Semi-Automated Design of Data-Intensive Architectures

Arianna Dragoni
Politecnico di Milano
Milano, Italy
arianna.dragoni@polimi.it

Alessandro Margara
Politecnico di Milano
Milano, Italy
alessandro.margara@polimi.it

***Abstract*—Today, data guides the decision-making process of most companies. Effectively analyzing and manipulating data at scale to extract and exploit relevant knowledge is a challenging task, due to data characteristics such as its size, the rate at which it changes, and the heterogeneity of formats. To address this challenge, software architects resort to build complex data-intensive architectures that integrate highly heterogeneous software systems, each offering vertically specialized functionalities. Designing a suitable architecture for the application at hand is crucial to enable high quality of service and efficient exploitation of resources. However, the design process entails a series of decisions that demand technical expertise and in-depth knowledge of individual systems and their synergies.**

To assist software architects in this task, this paper introduces a development methodology for data-intensive architectures, which guides architects in (i) designing a suitable architecture for their specific application scenario, and (ii) selecting an appropriate set of concrete systems to implement the application.

To do so, the methodology grounds on (1) a language to precisely define an application scenario in terms of characteristics of data and requirements of stakeholders; (2) an architecture description language for data-intensive architectures; (3) a classification of systems based on the functionalities they offer and their performance trade-offs.

We show that the description languages we adopt can capture the key aspects of data-intensive architectures proposed by researchers and practitioners, and we validate our methodology by applying it to real-world case studies documented in literature.

***Index Terms*—software architectures, data-intensive architectures, architecture definition language, architecture design, design methodology.**

I. Introduction

As data increasingly drive decision-making processes, data management becomes a key concern for modern organizations, which need to acquire, persist, analyze, and make data available to internal and external stakeholders based on their needs. Over the years, specialized systems have been designed to manage and process data at scale [1]. Each of them is optimized for specific tasks but no single system meets all organizational requirements [2]. As a consequence, engineers resort to build complex *data-intensive architectures* that integrate multiple such systems to meet the demand of the scenario at hand.

Devising a suitable architecture for a given scenario is challenging, as it requires careful consideration of numerous factors, including input data volume, generation frequency, and format, the type of processing to be performed, the scale of the scenario in terms of number and heterogeneity of data consumers, just to mention a few. To mitigate this problem, various reference data-intensive architectures have been defined in the literature, such as the well-know lambda and kappa architectures [3]. However, engineers still need to manually select the most suitable architecture for their application scenario. Moreover, reference architectures are abstract and high-level, thus requiring additional work to properly configure them for the requirements of the company. In summary, designing a data-intensive architecture for a given application scenario frequently remains a manual and unstructured process, where the software architect's decisions often remain implicit and undocumented.

Building on this observation, we propose a methodology to systematize and semi-automate the definition of data-intensive architectures. Our methodology works in two steps. (1) It takes in input the description of the application scenario and automatically translates it into a suitable data-intensive architecture. (2) It proposes concrete software systems to implement the architecture and provide all the required functionalities. By exploiting a rigorous process, the methodology helps engineers in taking the critical choices behind the definition of data-intensive architectures, making the motivations behind each choice explicit and well documented, thus also simplifying future evolution. To make this possible, we propose formal languages to precisely describe application scenarios, data-intensive architectures, and distributed data management and processing systems.

In summary, our paper brings the following contributions. (1) It proposes a *scenario description language* that software architects can adopt to precisely describe their application scenario, including its characteristics and requirements. Acknowledging the central role of data for architectural concerns [4], the language focuses on data characteristics and data consumers requirements. (2) It introduces an *architecture description language* to describe data-intensive architectures. We show that the language can capture reference architectures from the literature and detail the specific way in which they are configured to fulfill the scenario requirements. (3) It maps the components of a data-intensive architecture to concrete data management and processing systems based on the functionalities and guarantees they offer. This mapping builds on a

modeling and classification study from the recent literature [1]. (4) Based on the above conceptual framework, it introduces a novel semi-automated methodology to define a data-intensive architecture starting from a given application scenario.

The paper describes the methodology in details and evaluates its effectiveness in capturing the key requirements of application scenarios and converting them to suitable software architectures.

The paper is organized as follows. Section II presents the languages we use to define application scenarios and software architectures. Section III details our methodology for deriving data-intensive architectures from application scenarios. Section IV evaluates the methodology in terms of expressivity, effectiveness, and efficiency. Finally, Section V surveys related work and Section VI concludes the paper indicating future research directions.

II. Describing Software Scenarios and Architectures

This section overviews our methodology (Section II-A) and presents the formalism we use to define application scenarios (Section II-B), data-intensive architectures (Section II-C), and the technologies used to implement architectural components (Section II-D).

A. Overview

Fig. 1 provides a visual overview of our proposed methodology. Engineers define their application scenario by providing a *scenario description*. The methodology works in two steps: (1) the *architecture definition* step produces an *architecture description* consisting of abstract components that communicate with each other to realize the overall behavior of the application; (2) the *selection of systems* step proposes data management and processing systems to implement the components of the architecture, thus realizing a *concrete architecture*.

We make this possible by introducing formal languages for scenario and architecture descriptions. The *scenario description language* (SDL – Section II-B) defines the main functional and non-functional requirements of a scenario. The *architecture description language* (ADL – Section II-C) specifies abstract components as core building blocks for data-intensive architectures. We use an existing *systems model and taxonomy* [1] to recommend systems for implementing each abstract component.

This section presents the SDL and ADL in detail, with their concepts and relations illustrated in Fig. 2 as a UML diagram. Section III details the logic behind the two methodology steps: architecture definition and selection of systems.

B. Scenario Description Language (SDL)

The SDL (upper part of Fig. 2) formalizes application scenarios as stages data transformation that address user needs.

Scenarios are modeled as directed graphs, where *nodes* abstract operations on data (generation, manipulation, consumption) and *edges* denote data flows. We denote the source nodes as *producers* that generate input data for the application, sink nodes as data *consumers*, and intermediate nodes as *actions* that transform input data into output data.

Data consists of immutable elements called *data items* that are transferred from node to node across edges. Each edge e has an associated *data type* that indicates the common structure (if any) of all data items traversing e. Data types are categorized as *structured*, *unstructured*, or *semistructured*. A structured data type imposes a fixed schema to all data items, meaning that all data items consist of the same list of typed attributes. For instance, in an environmental monitoring application, temperature data items may all be characterized by a location (of type string), a timestamp (of type long), and a value (of type float). Conversely, an unstructured data type does not impose any constraint, as in the case of free text documents. Semistructured data types sit in between: they define a structure but they allow some degree of flexibility in the number of attributes. For instance XML and JSON objects may include variable-length lists or maps. Each edge also has a *frequency* attribute, indicating how often downstream nodes request data from that edge.

Each producer node in a scenario represents a set of real-world entities that produce data with similar characteristics, for instance a set of similar sensors in an environmental monitoring scenario. Similarly, each consumer node represents a set of users that are interested in the same results – that is, they require the same actions to be applied on input data – and have the same requirements in terms of frequency of requests and guarantees on data. The frequency attribute of a consumer's incoming edge models its request frequency. Furthermore, each consumer has an associated *delivery guarantee* property, which indicates whether the consumer tolerates loss of items (items are delivered with *at-most once* guarantees), duplicates (*at-least once* guarantees), or it requires all results to be delivered as if all input items were processed once and only once (*exactly once* guarantees). For instance, a consumer in an environmental monitoring scenario may be interested in weekly average of temperatures (result of actions), updated every second (frequency of requests), and may tolerate loss of data (at-most once delivery guarantee).

A scenario may involve different consumers, indicating different requirements for data consumption. In these cases, we denote as *data flow* for a consumer c the sub-graph that includes all and only the nodes that are directly or indirectly connected to c and the edges between these nodes. Intuitively, a data flow shows the input data and actions needed to meet a specific customer's requirements.

Actions have associated *input cardinality* and *output cardinality* attributes, which indicate the amount of data considered and generated by the action at each evaluation. These attributes serve as proxies for action complexity, as further detailed when discussing the computational costs in the ADL – cfr Section II-C. We consider two types of actions. (1) *Processing* actions have a single incoming edge. (2) *Merge* actions, by contrast, integrate data coming from multiple incoming edges. Both processing and merge actions may have one or more outgoing edges. Processing actions can be further specialized

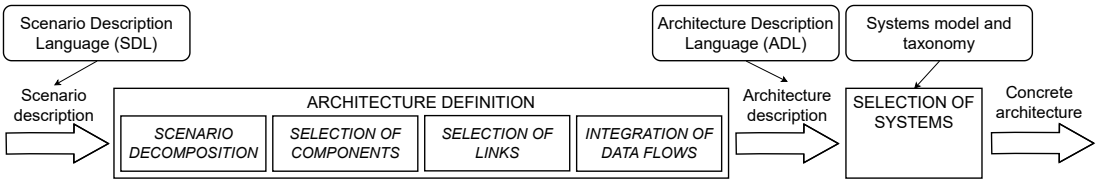


Fig. 1: Overview of the methodology.

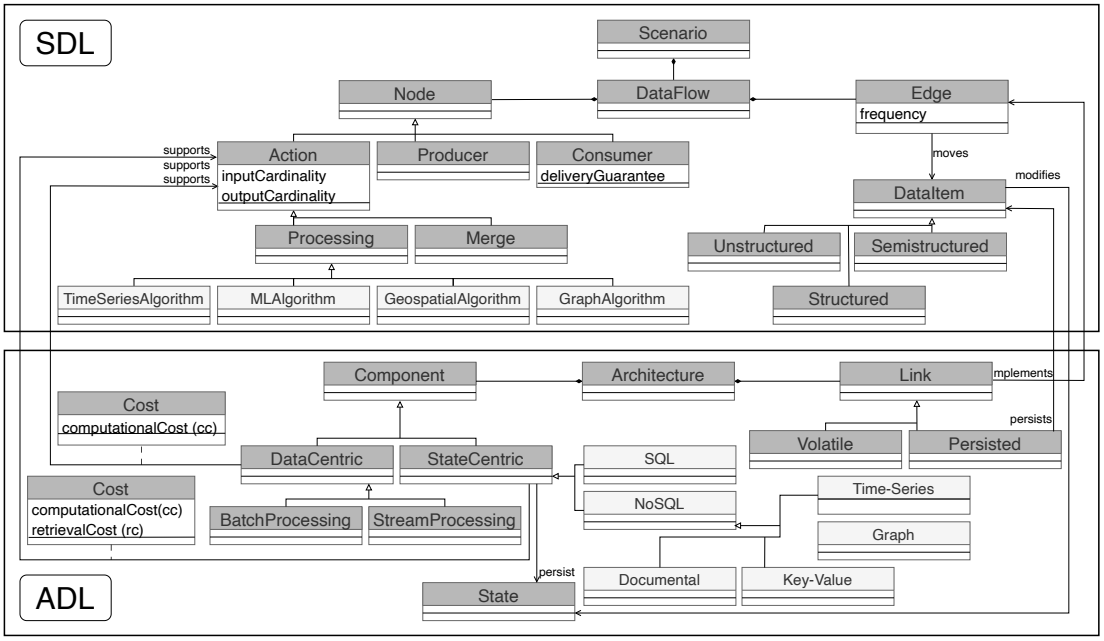


Fig. 2: UML class diagram for the Scenario Description Language (SDL) and the Architecture Description Language (ADL).

based on the type of data transformations they define. To ensure adaptability across contexts, we avoid prescribing a fixed catalog of transformations. Depending on the specific application domain, transformations may represent time series analysis, machine learning algorithms, graph algorithms, or other. In Fig. 2 illustrates examples, without aiming for exhaustiveness.

C. Architecture Description Language (ADL)

The ADL (lower part of Fig. 2) formalizes data-intensive architectures as graphs of software *components* that communicate by exchanging data items through *links*. Components model classes of systems with similar characteristics and links represent technologies to transfer data across systems.

The ADL captures (1) the capabilities that each software component exposes, and (2) the way in which software components receive and deliver data items. Methodologically, we developed the ADL iteratively, starting from both reference architectures [3] and real-world use cases [5] documented in the literature, and refining it until we could capture all the key features we extrapolated from the literature. We evaluate the modeling capabilities of the language in Section IV.

Each component *supports* a set of actions it can implement. For example, a component that supports a time series algorithm is capable of handling input data in the form of time series and to perform the specified algorithm. Similarly, each link *implements* an edge, meaning it implements a way to transfer data across the actions required in a scenario. As discussed in Section III, the first step of our methodology selects suitable components and links to implement the actions and edges defined in a scenario.

Links may either persist or not persist data elements traversing them, so we provide two specialization: *persistent* and *volatile* links. For example, links that model TCP connections are volatile (they deliver the data from sender to receiver, but do not store it for future retrieval), whereas links that model persistent queuing systems such as Apache Kafka or distributed filesystems such as HDFS enable receiving components to retrieve data more than once. The ADL is open to future extensions and refinements through specializations of these two classes.

Data items traversing links represent domain information, for instance new purchases performed by the customers of an e-commerce application. Components transform input data into output data according to the action they implement. For instance, a recommender system transforms input data (customers preferences) into output data (personalized recommendations). We classify components into two types: (i) *Data-centric* components apply data transformations either contin-

uously, as new data items are received, or upon request, and deliver their results through outgoing links. *Stream processing* components continuously transform data, as in the case of online anomaly detection systems that signal anomaly alerts by searching for suspicious patterns in the recent history of received items. *Batch processing* components compute data on demand or periodically, as in the case of systems that re-train a machine learning model daily, using the entire data available when they perform their computation. (ii) *State-centric* components store an internal *state* representing their view of the application domain. Upon receiving input data items, they update their internal state and make it accessible to downstream components through queries. For instance, a data store may store the last 5 purchases of each user and make them accessible upon request.

Both data-centric and state-centric components perform actions. However, data-centric components actions produce output data directly, while state-centric components actions update the component's internal state. Accordingly, we model the *cost* of performing an action a on a component c differently for the two cases (cfr *Cost* association classes in Fig. 2): data-centric incur a *computational cost* every time they execute the action, while state-centric component pay an *update cost* for updating their state at each execution of an action and a *retrieval cost* to retrieve information stored in their state and propagate it to downstream components. These costs are modeled as functions of the input and output data sizes, as specified by the action's input and output cardinality attributes. Currently, architects are expected to define cost functions for the considered components and actions. In the future, the adoption of this methodology may lead to a shared library of cost functions for common components.

D. Systems model and taxonomy

Given the evolving landscape of data-intensive systems, we avoid fixed component categorizations. In the ADL in Fig. 2, we follow the established classification of data-centric components into batch and stream processing systems: this is the level of granularity at which the current implementation of our methodology works. We further show examples of classes of systems that reflect the current state of the field according to recent work [1] (light gray classes in Fig. 2 – e.g., time-series or graph processing systems), and we foresee future evolution and specializations through extensions. Our methodology is agnostic of the specific characteristics of each class of systems, and only requires knowledge about the actions that each class (i.e., component) supports and an estimate of the cost functions associated to running a given action on a given component. Being conceived as a decision support tool, precise estimates of costs are not necessary, but a good indication of the relative differences between components can provide useful suggestions to software architects.

III. Methodology

This section describes the two steps of our methodology – *architecture definition* (Section III-A) and *selection of systems* (Section III-B) – in detail (cfr Fig. 1).

A. Architecture definition

The first step of the methodology translates a scenario description expressed in SDL into an architecture description expressed in ADL. It (i) decomposes a scenario into its constituting data flows; (ii) selects suitable components for each action in each data flow; (iii) selects suitable links for each edge in each data flow; (iv) merges individual data flows into a single architecture.

Scenario decomposition. The methodology decomposes the input scenario into individual data flows, where each data flow includes all and only the actions that are required to satisfy the requests of a single consumer. Nodes shared among multiple data flows are repeated within each of the data flows they appear in.

Selection of components. Each data flow is converted into an architecture description, where each node is translated into a component and each edge is translated into a link.

First, the methodology selects the most suitable type of component to implement each action. We formulate this choice as an optimization problem that associates a cost to each selection of components and aims to minimize the overall cost for each data flow.

Given a data flow DF, we denote the set of its internal nodes (excluding producers and consumers) as N and the sets of its edges as E. Each internal node $n \in N$ defines an action a_n to be performed on its input data. Recall that in a data flow a node can have multiple incoming edges (in the case of a merge node) and multiple outgoing edges. We denote the incoming frequency f_n^{in} of a node $n \in N$ as the maximum of the frequencies associated to its incoming edges, and the outgoing frequency f_n^{out} of n as the maximum of the frequencies associated to its outgoing edge. Intuitively, the incoming frequency represents the maximum frequency at which input data is requested to upstream components, and the outgoing frequency is the frequency at which new results are needed for (requested by) downstream component.

For each node $n \in N$, the optimization problem needs to decide the best component to implement n. We consider three macro classes of components: state-centric, data-centric batch processing, data-centric stream processing. We assume that the cost functions for implementing action a_n using a given class of components is known and fixed for any concrete system belonging to that class and supporting action a_n. As discussed earlier, a coarse-grained estimate of the costs for a given class of systems is sufficient for the purpose of a decision support tool. However, our conceptual framework is open to extension through the definition of finer grained classes of components if needed.

We encode the decision with three Boolean variables x_n^{sc}, x_n^{dc-b}, x_n^{dc-s} for each node $n \in N$, where $x_n^{sc} = 1$ indicates that node n is implemented with a state-centric component, $x_n^{db-b} = 1$ indicates that node n is implemented with a data-centric batch processing component, and $x_n^{db-s} = 1$

indicates that node n is implemented with a data-centric stream processing component. Each node $n \in N$ is implemented by exactly one component, as modeled by the following constraint.

$$\forall_{n \in N} \;\; x_n^{sc} \; + \; x_n^{db-b} \; + \; x_n^{db-s} \; = \; 1$$

We call c_n^{sc} the cost for implementing action a_n of node n using a state-centric component. A state-centric component pays a computational cost cc for each input data item, which represents the cost of computing the action and updating the state with the result of the computation, and a retrieval cost rc for each request coming from downstream components. We assume cc to be a function of the input data to be used at each computation, as modeled by the input cardinality attribute ic_{a_n}, and rc to be a function of the output data to be delivered downstream for each request, as modeled by the output cardinality attribute out_{a_n}. Therefore, its cost depends on the incoming and outgoing frequencies of n, and on the input and output cardinality ic_{a_n} and out_{a_n} as follows.

$$c_n^{sc} \; = \; f_n^{in} \cdot cc(ic_{a_n}) \; + \; f_n^{out} \cdot rc(out_{a_n})$$

We call c_n^{dc-b} the cost for implementing action a_n of node n using a data-centric batch-processing component. This type of components pay a computational cost cc every time the computation is triggered (based on the request of downstream components). We assume the computational cost to be a function of the input data to be used at each computation, as modeled by the input cardinality attribute ic_{a_n} for action a_n. Therefore, the cost c_n^{dc-b} depends on the outgoing frequency of n and the input cardinality of a_n, as follows.

$$c_n^{dc-b} \; = \; f_n^{out} \cdot cc(ic_{a_n})$$

We call c_n^{dc-s} the cost for implementing action a_n of node n using a data-centric stream-processing component. This type of components pay a computational cost cc every time a new data item enters the node, which depends on the input cardinality attribute ic_{a_n} for action a_n. Therefore, the cost c_n^{dc-s} depends on the incoming frequency of n and the input cardinality of a_n as follows.

$$c_n^{dc-s} \; = \; f_n^{in} \cdot cc(ic_{a_n})$$

If an action is not supported by a class of components, we assume its cost to be infinite.
The objective of the optimization problem is to minimize the overall cost:

$$\min \left(\sum_{n \in N} \; c_n^{sc} \cdot x_n^{sc} \; + \; c_n^{dc-b} \cdot x_n^{dc-b} \; + \; c_n^{dc-s} \cdot x_n^{dc-s} \right)$$

Selection of links. After selecting the components for implementing each action, the methodology decides the type of links used to implement edges that transfer data between components. As for the case of components, we currently consider two macro-classes of links, namely persistent and volatile links, where persistent links retain data items, allowing the outgoing component to retrieve them multiple times.

To decide whether a link is persistent or volatile, we consider the requirements of the component that consumes data from that link. Let us denote C the consumer of data flow DF, and $P = \{p^1, \ldots, p^n\}$ the set of producers of DF. Let us denote u_l and d_l the upstream and downstream components connected by link l. The selection algorithm works as follows.

1) If the upstream component is a producer, that is, $u_l \in P$, and C requires at least or exactly once delivery guarantees, l is a persistent link. Intuitively, this choice ensures that input data is persisted as soon as it enters the system and can be replayed in the case of failures to satisfy the requirement of the consumer in terms of delivery guarantees.
2) Otherwise, link l is volatile. Further cases that require persistent links will be considered after integrating individual data flows, as detailed later.

Integration of data flows. After determining components and links for each data flow independently, the methodology combines the data flows to define a single architecture.

Let us denote DF the set of data flows composing a scenario, and N the set of nodes in that scenario. Recall that a single node $n \in N$ may be replicated within multiple data flows, and for each data flow the methodology has associated a component to each node based on the specific requirements of that data flow.

Given a node $n \in N$ that is part of a data flow $DF_i \in DF$, we denote c_n^i the component associated to n in DF_i. Let us define $DF^n = \{DF_1^n, \ldots DF_m^n\} \subseteq DF$ the set of data flows that include node $n \in N$, and C^n the components that implement node n in the data flows in DF^n.

The methodology selects the minimum set of components required to implement node n as follows:

1) If C^n contains both a data-centric stream processing component and data-centric batch processing component, only the stream processing component is preserved. Indeed, both components compute the same results, and stream processing provides stricter guarantees in terms of response time for downstream components.
2) If C^n contains a state-centric component, the component is preserved. Indeed, the component may be used to store the results of processing and make it available to downstream components upon request.

Notice that this procedure enables a single node to be implemented through multiple components: a data-centric one and a state-centric one. The semantics of this case is that the data-centric component is used to perform the computation, and its results are stored in a state-centric component for later retrieval.

At the end of this process, the methodology has selected a minimum set of components C^n to implement each node $n \in N$. At this point, the methodology decides the links for connecting them. We preserve any link l connecting an

upstream component u_l and a downstream component d_l in at least one data flow. We select the persistency of a link l as follows.

1) A link l connecting a producer to an internal component is persistent if it is set as persistent in at least one data flow. Intuitively, the requirement for persistency derives from the delivery guarantees set by consumers.
2) If the components associated to the downstream node of l – that is, C^{d_l} – include a batch processing component, and the set of components associated to the upstream node – that is, C^{u_l} – does not include a state-centric component, then l is persistent. Intuitively, a batch processing component needs to retrieve its input data for each computation. This can be done by pulling from the upstream component in the data flow, if that component is state-centric, or by accumulating data into a persistent link.

B. Selection of systems

Components and links in architecture descriptions represent generalizations of concrete technologies that are suitable to satisfy the requirements of a given scenario. Specifically, components represent classes of data management or data processing systems that support the actions defined in the nodes of a scenario. Likewise, links represent technologies and systems for data transfer that are suitable to connect components and satisfy the requirements of consumers.

Moving from components to concrete systems involves aspects that go beyond the technical characteristics of each system, such as monetary costs or previous knowledge of specific systems. Both the set of available systems and the motivations to favor one system over another may change over time. For these reasons, our methodology does not select *a single* system for each component in the architecture, but rather lists the systems that concretely provide the features defined in the component. For instance, it lists suitable NoSQL data stores to implement NoSQL state-centric components or distributed stream processors to implement data-centric stream processing components.

This choice enables our methodology to evolve over time and to specialize if better categorizations of systems become available for specific application domains. Currently, we rely on a taxonomy of systems discussed in recent literature [1]. This taxonomy guides both the classifications of components and the list of systems that may implement each component, which guarantees the consistency between architecture definition and selection of systems.

IV. Evaluation

In this section, we evaluate our methodology in terms of modeling capabilities, effectiveness, and efficiency. Specifically, our evaluation aims to answer the following three research questions.

(RQ1) Is the methodology suitable to model real-world scenarios and architectures?
(RQ2) Does the methodology guide software architects towards appropriate architectures for a given scenario?
(RQ3) What is the time required to compute an architecture starting from a scenario?

To answer the questions above, we organize our evaluation in three parts.

1) We analyze four reference architectures for data-intensive applications from the literature, and we show that our languages are indeed suitable to model the requirements that motivate these architectures (for the SDL) and their structure in terms of components and links (for the ADL) – Section IV-A.
2) We rely on a concrete use case discussed in the literature [6], we model its requirements using our SDL, and we show that the methodology generates an architecture description that closely resembles the one in the original paper – Section IV-B.
3) We execute our methodology on synthetic scenarios of increasing complexity, and we show that the methodology provides suggestions within tens of seconds even for the most demanding scenarios with hundreds of nodes – Section IV-C.

The code of our methodology, the scenarios used in the evaluation, and the scripts used to conduct the experiments are available at https://github.com/deib-polimi/methodology_simulations.git.

A. Modeling capabilities

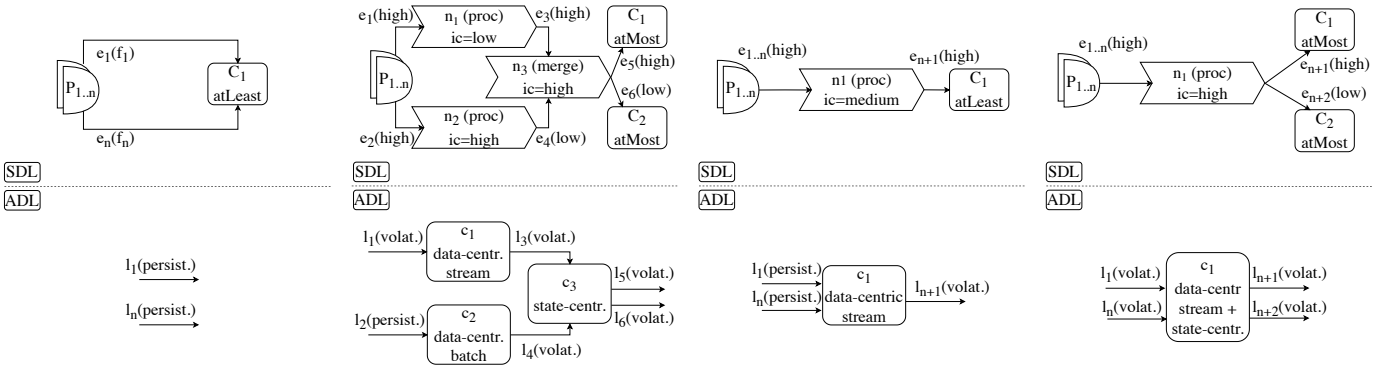
To validate the modeling capabilities of our languages, we rely on four reference architectures that are frequently discussed in the literature [3], [7]: data lake, liquid, lambda, and kappa. These are high-level architectural patterns that practitioners have defined to capture the landscape of data-intensive applications.

We extract from the literature the requirements that guide towards the use of a given architecture, and we use them to model corresponding scenarios using our SDL. We only assign attributes that are relevant for a given architecture, leaving the others unspecified. We run our methodology on such scenarios and we verify that we indeed obtain the reference architectures as presented in the literature.

Fig. 3 illustrates the scenario and architecture descriptions for the reference architectures. Next, we report relevant observations that derive from modeling the architectures within the framework of our methodology.

Data lake. The data lake architecture captures the requirements of storing data produced at different frequencies and formats from heterogeneous producers.

We model these requirements using our SDL as shown in the upper part of Fig. 3a. We represent the producers as $P_1 \dots P_n$, which produce data with frequencies $f_1 \dots f_n$ over the edges $e_1 \dots e_n$. The data lake architecture does not prescribe any specific processing to be performed on such data. Accordingly, in our SDL, we model the processing logic by means of a



(a) Data Lake (b) Lambda (c) Liquid (d) Kappa

Fig. 3: Scenario (top) and architecture (bottom) descriptions for reference architectures.

single consumer (C_1) that requires data to be persisted (at least once delivery guarantee).

Our methodology produces the architecture description shown in the lower part of Fig. 3a. It correctly identifies the need for storing input data as it enters the system, and models this need using persistent ingestion links – $l_1 \dots l_n$ in Fig. 3a.

Lambda. The lambda architecture captures the requirements of performing different types of computations on input data. These computations range from lightweight processing of high-frequency data to heavy computations on large volumes of data (e.g., historical data) that cannot be easily performed in an incremental fashion.

We model these requirements in our SDL (upper part of Fig. 3b) using two processing nodes n_1 and n_2, which consume different volumes of data at each evaluation, as modeled through different input cardinality ic in Fig. 3b (indicated as low for n_1 and $high$ for n_2). For simplicity, we omit the output cardinality in all the actions in Fig. 3, and we always assume it to be low, meaning that the volume of data is reduced after performing an action. The results of n_1 and n_2 are delivered to consumers. The need for integrating these results is modeled through a merge node – n_3 in Fig. 3b. Consumers request output data with different requirements in terms of frequency. We model them using C_1 and C_2 in Fig. 3b, where C_1 consumes data with high frequency through edge e_5 and C_2 consumes data with low frequency through edge e_6.

Our methodology produces the architecture description shown in the lower part of Fig. 3b. It suggests using two separate data-centric components, a stream processing one (c_1 in Fig. 3b) for lightweight computations and a batch processing one (c_2 in Fig. 3b) for heavy computations. Regardless of the delivery guarantees of producers, it always suggests persisting data before ingesting it into the batch processing component. Furthermore, the architecture persists the results of heavy computations within a state-centric merge component, allowing consumers to retrieve them on demand.

Liquid. The liquid architecture presents two differences with respect to the lambda architecture in terms of requirements: (i) Heavy computations can be easily made incremental, thus reducing their processing cost upon receiving a new input data item. We model this difference by exploiting a single processing node n_1, without differentiating the computational costs of different operations – modeled through a medium input cardinality ic – cfr Fig. 3c. (ii) All consumers request results at high frequency. We model this requirement through a single consumer C_1 in Fig. 3c.

Our methodology produces the architecture description shown in the lower part of Fig. 3c. It captures the differences with respect to the lambda architecture and proposes a data-centric stream processing component for all input data – Fig. 3c.

Kappa. The kappa architecture serves consumers with different needs, which we model in the scenario description in the upper part of Fig. 3d using consumers C_1 and C_2 with different input frequencies (high and low, respectively). Computations on input data may become expensive as the input data grows, but the typical adoption of this architecture reduces the amount of data to be considered (e.g., by limiting the temporal range under analysis for computations on historical data). In this way, the computational cost remains similar across operations. In our scenario description, we model this assumption through a single processing node n_1.

Our methodology produces the architecture description shown in the lower part of Fig. 3d. It captures the different requirements of the consumers by suggesting two different components for node n_1 – a data-centric stream processing one and a state-centric one. The state-centric component stores the results of stream processing computations, making them available on demand to consumers with low request frequency.

B. Effectiveness

To measure the effectiveness of our methodology, we rely on the concrete use case discussed in [6], which describes the requirements and the internal architecture of Facebook.

Fig. 4 illustrates the scenario description we extrapolated based on the information in [6]. We identify two producers:

P_1, which continuously delivers event logs and clicks from users, and P_2, which delivers users information daily.

A processing node n_1 aggregates logs from P_1. Based on information from the application domain, we assume that n_1 receives data with a rate in the order of milliseconds (edge e_1 in Fig. 4), and the aggregation is performed on groups of tens of input data items that reduced to a single output data item (that is, the input cardinality ic for node n_1 is 10 and its output cardinality oc is 1).

A merge node n_2 processes data coming from n_1 and P_2: as stated in [6], logs from n_1 are evaluated at intervals of 5–15 minutes, and user data from P_2 at daily intervals. We assume the merge action to consume and output large volumes of data. In the scenario in Fig. 4, we model this assumption by assigning an input cardinality parameter to n_2 that is 10 times larger than the input cardinality of n_1 and an output cardinality parameter that is equal to the input cardinality.

According to [6], there are three classes of users – that is, consumers in our SDL, which we model with consumers C_1, C_2, C_3: C_1 demands simple but frequent computations (we assume a rate of requests in the order of milliseconds), C_2 and C_3 demand complex computations with a rate of hours or days, respectively. We model the computations required by each of these consumers with three additional processing nodes: n_3 processes data for consumer C_1 – we model the simplicity of the processing task by assigning an input and output cardinality of 1; n_4 and n_5 process data for consumers C_2 and C_3, respectively – we model the complexity of these tasks by assigning an input cardinality of 100 and an output cardinality of 10. Although this is not specified in [6], we assume consumers to tolerate loss of data. As discussed later, this choice does not affect the results obtained with our methodology.

Since there is no detailed information on the specific actions performed by nodes, we assumed computational cost functions to be linearly proportional to the input cardinality and to be identical for all systems. Alternative choices may affect the final result. Finally, from [6], we know that data items traversing all edges are structured, and we omit it for simplicity in Fig. 4.

Starting from the scenario description in Fig. 4, our methodology extracts three data flows DF^1, DF^2, DF^3 for consumers C_1, C_2, and C_3, respectively, and assigns them the components shown in Fig. 5. Each data flow includes a component for node n_1 and one for node n_2, which are required by all three consumers. Node n_1 is associated with a data-centric batch processing component in all data flows. Indeed, it receives data at a high frequency, but it produces output data periodically, every few minutes. Node n_2 is associated to a data-centric stream processing component in DF^1 and to a data-centric batch processing component in DF^2 and DF^3, due to the heterogeneous rates of requests coming from the consumers. Finally, n_3, n_4 and n_5 are associated to data-centric components. For each of these components, the rate of output requests is equal to the rate of input data. Thus, the choice of a stream or batch processing component depends on the computational cost function associated to the specific action for each of the two technologies. Unfortunately, this information cannot be easily extrapolated from [6], so we assign identical cost functions, in which case our methodology suggests a stream processing component to minimize the latency for delivering results.

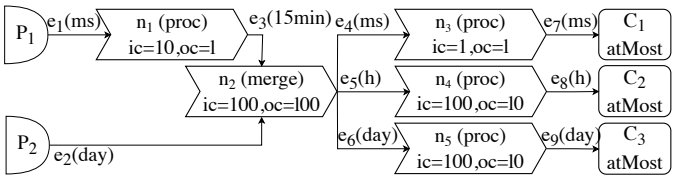


Fig. 4: Facebook use case: scenario description.

The final architecture after merging the components of the three data flows is shown in Fig. 6. Notice in particular that a stream processing component is selected for node n_2, as a result of merging batch and stream processing components from different data flows.

The architecture obtained by running our methodology is structurally identical to the one described in the original paper [6]. This confirms that our methodology guides towards architectural choices that are consistent with the ones described for the use case. In terms of technologies, the paper adopts batch processing technologies for data-centric components due to the systems available at that time (when stream processing components were not available). The actual choice of the classes of systems to adopt for each component depends on the cost functions the efficiency of each class in implementing a given action, which varies as new technologies become available.

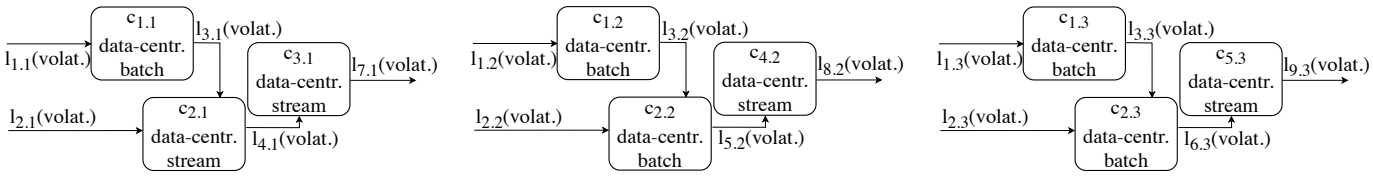
C. Efficiency

To evaluate the execution time of our methodology, we rely on synthetic scenarios of increasing complexity, and we run the entire methodology. Our methodology is written in Python. To solve the linear optimization problem, we rely on the PuLP library version 2.8[1]. We execute all the experiments on a M3 MacBook Pro with 24GB of RAM running MacOS 15.0.1, and using Python 3.11.9. We run each experiment three times, and we report the average value.

In generating synthetic scenarios, we consider two cases: (i) We generate a single data flow and increase the number of nodes in that data flow; (ii) We consider an increasing number of data flows, with a fixed number of nodes. Intuitively, the first case evaluates the complexity of the optimization problem, whereas the second case evaluates the complexity of the data flow integration problem. We report the processing times we measured for the two cases in Fig. 7a and Fig. 7b, respectively.

The results show that our methodology can provide results within seconds of execution even for (unrealistically) demanding scenarios of hundreds of nodes or hundreds of consumers. This demonstrates that the methodology is suitable as an

[1]https://coin-or.github.io/pulp/



(a) DF^1 (b) DF^2 (c) DF^3

Fig. 5: Facebook use case: components for each data flows.

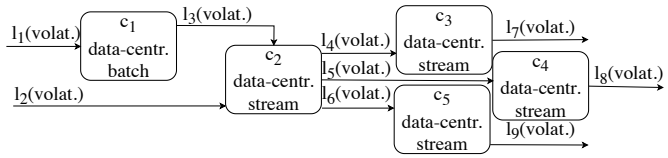


Fig. 6: Facebook use case: architecture description.

interactive decision support tool, where software architects may even step-wise refine their assumptions on the scenario to validate the impact on the suggested architecture.

D. Discussion and threats to validity

Discussion. The results we measured in the previous sections lead us to conclude that our methodology is indeed capable of capturing the key requirements of data-intensive scenarios and the recurring patterns of data-intensive architectures. Moreover, it suggests appropriate architectures for the scenarios at hand. The execution times we measured testify that the methodology can be used as an interactive decision support tool even in the presence of complex scenarios.

Internal threats. In the effectiveness validation, we extrapolated some parameters of the Facebook architecture, but we did so in a manner that appears realistic given the studied scenario. Furthermore, the difference between the data volumes and rates at play is such that we are confident the methodology would not vary for minimal parameter changes. The selection of actual systems depends on the cost of executing the specific actions on the available technologies, which is out of the scope of our evaluation.

External threats. The case studies we selected may not be representative of the entirety of cases, but we drew them from literature surveys aiming to be comprehensive.

Construct and conclusion threats. We modeled scenarios based on our understanding of scenarios and architectures, but the terminology in the papers used is clear and allows us to be confident in the correct modeling.

V. Related Work

Our work crosses the boundaries of various research topics.

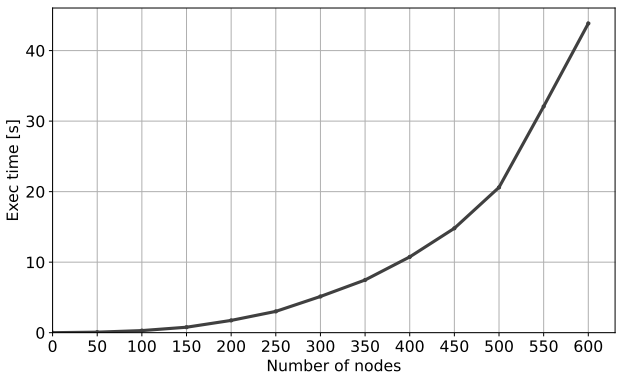
At the core of our work lies an Architecture Description Language (ADL) tailored for data-intensive applications. ADLs define the components of a software architecture and their interrelationships [8], [9]. Several architectures have been proposed to satisfy the needs of data-intensive applications, such as the lambda and kappa architectures [10]. Few studies have explored ADLs for data-intensive systems, notable examples being the UML profile in the DICE project [11] and the DAF architecture description framework [12]. Our ADL builds on a systematic analysis, modeling, and survey of data-intensive systems [1], encompassing both system [13] and software engineering [3] concerns. The resulting ADL overlaps with the models discussed above, which increases our confidence on its soundness. With respect to existing proposals, our work goes beyond the definition of an ADL and provides a methodology to automatically derive a data-intensive architecture from an application scenario.

Our Scenario Description Language (SDL) captures the requirements of the various stakeholders of a data-intensive application. As acknowledged in recent studies [3], the data-intensive domain lacks established requirement engineering practices. Yet, the literature in the area clearly points out the central role of data characteristics to capture the requirements of such applications [4], which are what our language focuses on. Related work in the fields target the specification and analysis of non-functional data requirements. The interested reader can refer to the survey in [3]. Compared to these proposals, our SDL offers greater granularity by detailing all processing stages data passes through, moving from sources to consumers.

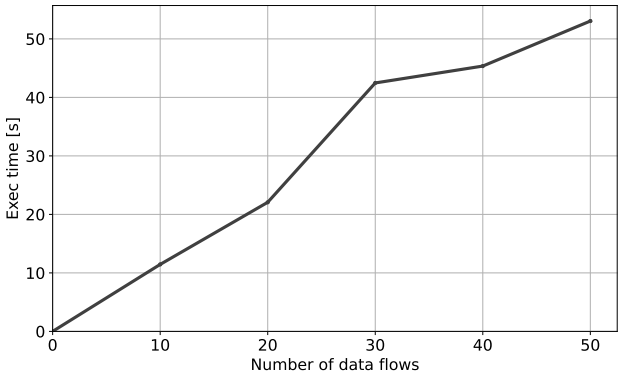
Our use of data characteristics and quality metrics to guide architectural decisions draws inspiration from architectural tactics [14]. Along this line, future work will explore additional metrics, such as fault-tolerance and data privacy, currently beyond the scope of our methodology.

Our definition of scenarios as graphs of transformations draws inspiration from the dataflow programming model [15], widely adopted in data analytics [1]. Model-driven development approaches for stream processing applications using dataflow programs have been explored in the literature [16]. Our work pursues similar goals, extending them to a broader range of applications and architectural solutions.

Finally, our work maps architectural components onto concrete systems. In doing so, it relies on the vast literature on data-intensive systems [13] and in particular on the model and survey in [1]. An interesting area of investigation for future work is the automated generation of configuration and deployment specifications of software systems, also known as infrastructure-as-code [11], [17]. Integrating this with our methodology would streamline the lifecycle of data-intensive



(a) Single data flow



(b) Multiple data flows (300 nodes)

Fig. 7: Execution time of the methodology.

applications, enabling semi-automated deployment from high-level requirements.

VI. Conclusions

This paper introduced a methodology to support software architects in designing complex data-intensive architectures. The methodology starts from a description of the application scenario at hand in terms of data characteristics and requirements of stakeholders. It produces an architecture consisting of abstract components, and it further suggests concrete software systems that may implement each of these components.

As software systems are increasingly built as compositions of vertically-specialized services, architectural decisions become even more important to realize solutions that efficiently fulfill the requirements of stakeholders. In this context, our methodology offers a systematic way to document and evaluate architectural design decisions. Confident in the potential of our work, we plan to extend it along several directions. These include broadening the methodology's scope from data-intensive systems but also operational systems, along with exploring the interplay between these two domains. Additionally, we aim to develop tools to analyze the impact of architectural changes during software maintenance. Lastly, we intend to create a detailed catalog of architectural tactics, providing more targeted guidance for the selection and configuration of systems.

Acknowledgements

We acknowledge financial support from the PNRR MUR project PE0000021-E63C22002160007-NEST.

References

[1] A. Margara, G. Cugola, N. Felicioni, and S. Cilloni, "A model and survey of distributed data-intensive systems," *ACM Comp. Sur.*, vol. 56, no. 1, pp. 1–69, 2023.

[2] M. Stonebraker and U. Cetintemel, ""one size fits all": An idea whose time has come and gone," in *Int. Conf. on Data Engineering*, ser. ICDE '05. IEEE, 2005, p. 2–11.

[3] A. Davoudian and M. Liu, "Big data systems: A software engineering perspective," *ACM Comp. Sur.*, vol. 53, no. 5, pp. 1–39, 2020.

[4] N. Ford, M. Richards, P. Sadalage, and Z. Dehghani, *Software Architecture: The Hard Parts*. O'Reilly, 2021.

[5] P. Pääkkönen and D. Pakkala, "Reference architecture and classification of technologies, products and services for big data systems," *Big data research*, vol. 2, no. 4, pp. 166–186, 2015.

[6] A. Thusoo, Z. Shao, S. Anthony, D. Borthakur, N. Jain, J. Sen Sarma, R. Murthy, and H. Liu, "Data warehousing and analytics infrastructure at facebook," in *Int. Conf. on Management of Data*, ser. SIGMOD '10. ACM, 2010, pp. 1013–1020.

[7] F. Nargesian, E. Zhu, R. J. Miller, K. Q. Pu, and P. C. Arocena, "Data lake management: challenges and opportunities," *Proc. VLDB Endow.*, vol. 12, no. 12, p. 1986–1989, 2019.

[8] N. Medvidovic and R. N. Taylor, "A classification and comparison framework for software architecture description languages," *Tran. Soft. Eng.*, vol. 26, no. 1, p. 70–93, 2000.

[9] I. Malavolta, P. Lago, H. Muccini, P. Pelliccione, and A. Tang, "What industry needs from architectural languages: A survey," *IEEE Trans. Softw. Eng.*, vol. 39, no. 6, pp. 869–891, 2013.

[10] J. Lin, "The lambda and the kappa," *IEEE Internet Computing*, vol. 21, no. 5, p. 60–66, 2017.

[11] M. Artac, T. Borovšak, E. Di Nitto, M. Guerriero, D. Perez-Palacin, and D. A. Tamburri, "Infrastructure-as-code for data-intensive architectures: A model-driven development approach," in *Int. Conf. on Software Architecture*, ser. ICSA '18, 2018, pp. 156–165.

[12] M. Abughazala, H. Muccini, and M. Sharaf, "Architecture description framework for data-intensive applications," in *Int Conf on Intelligent Data Science Technologies and Applications*, ser. IDSTA '23, 2023, pp. 99–106.

[13] M. Kleppmann, *Designing Data-Intensive Applications*. O'Reilly, 2017.

[14] G. Márquez, H. Astudillo, and R. Kazman, "Architectural tactics in software architecture: A systematic mapping study," *Journal of Syst. and Softw.*, vol. 197, no. C, 2023.

[15] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle, "The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing," *Proc. VLDB Endow.*, vol. 8, no. 12, p. 1792–1803, 2015.

[16] M. Guerriero, D. Tamburri, and E. Di Nitto, "Streamgen: Model-driven development of distributed streaming applications," *Tran. Soft. Eng. and Methodology*, vol. 30, pp. 1–30, 01 2021.

[17] K. Morris, *Infrastructure as Code*. O'Reilly, 2020.