas Cardozo, Ivana Dusparic, and Christian Cabrera. Prevalence of Code Smells in Reinforcement Learning Projects . In *2023 IEEE/ACM 2nd International Conference on AI Engineering – Software Engineering for AI (CAIN)*, pages 37–42. IEEE Computer Society, 2023. doi:10.1109/CAIN58948.2023.00013.
- [Cen24] Google Cloud Architecture Center. Guidelines for developing high-quality, predictive ML solutions. Google Cloud Web site, 2024. cloud.google.com/architecture/guidelines-for-developing-high-quality-ml-solutions.
- [CKB⁺23] G. C. Chouliaras, K. Kielczewski, A. Beka, D. Konopnicki, and L. Bernardi. Best practices for machine learning systems: An industrial framework for

- analysis and optimization, 2023. URL: arxiv.org/abs/2306.13662, [arXiv:2306.13662](https://arxiv.org/abs/2306.13662).
- [Cra24] Craftwork. Common Pitfalls When Implementing MLOps. Medium Web site, 2024. medium.com/@craftworkai/common-pitfalls-when-implementing-mlops-c6880930ab29.
- [Cun95] Ward Cunningham. Micropatterns. <http://wiki.c2.com/?MicroPatterns>, 1995. Accessed: 2025-11-23.
- [Dav95] Alan M. Davis. *Software Requirements: Objects, Functions, and States*. Prentice Hall, 1995. ISBN 0-13-846478-2.
- [Dez18] E. Dezhic. Machine Learning Rules in a Nutshell. Medium Web site, 2018. medium.com/data-science/machine-learning-rules-in-a-nutshell-dfc3e6839163.
- [DMG17] Paul M. Duvall, Steve Matyas, and Andrew Glover. *Continuous Delivery with Docker and Jenkins*. Addison-Wesley Professional, Boston, MA, 2017.
- [DVC] DVC: Data version control. <https://dvc.org>. Accessed: 2024.
- [eug23] eugeneyan. More Design Patterns For Machine Learning Systems. eugeneyan blog, 2023. URL: eugeneyan.com/writing/more-patterns/.
- [Fag76] Michael E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182–211, 1976. [doi:10.1147/sj.153.0182](https://doi.org/10.1147/sj.153.0182).
- [FFR22] H. Foidl, M. Feldere, and R. Ramler. Data smells: Categories, causes and consequences, and detection of suspicious data in ai-based systems. In *Proceedings of CAIN 2022*, pages 229–239. IEEE, 2022. URL: <https://doi.ieeecomputersociety.org/10.1145/3522664.3528590>, [doi:10.1145/3522664.3528590](https://doi.org/10.1145/3522664.3528590).
- [FOVF20] Eduardo Fernandes, Jorge Oliveira, Gustavo Vale, and Eduardo Figueiredo. Code smells and refactoring: A tertiary systematic review of challenges and observations. *Journal of Systems and Software*, 164:110610, April 2020. [doi:10.1016/j.jss.2020.110610](https://doi.org/10.1016/j.jss.2020.110610).
- [Fow99] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, 1999.

- [FS01] M. Fowler and K. Scott. *UML Distilled: Second Edition*. Object Technology Series. Addison-Wesley, 2001.
- [GALP23] Giammaria Giordano, Giusy Annunziata, Andrea De Lucia, and Fabio Palomba. Understanding developer practices and code smells diffusion in ai-enabled software: A preliminary study. In *Proc. of IWSM/MENSURA 23*. CEUR-WS.org, 2023. URL: ceur-ws.org/Vol-3543/paper18.pdf.
- [Gat22] Gatha. That Secret Sauce To Great ML Projects: Best Practices. Censius blog, 2022. URL: censius.ai/blogs/ml-development-best-practices.
- [GHJV94a] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Boston, MA, 1994.
- [GHJV94b] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Boston, MA, 1994.
- [Git] Git LFS: Git large file storage. <https://git-lfs.com>. Accessed: 2024.
- [GK18] Vahid Garousi and Baris Küçük. Smells in software test code: A survey of knowledge in industry and academia. *Journal of Systems and Software*, 138:52–81, 2018.
- [GMV⁺21] Timnit Gebru, Jamie Morgenstern, Briana Vecchione, Jennifer Wortman Vaughan, Hanna Wallach, Hal Daumé, and Kate Crawford. Datasheets for datasets. *Communications of the ACM*, 64(12):86–92, 2021.
- [Gup20] R. Gupta. Checklist For (Any) Machine Learning Project. Medium Web site, 2020. medium.com/@rahulgupta1/checklist-for-any-machine-learning-project-be276ab8a9d5.
- [Hal19] M. Hall. Machine learning project review checklist. Agile* Blog, 2019. agilescientific.com/blog/2019/4/9/machine-learning-project-review-checklist.
- [Hal20] M. Hall. Does your machine learning smell? Agile* Blog, 2020. agilescientific.com/blog/2020/12/15/does-your-machine-learning-smell.
- [HGE25] Z. Hakim, R. Gupta, and I. Ejindu. What are the most common machine learning software design patterns and unit patterns? LinkedIn Web site, 2025. [/www.linkedin.com/advice/0/what-most-common-machine-learning-software-design-patterns](https://www.linkedin.com/advice/0/what-most-common-machine-learning-software-design-patterns).

- [HHB23] L. Heiland, M. Hauser, and J. Bogner. Design patterns for ai-based systems: A multivocal literature review and pattern repository. In *Proceedings of CAIN 2023*, pages 184–196, 2023. doi:[10.1109/CAIN58948.2023.00035](https://doi.org/10.1109/CAIN58948.2023.00035).
- [HLB⁺24] J. Heli, P. Lagoc, J. Bogner, G. Lewis, H. Muccini, and I. Ozkaya. A synthesis of green architectural tactics for ml-enabled systems. In *Proceedings of ICSE-SEIS’24*, page 130–141. ACM, 2024. doi:[10.1145/3639475.3640111](https://doi.org/10.1145/3639475.3640111).
- [HNS00] Christine Hofmeister, Robert Nord, and Dilip Soni. *Applied Software Architecture*. Addison-Wesley, 2000.
- [HS25] N. Hahn and A. Sales. Data smells are sneaky. In J. Filipe, M. Smialek, A. Brodsky, and S. Hammoudi, editors, *Proceedings of ICEIS 2025, Volume 1*, pages 479–488. SCITEPRESS, 2025. doi:[10.5220/0013285500003929](https://doi.org/10.5220/0013285500003929).
- [HTK23] S. H. Al Harbi, L. Nganyewou Tidjon, and F. Khomh. Responsible design patterns for machine learning pipelines. arxiv.org Web site, 2023. URL: arxiv.org/abs/2306.01788.
- [ICIA24] Adebimpe Bolatito Ige, Naomi Chukwurah, Courage Idemudia, and Victor Ibukun Adebayo. Managing data lifecycle effectively: Best practices for data retention and archival processes. *International Journal of Engineering Research and Development*, 20(7), 2024. URL: www.ijerd.com.
- [Inf20] InfoQ. Frameworks require decisions. <https://www.infoq.com/articles/frameworks-require-decisions/>, 2020. Accessed: 2025-11-04.
- [Inq96] Inquiry Board. Ariane 5 - flight 501 failure: Inquiry board report. Technical report, European Space Agency, Paris, France, July 1996. Report on the failure of Ariane 5 Flight 501, 4 June 1996.
- [ISW25] V. Indykov, D. Struber, and R. Wohlrab. Architectural tactics to achieve quality attributes of machine-learning-enabled systems: a systematic literature review. *Journal of Systems and Software*, 223:112373, 2025. doi:[10.1016/j.jss.2025.112373](https://doi.org/10.1016/j.jss.2025.112373).
- [JBR97] Ivar Jacobson, Grady Booch, and James Rumbaugh. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, 1997.
- [Kao21] W. Shiang Kao. Machine Learning Project Checklist. DataRobot blog, 2021. www.datarobot.com/blog/machine-learning-project-checklist/.

- [KHDW16] Gene Kim, Jez Humble, Patrick Debois, and John Willis. *The DevOps Handbook*. IT Revolution Press, LLC, Portland, OR, 2016.
- [KN23] S. Kapoor and A. Narayanan. Leakage and the reproducibility crisis in machine-learning-based science. *Patterns*, 4(9), 2023. doi:[10.1016/j.patter.2023.100804](https://doi.org/10.1016/j.patter.2023.100804).
- [KZA⁺20] S. Khan, U. Zdun, P. Avgeriou, R. Capilla, and N. Harrison. Detecting, tracing, and monitoring architectural tactics in code. *IEEE Transactions on Software Engineering*, 46(10):1081–1102, 2020. doi:[10.1109/TSE.2018.2886874](https://doi.org/10.1109/TSE.2018.2886874).
- [Lak] LakeFS: Data version control for your data lake. <https://lakefs.io>. Accessed: 2024.
- [Lau23] G. Lauzier. Ethical guidelines for AI and machine learning. Medium Web site, 2023. URL: medium.com/@guillaumelauzier/ethical-guidelines-for-ai-and-machine-learning-af24504d46c8.
- [LLZ⁺22] Sin Kit Lo, Qinghua Lu, Liming Zhu, Hye-Young Paik, Xiwei Xu, and Chen Wang. Architectural patterns for the design of federated learning systems. *Journal of Systems and Software*, 191:111357, 2022. doi:[10.1016/j.jss.2022.111357](https://doi.org/10.1016/j.jss.2022.111357).
- [LMHPS24] D. Leuthe, T. Meyer-Hollatz, T. Plank, and A. Senkmüller. Towards sustainability of ai – identifying design patterns for sustainable machine learning development. *Information Systems Frontiers*, 26, 2024. doi:[10.1007/s10796-024-10526-6](https://doi.org/10.1007/s10796-024-10526-6).
- [LN05] Phillip A. Laplante and Colin J. Neill. *Antipatterns: Management, Development, and Modernization*. CRC Press, Boca Raton, FL, 2005.
- [Lon24] M. A. Lones. Avoiding common machine learning pitfalls. *Patterns*, 5(10), 2024. doi:[10.1016/j.patter.2024.101046](https://doi.org/10.1016/j.patter.2024.101046).
- [LSed] Leonard and Swanepoel. Project portfolio management implementation pitfalls. *SA Journal of Industrial Engineering*, Year Required.
- [LVC89] Mark A. Linton, John M. Vlissides, and Paul R. Calder. Composing user interfaces with InterViews. *Computer*, 22(2):8–22, 1989. doi:[10.1109/2.19829](https://doi.org/10.1109/2.19829).
- [LZX⁺24] Qinghua Lu, Liming Zhu, Xiwei Xu, Jon Whittle, Didar Zowghi, and Aurelie Jacquet. Responsible ai pattern catalogue: A collection of best practices

- for ai governance and engineering. *ACM Computing Surveys*, 56(7), 2024. doi:[10.1145/3626234](https://doi.org/10.1145/3626234).
- [Mak18] P. Makhfi. 9 pitfalls to avoid in building a successful machine learning program. DataDecisionMakers blog, 2018. URL: venturebeat.com/ai/9-pitfalls-to-avoid-in-building-a-successful-machine-learning-program/.
- [Mat20] J. Matthews. Patterns and antipatterns, principles, and pitfalls: Accountability and transparency in artificial intelligence. *AI Magazine*, 41(1):82–89, 2020. doi:[10.1609/aimag.v41i1.5204](https://doi.org/10.1609/aimag.v41i1.5204).
- [McA20] M. McAteer. Nitpicking Machine Learning Technical Debt. Matthew McAteer blog, 2020. matthewmcateer.me/blog/machine-learning-technical-debt/ NitpickingMachineLearningTechnicalDebt.
- [MDVC98] Gerard Meszaros, Jim Doble, John Vlissides, and James O. Coplien. *Pattern Languages of Program Design 3*. Addison-Wesley, 1998.
- [MFAFM17] S. Martínez-Fernández, C. P. Ayala, X. Franch, and H. M. Marques. Benefits and drawbacks of software reference architectures: A case study. *Information and Software Technology*, 91, 2017.
- [MHBLV⁺23] Anamaria Mojica-Hanke, Andrea Bayona, Mario Linares-Vásquez, Steffen Herbold, and Fabio A. González. What are the machine learning best practices reported by practitioners on stack exchange? arxiv.org web site, 2023. URL: <https://arxiv.org/abs/2301.10516>.
- [Mol23] M. E. Mollestad. Mitigating hidden technical debt in machine learning systems, 2023. Master thesis. URL: ntnuopen.ntnu.no/ntnu-xmlui/handle/11250/3094281.
- [MS23] A. Mailach and N. Siegmund. Socio-technical anti-patterns in building ml-enabled software: Insights from leaders on the forefront. In *Proceedings ICSe 2023*, pages 690–702, 2023. doi:[10.1109/ICSE48619.2023.00067](https://doi.org/10.1109/ICSE48619.2023.00067).
- [NBP24] R. Nazir, A. Bucaioni, and P. Pelliccione. Architecting ml-enabled systems: Challenges, best practices, and design decisions. *Journal of Systems and Software*, 207, 2024. doi:[10.1016/j.jss.2023.111860](https://doi.org/10.1016/j.jss.2023.111860).
- [OBI⁺22] D. O'Brien, S. Biswas, S. Imtiaz, R. Abdalkareem, E. Shihab, and H. Rajan. 23 shades of self-admitted technical debt: an empirical study on machine learning software. In *Proceedings of ESEC/FSE 2022*, page 734–746. ACM, 2022. doi:[10.1145/3540250.3549088](https://doi.org/10.1145/3540250.3549088).

- [Pet15] Rob Pettit. Chapter 13: Patterns and tactics. Lecture notes, SWE 443, George Mason University, 2015. Spring 2015.
- [Pho22] J. Phoenix. 11 feature engineering tactics for your machine learning project. JustUnderstandingData Blog, 2022. URL: [/understandingdata.com/posts/11-feature-engineering-tactics-for-an-ml-project/](https://understandingdata.com/posts/11-feature-engineering-tactics-for-an-ml-project/).
- [PRTV10] Ken Peffers, Marcus Rothenberger, Tuure Tuunanen, and Reza Vaezi. Design science research evaluation. In *Proceedings of the Design Science Research in Information Systems and Technology*, Las Vegas, NV, USA, 2010. Exact venue/year should be verified from the original publication source.
- [PZK22] Mahima Pushkarna, Andrew Zaldivar, and Oddur Kjartansson. Data cards: Purposeful and transparent dataset documentation for responsible AI. In *ACM Conference on Fairness, Accountability, and Transparency (FAccT)*, pages 1776–1826, 2022. doi:10.1145/3531146.3533231.
- [RIT23] RITHP. Avoiding Ethical Pitfalls in Data Science: A Checklist for Practitioners. Medium Web site, 2023. medium.com/@rithpansanga/avoiding-ethical-pitfalls-in-data-science-a-checklist-for-practitioners-153328115640.
- [RLS24] Chiara Rucco, Antonella Longo, and Motaz Saad. Efficient data ingestion in cloud-based architecture: a data engineering design pattern proposal. In *ITADATA2024: The 3rd Italian Conference on Big Data and Data Science*, 2024. arXiv:2503.16079v2. arXiv:2503.16079.
- [RLS26] Chiara Rucco, Antonella Longo, and Motaz Saad. Mind: A metadata-driven ingestion design pattern for efficient data ingestion. *Big Data Research*, 43:100574, 2026. doi:10.1016/j.bdr.2025.100574.
- [Rom24] N. Romanenko. Common Pitfalls in AI/ML Projects and How to Avoid Them. Svitla Blog, 2024. svitla.com/blog/common-pitfalls-in-ai-ml/.
- [RRDNP24] Gilberto Recupito, Raimondo Rapacciuolo, Dario Di Nucci, and Fabio Palomba. Unmasking data secrets: An empirical investigation into data smells and their impact on data quality. In *Proceedings of CAIN '24*, page 53–63. ACM, 2024. doi:10.1145/3644815.3644960.
- [RSL25] Chiara Rucco, Motaz Saad, and Antonella Longo. Formalizing etlt and eltl design patterns and proposing enhanced variants: A systematic framework for modern data engineering. *arXiv preprint arXiv:2511.03393*, November 2025. Preprint. Department of Innovation Engineering, University of Salento, Lecce, Italy. URL: <https://arxiv.org/abs/2511.03393>.

- [RW12] Nick Rozanski and Eoin Woods. *Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives*. Addison-Wesley, Boston, MA, 2nd edition, 2012.
- [Sah24] S. Sahu. AI Ethics Checklist. Medium Web site, 2024. medium.com/@ssahuupgrad-93226/ai-ethics-checklist-df1c11e56631.
- [Sam24] SamiraAlipour. A Comprehensive Guide to Outliers in Machine Learning: Detection, Handling, and Impact. Medium Web site, 2024. medium.com/samiraalipour/a-comprehensive-guide-to-outliers-in-machine-learning-detection-handling-and-impact-f7d965bba7a5.
- [SCvD22] Arumoy Shome, Luís Cruz, and Arie van Deursen. Data smells in public datasets. In *Proceedings of the CAIN '22*, page 205–216. ACM, 2022. doi: [10.1145/3522664.3528621](https://doi.org/10.1145/3522664.3528621).
- [Sha25] P. Sharma. Code Review Guidelines for Data Science Projects: Do's, Don'ts, and Metric Evaluation. Medium Web site, 2025. medium.com/@sharmapraveen91/code-review-guidelines-for-data-science-projects-dos-don-ts-and-metric-evaluation-65fe732e4846.
- [She25] S. Shevade. Avoid these 8 mistakes when using ai in healthcare. HackerNoon blog, 2025. URL: hackernoon.com/avoid-these-8-mistakes-when-using-ai-in-healthcare.
- [SHHJ22] H. Seung-Ho and C. Ho-Jin. Checklist for validating trustworthy ai. In *Proceedings of BigComp 2022*, pages 391–394, 2022. doi: [10.1109/BigComp54360.2022.00088](https://doi.org/10.1109/BigComp54360.2022.00088).
- [SKvdBHV20] A. Serban, Koen K. van der Blom, H. Hoos, and J. Visser. Adoption and effects of software engineering best practices in machine learning. In *Proceedings of the ESEM 2020*. ACM, 2020. doi: [10.1145/3382494.3410681](https://doi.org/10.1145/3382494.3410681).
- [SMS⁺22] S.M. Srinivasan, S. Mahbub, R.S. Sangwan, Y. Badr, and P. Mukherjee. Pattern language for designing distributed ai systems. In R. Qiu, W.K.V. Chan, W. Chen, Y. Badr, and C. Zhang, editors, *City, Society, and Digital Transformation*, pages 467–477. Springer, 2022.
- [Som16] Ian Sommerville. *Software Engineering*. Pearson Education, 10th edition, 2016.
- [Sra24] L. Sragne. Productionised machine learning has several chicken-and-egg situations. Linkedin Web site, 2024.

www.linkedin.com/posts/laszlosragner_productionised-machine-learning-has-several-activity-7069673304906690561-TZZx/.

- [SS18] Tushar Sharma and Diomidis Spinellis. A survey on software smells. *Journal of Systems and Software*, 2018. URL: <https://www.sciencedirect.com/science/article/pii/S016412121830030X>, doi: [10.1016/j.jss.2018.01.030](https://doi.org/10.1016/j.jss.2018.01.030).
- [SSK19] Daniel Szopinski, Thorsten Schoormann, and Dennis Kundisch. Because your taxonomy is worth it: Towards a framework for taxonomy evaluation. *Research Paper*, 2019. Presented at the European Conference on Information Systems (ECIS) 2019 Workshop.
- [STR24] C.C. Syreeta, V. Sai Teja, and P. Rayhaan. What are common data engineering anti-patterns to avoid? LinkedIn Web site, 2024. /www.linkedin.com/advice/3/what-common-data-engineering-anti-patterns-avoid-8rf7e.
- [SvdBV23] A. Serban, K. van der Blom, and J. Visser. Engineering best practices for Machine Learning. GitHub Web site, 2023. se-ml.github.io/practices/.
- [Til21] Tom Tillo. A checklist for data pre-processing before you build your machine learning model. Towards Data Science web site, 2021. URL: towardsdatascience.com/a-checklist-for-data-pre-processing-before-you-build-your-machine-learning-model-91d2d04dc53f/.
- [TT07] Yoshiki Tahara and Tetsuo Tamai. A mini pattern language for teaching object-oriented design. In *Proceedings of the 14th Conference on Pattern Languages of Programs (PLoP)*, Monticello, IL, 2007. ACM. Available online via ACM Digital Library.
- [U.S03] U.S. Department of Health & Human Services. Summary of the hipaa privacy rule. <https://www.hhs.gov/hipaa/for-professionals/privacy/laws-regulations/index.html>, 2003. Accessed: 2026.
- [Via22] A. Via. ML bias: guidelines to make ML models ethical and reduce risks of discrimination. Medium Web site, 2022. URL: medium.com/glovo-engineering/ml-bias-guidelines-to-make-ml-models-ethical-and-reduce-risks-of-discrimination-1c7165f74210.
- [VK24] A. Virmani and M. Kuppam. Mlops antipatterns and mitigation approaches. *Journal of Computer Trends and Technology*, 2024. doi: [10.14445/22312803/IJCTT-V72I2P102](https://doi.org/10.14445/22312803/IJCTT-V72I2P102).

- [WF24] C.Y. Wijaya and J. Ferrer. 6 Best Practices for Machine Learning. Non-Brand Data Web site, 2024. www.nb-data.com/p/6-best-practices-for-machine-learning.
- [WGM88] André Weinand, Erich Gamma, and Rudolph Marty. ET++ – an object-oriented application framework in C++. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 46–57, 1988.
- [Wij24a] C. Y. Wijaya. Building a Robust Machine Learning Pipeline: Best Practices and Common Pitfalls. Machine Learning Mastery blog, 2024. URL: machinelearningmastery.com/robust-machine-learning-pipeline-best-practices-common-pitfalls/.
- [Wij24b] C. Yudha Wijaya. Do you want to develop an ml model? avoid these data common pitfalls. Non-Brand Data web site, 2024. URL: www.nb-data.com/p/do-you-want-to-develop-an-ml-model.
- [WKY⁺22] H. Washizaki, F. Khomh, Y.G. Guéhéneuc, H. Takeuchi, S. Okuda, N. Natori, and N. Shioura. Software engineering patterns for machine learning applications (sep4mla): part 2. In *Proceedings of the 27th Conference on Pattern Languages of Programs*. The Hillside Group, 2022.
- [WMA19] Faisal Wedyan, Yousef Bani Mustafa, and Hussam Alzu’bi. Impact of design patterns on software quality: a systematic literature review. *IET Software*, 13(2):85–94, 2019. doi:10.1049/iet-sen.2018.5446.
- [WUKG19] H. Washizaki, H. Uchida, F. Khomh, and Y.G. Gueheneuc. Studying software engineering patterns for designing machine learning systems. In *Proceedings of IWESEP 2019*, pages 49–495, 2019. doi:10.1109/IWESEP49350.2019.00017.
- [XGLR21] B.E. Ximenes and Geber G. L. Ramalho. Concrete ethical guidelines and best practices in machine learning development. In *Proceedings of ISTAS 2021*, pages 1–8, 2021. doi:10.1109/ISTAS52410.2021.9728979.
- [ZC05] Didar Zowghi and Chad Coulin. Requirements elicitation: A survey of techniques, approaches, and tools. In *Engineering and Managing Software Requirements*, pages 19–46. Springer, Berlin, Heidelberg, 2005.
- [ZCvD22] Haiyin Zhang, Luís Cruz, and Arie van Deursen. Code smells for machine learning applications. In *Proceedings of CAIN ’22*, page 217–228. ACM, 2022. doi:10.1145/3522664.3528620.

- [Zdr24] O. Zdrok. 9 AI Implementation Pitfalls That Can Cripple Any Project. Shelf Blog, 2024. shelf.io/blog/9-ai-implementation-pitfalls/.
- [ZMZ23] J.-J. Zhu, Yang M, and J. R. Zhiyong. Machine learning in environmental research: Common pitfalls and best practices. *Environmental Science & Technology*, 2023. doi:[10.1021/acs.est.3c00026](https://doi.org/10.1021/acs.est.3c00026).
- [Zwi22] T. Zwingmann. 7 Painful Pitfalls Of Machine Learning And How To Avoid Them”. Typeshare Web site, 2022. URL: [typeshare.co/tobiaszwingmann/posts/-Mt4Vj8QgmHfu`vjXc-7](https://typeshare.co/tobiaszwingmann/posts/-Mt4Vj8QgmHfu`vjXc-7).