IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 0

Multitenancy in Single Instance of Data Persistence Service for Supporting Low-Code Platforms

JÚLIO GUSTAVO COSTA[1], LUIZ M. G. GONÇALVES[2], and Samuel Xavier-de-Souza.[3]
[1]Programa de Pós-Graduação em Engenharia Elétrica e Computação/UFRN, Natal, RN, Brasil (e-mail: julio.costa.026@ufrn.edu.br)
[2]Departamento de Computação e Automação/UFRN, Natal, RN, Brasil (e-mail: lmarcos@dca.ufrn.br)
[3]Departamento de Computação e Automação/UFRN, Natal, RN, Brasil (e-mail: samuel@dca.ufrn.br)

Corresponding author: First A. Author (e-mail: julio.costa.026@ufrn.edu.br).

This work was supported by Coordenação de Aperfeiçoamento de Pessoal de Nível Superior—Brazil (CAPES)—Finance Code 001.

ABSTRACT Corporate digitalization, especially among Small and Medium Enterprises, has led to a significant increase in the demand for scarce professionals with expertise in the IT area, especially in Web Information Systems area. On the other hand, concerns about high levels of resource idleness in the cloud are constant. In this scenario, Low-Code Platforms have gained traction in the software industry, whose most commonly found service is the automation of code generation to perform data persistence tasks driven by data models. This approach, however, implies a mode of operation that is not the most mature among cloud computing providers, implying for each model a service instance running on the provider side. Here, we propose a data persistence service for these platforms that avoids code generation, since it interprets data models at runtime and operates in multi-tenant mode with a single service instance. This approach improves resource sharing, mitigating resource idleness within the platform. In addition, we present experiments to support the technical feasibility of the proposed approach. The proposed solution offers an alternative to code generation methods, with the potential to optimize resource utilization while preserving the flexibility to adapt to changes in data models as business needs evolve.

INDEX TERMS Model Driven Engineering; Low Code Development; Model Interpretation; Multi-tenancy Single Instance; Separation of Concerns

I. INTRODUCTION

The substantial increase in demand for the digitalization of business processes that have occurred in the last decade has brought with it a set of significant challenges and opportunities. The growing need for software systems, combined with the difficulty of training qualified professionals at the speed needed to meet demand, has reinforced the search for software development and operation approaches capable of performing this task in a more agile and simplified way, especially those of Small and Medium-sized Business (SMB). In this context, the so-called Low-Code Platforms (LCP) have gained traction [24], [28].

Recent studies [18], [23] highlight how LCPs can help mitigate the mismatch between supply and demand, either by being able to alleviate the dependence on scarce and expensive professionals with in-depth training or by being able to respond more quickly to demands for new applications, even if values such as quality are relativized in favor of results [9], especially in the context of SMB.

LCPs are designed to simplify the software development process by quickly enabling professionals without *in-depth* technical knowledge in software development to create software applications [17], [18] or to offer significant productivity gains for more experienced professionals. These platforms, on the one hand, typically provide an intuitive visual interface for these professionals to abstract away various technical aspects at the implementation level and focus more objectively on abstractions that more directly touch the business aspects of the software system [13], [17], [26] and, on the other hand, they also hide a series of infrastructure and database concerns related to the solution deployment process and its operation. These facilities, among others, are commonly obtained through the use of modeling mechanisms, generally originating from the Model-Driven Engineering (MDE) knowledge domain, capable of allowing the capture of system abstractions at their highest levels, close to the business abstraction levels [4], [23].

From this, underlying the working environment of users of LCPs, the software systems that emerge from their use follow a three-tier architecture: presentation (frontend), business logic (backend), and database (infrastructure) [19], especially in the context of the web Information Systems (SI). The presentation tier exhibits the least potential for resource sharing. Such presentation bundles are distributed and executed, typically not sharing the same host or compiled code instance. However, the remaining tiers offer more flexibility and can be structured to maximize the utilization of shared resources—code and host—on the provider side. This is convenient because cloud providers face challenges due to high idle resources or wasted computing capacity. Research by [1] and [29] highlight these concerns. To address this, software systems are designed with transparent mechanisms to distribute workloads dynamically across the provider's infrastructure. This ensures more efficient resource utilization.

Currently, approaches that support the operation of business logic layer (or backend layer) services on LCP platforms employ automated code generation directly from models. These approaches generate, for each model delivered by each client, a code instance that, when compiled and executed, becomes a service instance that meets the specific demand of that client (its model). In these terms, for each model, there is a compiled service instance dedicated to serving only the requests of the frontend layer of the model of the application—a situation in which reuse occurs only at the level of knowledge exposed in the models and not at the code or host level. In other words, the providers' operational mode is a Multi-Tenant Multi-Instance (MTMI) service model, with one instance for each client model.

In this regard, our research identified a opportunity regarding the design and operation of data persistence services in the context of LCP platforms. Despite our best research efforts, we found no studies exploring the Multi-Tenant Single-Instance (MTSI) operating mode at the backend level in such platforms. This is relevant because it is the most attractive cloud computing service mode for service providers with a higher degree of maturity in the service implementation—maximizing resource sharing and minimizing idleness. Furthermore, it is worth highlighting that data persistence service is one of the most present in the context of LCPs [17], [26].

We propose, discuss, implement, and present initial experiments with an MTSI data persistence service for LCP platforms. To achieve this, we choose the Runtime Model Interpretation [3], [6] approach from the MDE approaches. This approach maximizes decoupling between client domain models and the LCP platform's persistence service implementation. Our work offers at least two contributions. First, we discuss the design and implementation of the proposed service for LCPs, addressing the growing demand from LCP providers. Second, we present initial experiments that help to understand the feasibility of the approach from the perspective of evaluating the processing load dynamics and memory, introduced by the dynamic loading and runtime manipulation of data models.

The rest of the text is organized so that Section II presents a brief discussion of the relations between LCP and MDE, in which we highlight the design and how the design of our proposition fits into the relation. Section III, in turn, presents the implementation following this design. We classify and evaluate experimental data in Section IV. In the penultimate Section V, we put this work into perspective, discussing related works. Finally, in the conclusions, we offer a brief discussion about the limits and challenges of our proposition.

II. THE PROPOSED MULTI-TENANT PERSISTENCE SERVICE

Low-code platforms strongly rely on MDE approaches within their underlying architecture. This reliance is critical to achieving the productivity gains that LCPs promise. MDE methodologies involve system modeling tools, usually graphical, that allow *automated code generation* directly from user-created models—the cornerstone of LCP benefits.

However, the path we have chosen is different. Given the objective of maximizing the use of shared resources within the context of the service provider infrastructure, to achieve the proposed objective, we consider among the Model-Driven approaches the one capable of offering the most significant potential to decouple the implementation of the persistence service from the different business data domain models of different LCP clients, namely: *runtime model interpretation* approach.

In this approach, the business data domain is made known to the service only at runtime, loading and interpreting data models only at the time of the client application request. Instead of precompiling code for each data model, this service in an LCP retrieves and interprets the data specifications dynamically to build the persistence command required to fulfill the request and execute it against a database.

In this sense, the following discussion is directly associated with the issues surrounding building a single multi-tenant service instance for data persistence performed in the scope of the backend layer on the LCP provider backend side. Here and in the subsequent two sessions, we present the design, implementation, and initial experimentation of a persistence service driven by the interpretation of models at runtime. This will be done assuming that the working environment of its users centrally implies the use of data modeling tools, more precisely, a subset of UML Class Diagrams symbols, through which the entire process of the service instance will be conducted.

A. DATA METAMODELS DESIGN

As a first step towards implementing a data persistence service driven by runtime model interpretation, it is necessary to define the scope of data modeling possibilities that users of the service platform will have at their disposal. Thus, from the perspective of the defined scope, here we are addressing the central lines of the interpretation process for the persistence service, abstracting all other aspects of the scenario.

In this context, users of this service can abstract away the complexities related to the implementation, infrastructure, and deployment underlying their data models, whether at the backend software system level or at the database level, focusing only on the business aspects that can improve the understanding and definition of the data model. From there, we can focus on the mechanisms for interpreting models, in the sense that these mechanisms require a metamodel definition capable of helping them recognize the concepts and operations they can perform on the data [25]. Thus, all models must be structured and standardized according to the metamodel. Second, this must be done so that the model can evolve as the data in the business domain evolves.

The metamodel presented in Fig. 1 is a subset of the UML Class Diagrams metamodel. It subtracts the signature of class methods from the class representation. We chose this level of abstraction to allow new representation possibilities to be explored as the research progresses, in addition to being associated with the demands of the business logic/backend layer where the interpretation will occur. As an illustration of the expressiveness of this metamodel, Fig. 2 synthetically presents what can be considered an example of a data model.

In turn, the illustration of a specification that defines which account role can access or persist data can be seen in the partial data model in Fig. 3, according to the metamodel in Fig. 1 and the data model in Fig. 2. From the point of view of the operation of the interpretation engine, semantically, Fig. 3 should be understood as follows: Privilege (`GrantPolicy` instance) establishes a relationship between Maria (`Account` instance) and Waiter (`Role` instance) so that this relationship informs that Maria is a waitress. This relationship authorizes the system to accept Maria to perform reading operations on instances of Orders in the database.

Furthermore, the metamodel in Fig. 1 offers three opportunities for its users to indicate the use of relevant features: to perform concurrency control tasks when accessing data; the realization of data classes in a specific class of database (SQL, NoSQL classes, for example); and hooks for invoke functions defined and coded by users. Respectively, in Fig. 2, «`Concurrency`» denotes the arbitrary choice of one among many possibilities for implemented concurrency control algorithms when accessing data related to the `Order` class; «`SQL`», in turn, inform the need to use a specific type of database when loading classes at the database level; and the stereotypes «`BHook`» (class `Billing`) and «`AHook`» (classes `Card`, `OrderItem`) indicates to the interpreters the need to invoke functions immediately before and/or after the execution of the commands to persist changes on data linked to the object. Moreover, the symbols , !, and , respectively, mean that the value space of an attribute must contain only unique values, that an attribute must always have some value associated with it, and finally, that values associated with an attribute should not be persisted.

Finally, the expressiveness of models according to the presented metamodel is not limited to it. More expressive metamodels can be built to offer a broader range of functionalities to leverage the data persistence service. The metamodel offered in Fig. 1 is presented only as a starting point to understand the motivations for using runtime model interpretation mechanisms discussed in this work.

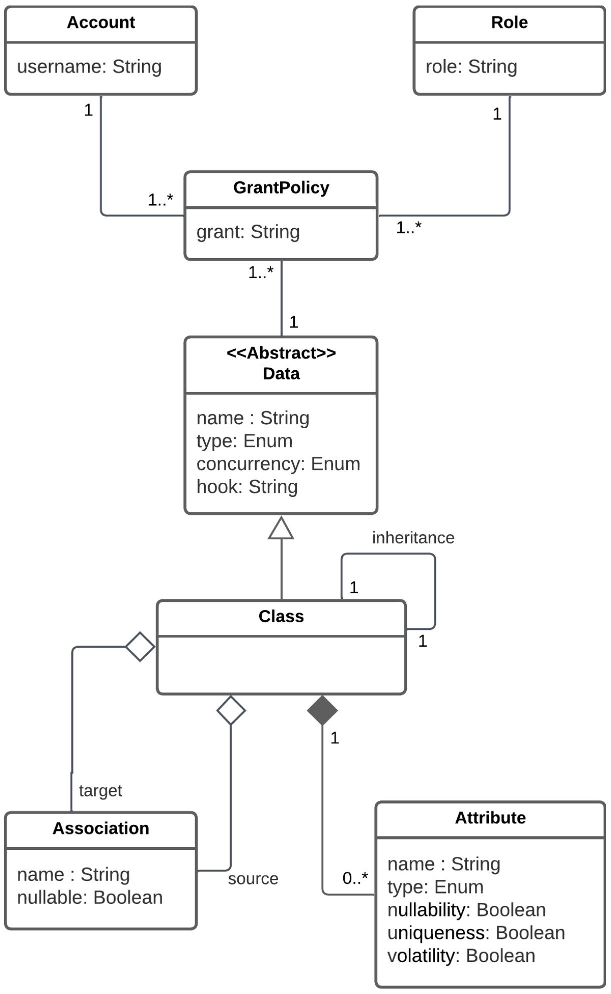

FIGURE 1. The proposed persistence data service metamodel.

B. MULTI-TENANT SINGLE SERVICE INSTANCE ARCHITECTURE DESIGN

Once the metamodels have been defined, reflecting on how the interpretation engines should assimilate the models, a unique service instance guides their execution in a multi-tenant manner. In that regard, Fig. 4 presents the architecture's first two layers of services: the Composable Modeling Service and the Data Definition Language (DDL) Statements Transformation Service.

The Composable Modeling Service is the first layer of the service and directly handles client requests made from data models. In this first layer, the Model Manager (MM) component is the artifact that receives requests directly from clients and dispatches them to other components. Furthermore, in the same layer, the Model Parser (MP) is the artifact responsible for analyzing the data model sent to the MM regarding its

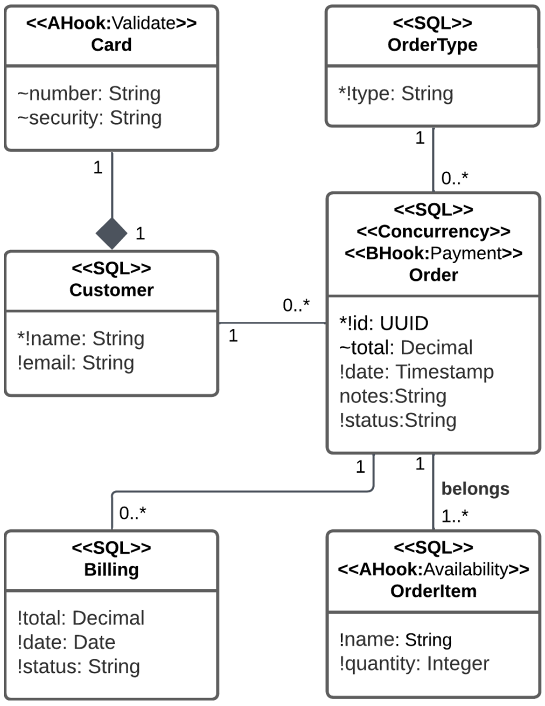

FIGURE 2. Data hierarchy model for a pizzeria service.

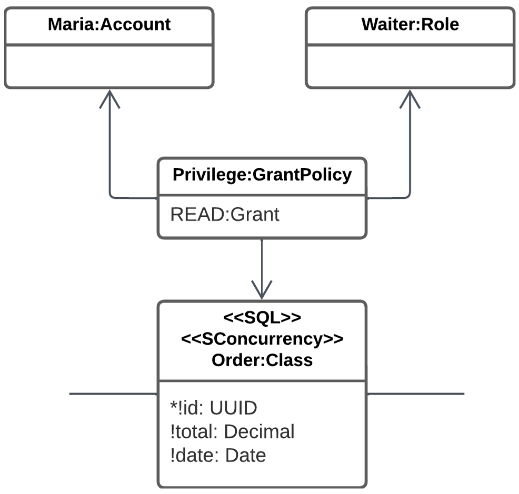

FIGURE 3. A partial data hierarchy model improved with the role-based access control specification.

adequacy to the metamodel in Fig. 1.

Another component present in this first layer is the `Function Deployer` (FD), responsible for receiving the functions that will be invoked immediately before or after the execution of the persistence commands. Through it, behaviors related to business rules and validations, for example, can be integrated into the execution of the model interpretation. This component constitutes an execution environment (virtual machines, containers, or similar) capable of delivering the appropriate computing environment for the function in terms of the chosen programming language and libraries—preferably through dynamic binds.

The Data Definition Language (DDL) Instruction Transformation Service (DDL-IT) is a second layer of the service. After passing the necessary model validations, one must perform the transformations required to transpose the representations in class diagrams to database representations (non-relational or relational) with their constraints indicated. This is done by the SQL Representation Transformer (SQL-RT) or NoSQL Representation Transformer (NoSQL-RT) components, both performing their transformations according to the metamodel in Fig. 1.

Fig. 5, in turn, presents two other groups of components. The Composable Model Interpretation Engine (CMIE) is the first layer of the persistence service and is immediately associated with client requests. The second layer is the Data Manipulation Language (DML) Instruction Transformation Service (DML-IT), which handles representation transformations. The first layer combines the Data Repository (DR), Data Handle Hook (DHH) and the Model Cache Manager (MCM) components.

The DR component receives and forwards commands that involve modifying the state of the data, or queries, directly to the representation transformers. However, based on the analysis of the metadata (of the model) to which the request is linked, it may invoke functions to be executed by the DHH immediately before sending these commands or queries to the transformers. It may also invoke functions in the DHH immediately after the commands are executed by the transformers. The MCM component, in turn, keeps the metadata of the most recently used models in memory in order to avoid constantly loading models from less agile reading devices.

In Fig. 5, there are also the NoSQL-RT and SQL-RT components. They receive commands and queries from DR and convert them to the appropriate DML format at the database level, according to the modeling delivered to the MM component.

It is also in this group of components, more specifically in the transformers, that we locate the access control evaluation logic. We chose these components because they directly handle the manipulation of data classes. This logic evaluates the authorization of the transformation engine to execute the request made to DR based on its type (read/write), identity of the requester, and class of data in the request. Such operations are performed as specified in the metamodel in Fig. 1 and in the pseudocode in Fig. 11, the latter seen later.

A third group of components, the Database Driver Service (DDS), is elaborated in Fig. 6, consisting of the NoSQL Database Driver (NoSQL-DD) and the SQL Database Driver (SQL-DD) components. These are tasked with executing specific instructions required by the NoSQL-RT or SQL-RT components, whether those components operate from the perspective of DDL-IT layer or DML-IT layer concerns. These instructions are the result of processing carried out by MM (on metadata deployment) or processing carried out by DR (on data access).

To maximize the applicability of the data persistence service proposed, the functionalities chosen to compose it are

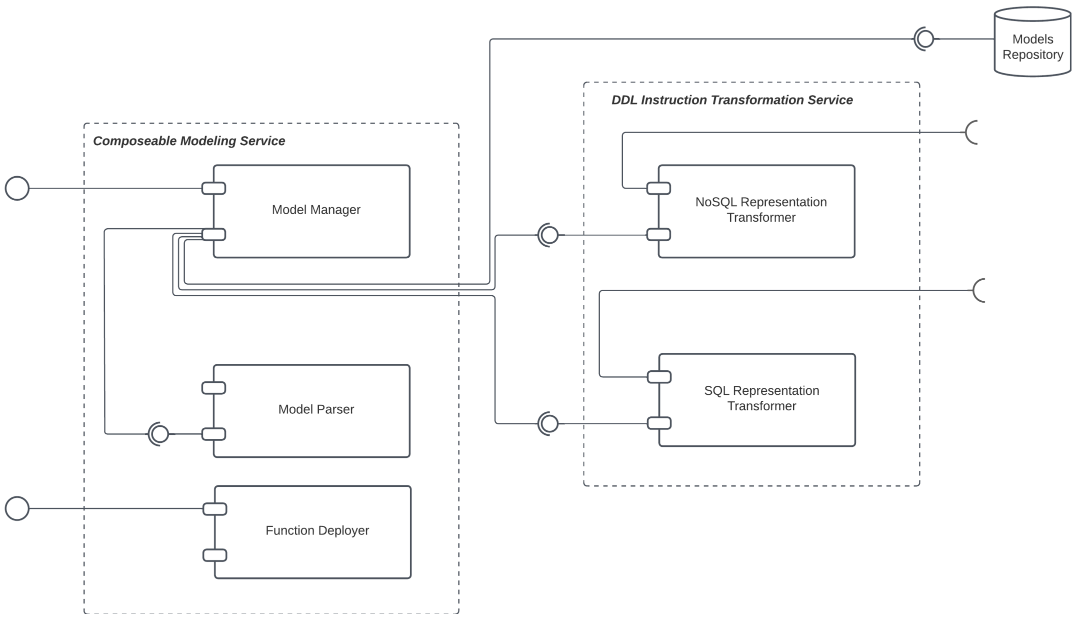
Models Repository
DDL Instruction Transformation Service
Composeable Modeling Service
Model Manager
NoSQL Representation Transformer
Model Parser
SQL Representation Transformer
Function Deployer

FIGURE 4. The synthetic proposed data persistence service design.

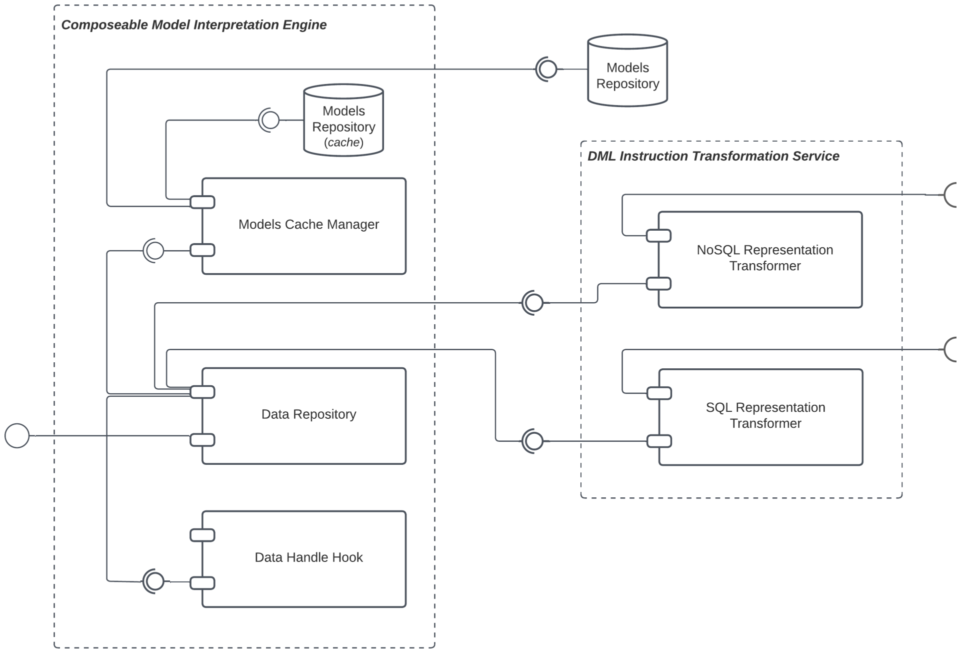
Composeable Model Interpretation Engine
Models Repository
Models Repository (cache)
DML Instruction Transformation Service
Models Cache Manager
NoSQL Representation Transformer
Data Repository
SQL Representation Transformer
Data Handle Hook

FIGURE 5. A synthetic data persistence service architecture for model interpretation engine.

transversal to any business domain of a given software application. In terms of implementation, given these concerns, each component should be viewed as a microservice—our strategy to ensure more effective separation of concerns and improve service scalability [15]. In the following section, we will delve into the implementation aspects of this approach.

III. IMPLEMENTATION

Our platform's first group of components, as depicted in Fig. 4, is responsible for receiving the models and deploying them according to the platform's resources. The second group, as shown in Fig.5, provides a set of computational services necessary to deliver the data access and persistence service compatible with the limits imposed by the meta-model. The third group sends the instructions generated by the transformers to be executed in the databases. The instances in this group are responsible for accessing and maintaining access to the databases. In this section, we only address the relevant points necessary for operating the model interpretation engines from an architectural perspective, aligning

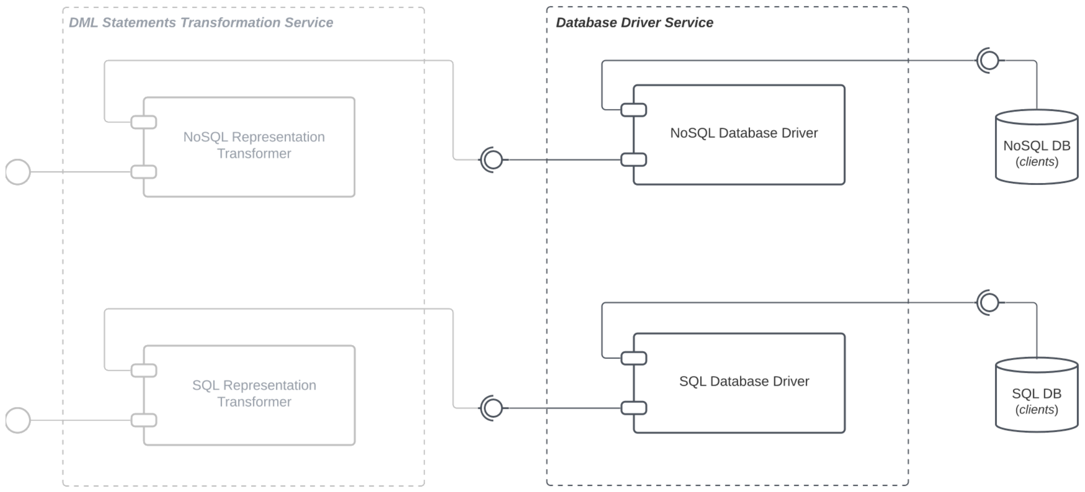
DML Statements Transformation Service
NoSQL Representation Transformer
SQL Representation Transformer
Database Driver Service
NoSQL Database Driver
SQL Database Driver
NoSQL DB (clients)
SQL DB (clients)

FIGURE 6. The database drivers service

the highlighted components with the metamodel described previously.

A. MODEL DEPLOYMENT

Our models are built on existing external data modeling platforms that use standardized file formats, such as XML Metadata Interchange (XMI), for UML Class Diagrams (CD). These files are delivered to the MM through the modeling service interface (see Fig. 4). Then, they must go through parsers, transformers, and drivers for deployment in the database.

Inside the platform, to facilitate the transfer of model metadata and their manipulation in memory, we use JSON technology as a data wrapper for exchanging data between components because it provides efficiency compared to XML [7], [20].

Upon receiving the XMI file that represents the data model, the MM will invoke the MP for a dual function. First, it evaluates the compliance of the delivered models with our metamodel (as shown in Fig. 1). Second, it must recognize marks on the model that indicate possible changes in the representation to allow corrections and/or evolutions of the business-domain data. If these marks are present in the data model and are recognized, the MM forwards them to either one of the transformation engines: SQL-RT or NoSQL-RT (see Fig. 7).

When the MP identifies such marks present in the data model, it will indicate to the MM the need for a change in the current state of the database representation. This component will then request the appropriate representation transformation component—SQL-RT or NoSQL-RT—to build and execute the statements suitable for the underlying databases authorized by the platform so that the new data representation can be deployed into the database. For example, a newly added data class (see Fig. 7, representation of `OrderItem`) in the model must be prefixed with the following two characters: `c:` for it to be deployed in the database. After editing the diagram in the modeling tool of the user's preference, it must be submitted in XMI format to the MM's external interface. The MM, in turn, will invoke the MP to perform validations in regard to the metamodel presented in Fig. 1. The MP, upon recognizing the specification in XMI format, must identify the presence of the `c:` characters prefixed to the name of the new `OrderItem` class. Upon recognizing these characters, the MP will return to the MM the set of metadata necessary to create the database statement that will create the table in a database, in this case, SQL (see the <<SQL>> stereotype in Fig. 7). This metadata will also include information related to constraints defined in the modeling itself (such as uniqueness, *nullability*, and volatility). Finally, once these statements are generated, they will be executed against the database either by the SQL-DD engine or the NoSQL-DD engine.

Moreover, to correctly instruct the transformation engines to *delete* data classes, the class names must be prefixed with `d:`—the same applies to attributes and associations. If any properties of data classes, attributes, or associations appear to the parser prefixed with the characters `u:`, operations to emphasize or modify the current representations will be performed. The symbol `!` should be used to denote that an attribute cannot exist without an associated value, as shown in Fig. 7. Similarly, $\star$ should be understood as the property of non-duplication of values in the attribute's value space. The $\sim$ character indicates that values associated with attributes prefixed by it should not persist in the database.

We highlight that the transformations discussed in the previous paragraphs can be carried out with the support of representation transformers such as the Eclipse Modeling Framework [1] (EMF), for example, or through a custom implementation of interest to the platform's development.

Furthermore, after a new model is deployed on the platform, they receive a unique identification key, arbitrarily called '`tenantId`', in order to distinguish them from each other, especially when interpreters need to use them.

[1]https://eclipse.dev/modeling/emf/

Therefore, to invoke different models (and their underlying databases) from a software application and integrate them, the `tenantIds` of each model need to be known by the user implementing frontends or functions executed through hooks (see Fig. 5 and the subsession III-C).

Finally, the FD component (see Fig. 4) succinctly represents the platform's ability to deploy containers within its context. Such containers are virtual machines (TypeScript or Python, for example), an appropriate set of libraries, and the client functions to be invoked by the interpreters at the appropriate time. We standardize the invocation of these functions through an HTTP/REST interface. The deployment occurs automatically through container managers like Docker, for example.

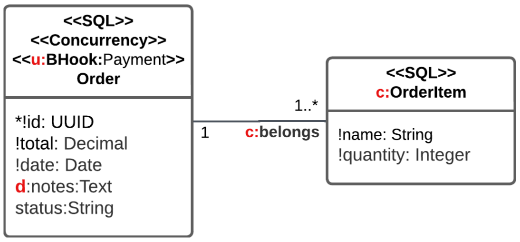

FIGURE 7. A Trace of a UML Class Diagram—Flagging commands to the Model Manager Component

```
{
  "model": {
    "data": {
      "classes": {
        "order": {
          "type": "sql",
          "roles": {
            "read": ["Waiter"]
            "write": ["Customer"]
          },
          "composedKey": ["customerid", "number"],
          "invokeBeforeHook":"payment",
          "attributes": {
            "number": {
              "type": "Long",
              "nullable": "false"
            },
            "date": {
              "type": "Datetime",
              "nullable": "false"
            },
            "total": {
              "type": "Money",
              "nullable": "true"
            }
          },
          "associations": {
            "belongs": {
              "type": "Set<OrdemItem>"
            }
          }
        },
        // Other classes...
      }
    }
  }
}
```

FIGURE 8. A shortcode in JSON description about the internal model representation.

There are two types of transformations during the model deployment. The first type occurs after using the MP, aiming to create and store a lighter representation of the model that better meets the purposes of manipulation by the components in the persistence service. This lighter representation is obtained by transforming an XMI file format (sent to the platform during model deployment) into a JSON format and store indexing it by its key `tenantId`—see Fig. 8.

After that transformation, the second type of transformation generates data-related statements of the underlying database in DDL format. These statements are primarily of four kinds: create, update, delete, and those related to constraint data manipulation.

At this point, we believe it is relevant to highlight the flexibility that comes together due to using the model interpretation approach. The flexibility we are talking about is related to the fact that it is possible to implement new data models or carry out updates without requiring the service to stop, in addition to the possibility of the platform remaining operational for all user data models despite the need to update a specific model.

B. MODEL CACHING

Storage and retrieval of models are significant implementation concerns. Here, the approach does not involve dealing with the challenges of large models (hundreds of megabytes of descriptive metadata) and even less with the systematic use of different types of diagrams and modeling languages [14], [15], [22]. The use of a Model Caching service is treated as relevant here in the potential sense that such a service platform would be needed to handle multiple data models from various clients.

In very simplified terms, the MCM component (see Fig. 10), in cases where the cache has reached its storage limit (arbitrarily set to 1000 stored models), one model will be removed from the cache so that the requested model can be stored into the cache. The criteria for choosing which models will be removed is the Least Recently Used (LRU).

C. DATA REPOSITORY

The DR component receives requests involving persistent commands and queries against the underlying database. Each of these requests carries the tuple `<authToken, tenantId, clazzName, data, command>`: `command` is either Create, Update, or Delete; `data` is the data used by the commands; `clazzName` is the class name to the data; `authToken` is a token that contains encoded information about the user's credentials; and `tenantId` is the key chosen as the data model indexer.

Upon receiving a request, the DR invokes the MCM to evaluate the cache. If the data model is present in its memory, it updates the model retention timestamp based on the current time (see Fig. 10) and sends the data model as a response to the DR. Then, the DR evaluates whether a hook needs to be invoked before the execution of persistence commands (see line 13 of Fig. 11). As previously defined (see Section II-A)), pseudocode for the hook can be found in Fig.12. Afterward, the DR invokes the appropriate transformation component, SQL-RT or NoSQL-RT, according to the class specification

```
class ModelManagerService {
  ModelParser parser;
  SqlRepresentationTransformer sqlTransformer;
  SqlDatabaseDriver sqlDbDriver;
  NoSqlRepresentationTransformer noSqlTransformer;
  NoSqlDatabaseDriver noSqlDbDriver;
  /**
   * @param XMI file, a Class Diagram
   * @param ’tenantId’ from an existing model
   * @return the ’tenantId’ of the deployed model.
   */
  String deployModel(XMI xmi, String tenantId) {
    DataModel dataModel;
    if (tenantId == null) {
      // UUID.randomGen() generates a hash, the
      // ’tenantId’ value.
      dataModel = new DataModel(UUID.randomGen())
        .parse(xmi).build();
    } else dataModel = new DataModel(tenantId)
      .parse(xmi).build();
    dataModel.getClasses().forEach(clazz -> {
      // to extract from the model the ’Data
      // Definition’ command
      if (clazz.getType().equals("SQL")) {
        DdlSqlStatement ddlStatement
          = sqlTransformer.extractDdlCmd(clazz);
        sqlDbDriver.ddl().execute(ddlStatement);
      } else if (clazz.getType().equals("NoSQL")) {
        DdlNoSqlStatement ddlStatement
          = noSqlTransformer.extractDdlCmd(clazz);
        sqlDbDriver.ddl().execute(ddlStatement);
      } else throw new Exception("unknown database");
    });
    return dataModel.persist();
  }
  JSON getDataModel(String tenantId) {
    return new Model(tenantId).getJsonRep();
  }
}
```

FIGURE 9. A short Java-like description about the implementation of the Model Manager Component.

```
class ModelCacheManagerService {
  Map<String, Map<String, Object>> cache;
  void put(String tenantId, DataModel dataModel) {
    if (cache.length() > 1000) {
      removeLRU();
    }
    Map<String, Object> entry;
    entry.put("usedOn", Calendar.now());
    entry.put("model", dataModel);
    cache.put(tenantId, entry);
  }
  DataModel getDataModel(String tenantId) {
    if (cache.contains(tenantId)) {
      cache.get(tenantId).put("usedOn", Calendar.now());
      return cache.get(tenantId).get("model");
    } else throw new Exception("model not found");
  }
  void removeLRU() {
    Map<Long, String> lru;
    cache.keys().forEach(key -> {
      lru.put(cache.get(key).get("usedOn"), key);
    });
    cache.remove(lru.get(order(lru.keys()).getLast()));
  }
}
```

FIGURE 10. A short in Java description about the implementation of Model Cache Manager Component.

```
01. class DataRepositoryService {
02.   ModelCacheManagerService mcmService;
03.   HookHandler hookHandler;
03.   /**
04.    * @param: the tuple <authToken,
05.    *   tenantId, clazzName, data, command>
06.    */
07.   void command(String authToken,
08.     String tenantId, String clazzName,
09.     JSON data, String command) {
10.     JSON clazzes = mcmService
11.       .getDataModel(tenantId);
12.     JSON clazz = clazzes.getJSON(clazzName);
13.     if (clazz.has("BHook")) {
14.       data = hookHandler.invoke(clazz.get("BHook"),
15.         authToken, tenantId, data, command, true);
16.     }
17.     if (clazz.get("Type").equals("SQL")) {
18.       DmlSQLStatement dmlStatement
19.         = sqlTransformer.set(authToken,
20.             clazzes, clazz, data, command)
21.           .isAuthorized()
22.           .extractDmlCmd();
23.       data = sqlDbDriver.dml().execute(dmlStatement);
24.     } else if (clazz.get("Type").equals("NoSQL")) {
25.       DmlNoSQLStatement dmlStatement
26.         = noSqlTransformer.set(authToken,
27.             clazzes, clazz, data, command)
28.           .isAuthorized()
29.           .extractDmlCmd();
30.       data = noSqlDbDriver.dml().execute(dmlStatement);
31.     }
32.     if (clazz.has("AHook")) {
33.       hookHandler.invoke(clazz.get("AHook"),
34.         authToken, tenantId, data, command, false);
35.     }
36.   }
37.   /**
38.    * @param the tuple <authToken, tenantId,
39.    *  clazzName, filter>;
40.    * @return the list of objects from a table,
41.    *  or collection;
42.    */
43.   JSON query(String authToken, String tenantId,
44.     String clazzName, JSON filter) {
45.     JSON clazzes
46.       = mcmService.getDataModel(tenantId);
47.     JSON clazz = clazzes.getJSON(clazzName);
48.     if (clazz.getType().equals("SQL")) {
49.       DmlSQLStatement dmlStatement
50.         = sqlTransformer.set(authToken,
51.             clazzes, clazz, filter)
52.           .isAuthorized()
53.           .extractDmlQuery();
54.       return sqlDbDriver.dml().query(dmlStatement)
55.         .toListOfObject();
56.     } else if (clazz.getType().equals("NoSQL")) {
57.       DmlNoSQLStatement dmlStatement
58.         = noSqlTransformer.set(authToken,
59.             clazzes, clazz, filter)
60.           .isAuthorized()
61.           .extractDmlQuery();
62.       return noSqlDbDriver.dml().query(dmlStatement)
63.         .toListOfObject();
64.     }
65.   }
66. }
```

FIGURE 11. A short Java description about the implementation of the Data Repository Component.

in the data model, considering the constraints imposed in the model itself (for example, uniqueness, nullability, and volatility). These components, in turn, assess the authorization to execute the request and, if authorized, transform the request into a format accepted by the underlying DML. In the final stage, a new hook execution evaluation is conducted (see line 32 of Fig. 11), and functions may or may not be invoked. The code in Fig.13 illustrates the body of a request to the DR.

After completing the representation transformation, the resulting statements need to be sent to the appropriate database driver component—either the SQL-DD or the NoSQL-DD—as depicted in Fig. 6.

IV. EXPERIMENTS AND RESULTS

In this section, we present experiments to evaluate the technical feasibility of the proposed persistence service platform regarding two technical aspects. First, we compare the average request processing time to a conventional implementation. Second, we evaluate distinct processing and memory loads and compare them to the same conventional implementation. These experiments' results establish a minimum empirical

```
class HookHandler {
  HttpClient httpClient;
  void invoke(String functionId,
    String authToken, String tenantId, String data,
    String command, Boolean isBefore) {
    JSON body = new JSON();
    body.put("authToken", authToken);
    body.put("tenantId", tenantId);
    body.put("functionId", functionId);
    body.put("command", command);
    body.put("data", data);
    body.put("isBefore", isBefore);
    HttpRequest request = HttpRequest.newBuilder()
      .POST(body)
      .uri("http://localhost:8080")
      .build();
    HttpResponse<String> response = httpClient
      .send(request,
        HttpResponse.BodyHandlers.ofString());
    if (response.statusCode() == 200)
      return response.body();
    else throw new Exception("Hook fail");
  }
}
```

FIGURE 12. A short in Java description about the implementation for Hooks.

```
{
  "authToken": "...",
  "tenantId": "...",
  "class": {
    "name": "order"
    "instance": {
      "number": "23",
      "total": "5.56",
      "date": "2023-12-03"
    }
  }
}
```

FIGURE 13. A short JSON description about body message request to Data Repository Component.

understanding of the technical implementation and operation challenges.

To evaluate both aspects, we carried out experiments related to data persistence and retrieval tasks in databases, measuring processing time and hardware load required for such tasks. In general terms, the experiments consisted of firing variable loads of asynchronous requests at maximum speed to persist or read data.

In this sense, we compared the effect of these requests on the operation of two different persistence APIs. The first is a conventional implementation of a data persistence service using the *Spring Boot JPA* framework with Java—implemented as a microservice for data persistence in accordance with the model in Fig. 2 and running a `t2.small` instance on Amazon Web Service (AWS) infrastructure. We chose a cloud execution environment since this is the intended execution context for the platform in production conditions.

The second is the implementation of the same service according to the proposed persistence service, submitted to the load of five instances of the same data model in Fig 2. The implementation follows the previously specified design, with the DR, MCM, SQL-RT, and NoSQL-RT components in a single microservice (from now on, we call this microservice `mDR`), deployed on a second `t2.small` instance. The SQL-DD component (from now on, we call this microservice `mSQLDD`), in turn, was implemented together in a third `t2.small` instance. The choice of a distributed architecture for implementation is more natural for improving the platform's management capacity, evolution, and scalability. Therefore, implementing it in these terms helps to approximate the operational conditions of these experiments to the real application. All these components are implemented in Java and use the Spring Boot framework.

The requests made to the persistence service were always in proportional quantities for each of the models and in a shuffled way during the realization of each of the four groups of experiments—as further described. Moreover, all messages were sent asynchronously at maximum speed for either implementation. While executing requests, we evaluated the CPU and memory utilization and the time elapsed average necessary to carry out the requests. These evaluations occur per `t2.small` and microservice.

We highlight that the requests sent to the persistence service platform, referring to the operations on five models related to writing or reading data, are counted in aggregate form to measure the CPU and memory effort and evaluate the elapsed time per request. Therefore, the CPU, Memory, and Elapsed Time columns in the following tables indicate the effort required to process the aggregate of requests.

Furthermore, to perform the *Create* operations, the same data class was employed for both implementations, namely the `Order` class. The same class was chosen for the *Update* and *Delete* operations. Regarding data reading operations (*Read*), the query involved retrieving, at a time, different records from two data classes, namely, `Order` and `OrderItem`—for each `Order` record, through a database-level Join, its set of `OrderItems`. None of the read operations involved querying data that did not exist in the database. Similarly, *Update* and *Delete* operations only dealt with data that existed in the database. When executed, *Create* operations always involved creating data that respected the constraints of the `Order` table in the database.

We created four groups of experiments. The first group, Table 1, involves executing a load of 200 requests, separated into two subgroups, with 100 requests for each subgroup. The first subgroup experimented only with shuffled requests of *Create*, *Update* e *Delete* operations, which targeted the "Java/Spring Boot JPA"-based service API—see line 1 of the Table 1, the Java/JPA implementation was coded into the microservice. The second subgroup, in turn, also aimed to investigate the behavior of the persistence API under the same set of operations. See line 2 in Table 1.

It is important to highlight that, regarding the choice of persistence load and data query request, based on the authors' experience in the context of Information Systems (IS) in SMEs, in general, it is quite unusual for such systems to be subjected to dealing with demands exceeding the order of thousands per second. This was the criterion that guided our choice of load for the experiments.

From the experiment in Table 1, we observed that the average elapsed time due to the adversarial Java/JPA implementation was close to one and a half times the average time

TABLE 1. First Experiment

Models	Implementation	Requests	Operation	CPU (%) (peak)	Memory(MB) (peak)	Elapsed Time - AVG(ms)
1	Java/JPA	100	C,U,D	1.1	222	34
5	Proposed Service	100	C,U,D	1.6(mDR), 1.8(mSQLDD)	165(mDR), 427(mSQLDD)	22

of the persistence API. As for CPU usage, concerning the effort to meet the demand of the `mDR` API, its usage was slightly higher than that of the adversarial API. The same can be said regarding the CPU behavior to serve `mSQLDB`. Regarding memory usage, the total memory used to serve requests from five models was 592 MB, compared to the total memory of 222 MB used to serve a single model in the adversarial API.

The second group, Table 2, involves executing a load of 1000 requests, separated into two subgroups, with 500 requests for each subgroup. The first subgroup experimented only with shuffled requests of *Create*, *Update* e *Delete* operations, which targeted the "Java/Spring Boot JPA"-based service API—see line 1 of that table, the Java/JPA implementation was coded into the microservice. The second subgroup, in turn, also aimed to investigate the behavior of the persistence service API under the same set of operations. See line 2 in Table 2.

From the experiment in Table 2, we observed that the average elapsed time due to the persistence API was equivalent to the average time of the adversarial API. As for CPU usage, concerning the effort to meet the demand of the `mDR` API, its usage was slightly higher than that of the adversarial API. The same can be said regarding the CPU behavior to serve `mSQLDB`. Regarding memory usage, the total memory used to serve requests from five models was 484 MB, compared to the total memory of 220 MB used to serve a single model in the adversarial API.

The third group, Table 3, involves executing a load of 200 requests, separated into two subgroups, with 100 requests for each subgroup. The first subgroup experimented only requests of *Read* operations, which targeted the Java/JPA-based service API—see line 1 of that table. The Java/JPA implementation was coded into the microservice. The second subgroup, in turn, also aimed to investigate the behavior of the persistence API under the same set of operations. See line 2 in Table 3.

From the experiment in Table 3, we observed that the average elapsed time due to the persistence API was close to one and a half times the average time of the adversarial API. As for CPU usage, concerning the effort to meet the demand of the `mDR` API, its usage was slightly higher than that of the adversarial API. The same can be said regarding the CPU behavior to serve `mSQLDB`. Regarding memory usage, the total memory used to serve requests from five models was equivalent to 394 MB, compared to the total memory of 202 MB used to serve a single model in the adversarial API.

The fourth group, Table 4, involves executing a load of 1000 requests, separated into two subgroups, with 500 requests for each subgroup. The first subgroup experimented only requests of *Read* operations, which targeted the Java/JPA-based service API (see line 1 of that table). The Java/JPA implementation was coded into the microservice. The second subgroup, in turn, also aimed to investigate the behavior of the persistence API under the same set of operations. See line 2 in Table 4.

Furthermore, from the experiment in Table 4, we observed that the average elapsed time due to the persistence service API was close to over 2/3 higher than the average time of the adversarial API. As for CPU usage, concerning the effort to meet the demand of the `mDR` API, its usage was slightly higher than that of the adversarial API. The same can be said regarding the CPU behavior to serve `mSQLDB`. Regarding memory usage, the total memory used to serve requests from five models was 578 MB, compared to the total memory of 218 MB used to serve a single model in the adversarial API.

The objective of these experiments is to evaluate how much the overhead of the interpreted approach causes performance degradation in the proposed persistence service compared to the conventional execution. The microservice that implements the conventional approach in a single `t2.small` instance receives the request, processes it, and dispatches it directly to the database. The proposed persistence service approach performs the same task through two microservices in different `t2.small` instances. However, the processing load necessary for the interpretation activity and the transformation between representations is concentrated in just one of the microservices—the `mDR`.

In this sense, in at least two of the four experiments, the proposed persistence service presents a processing time close to or shorter than the adversarial API in the context of operations that involve changing the state of data in the database. In *Read* operations, however, the proposed persistence service consistently underperformed.

It is also relevant to note that to run the data persistence service API for five data models, the proposed persistence service required a consumption of up to, in any case, less than double the consumption of the adversary API—the conventional implementation, meeting the demand for 1 data model consumed a maximum of 222 MB of memory in the experiments. Thus, the proposed persistence service suggests a more efficient use of memory under the conditions of these experiments.

V. RELATED WORK

Existing work [6], [10] shows that the model interpretation approach remains almost unexploited. Although we have extensively searched for research that exploited an interpretation strategy beyond the initial phases of software development (analysis, validation, and simulation), such studies

TABLE 2. Second Experiment

Models	Implementation	Requests	Operation	CPU (%) (peak)	Memory(MB) (peak)	Elapsed Time - AVG(ms)
1	Java/JPA	500	C,U,D	1.2	220	13
5	Proposed Service	500	C,U,D	1.8(mDR), 1.7(mSQLDD)	162(mDR) & 322(mSQLDD)	15

TABLE 3. Third Experiment

Models	Implementation	Requests	Operation	CPU (%) (peak)	Memory(MB) (peak)	Elapsed Time - AVG(ms)
1	Java/JPA	100	R	1.1	202	25
5	Proposed Service	100	R	1.7(mDR), 1.9(mSQLDD)	151(mDR), 243(mSQLDD)	35

TABLE 4. Fourth Experiment

Models	Implementation	Requests	Operation	CPU (%) (peak)	Memory(MB) (peak)	Elapsed Time - AVG(ms)
1	Java/JPA	500	R	1.9	218	17
5	Proposed Service	500	R	2.2(mDR), 1.8(mSQLDD)	162(mDR), 416(mSQLDD)	29

proved to be quite scarce. We highlight some of these works as follows.

In the *Deep Models@run.time for User-Driven Flexible Systems* [16], the focus is on the use of the *Deep Models* approach [2]. From this, the central concept that is explored is the *Clabject Hierarchies* [11] — an abstraction of a dual nature that authorizes the author to embody metadata and data in it, creating an opportunity to carry out modeling and code construction at the same time. The author finds in *Clabjects* an opportunity for constructs capable of mutating their representations at runtime, making the system more flexible and evolving synchronously in relation to its data model. In this sense, using this approach requires its adopters to be able to deal with *Clabjects*, a lesser-known concept. Besides employing model interpretation, our approach uses the well-known UML Class diagrams.

In *Modeling Low-Code Databases With Executable Models* [5], class diagrams are employed to generate database schemas automatically. The authors' approach captures data representations, performs transformations by mapping UML Class Diagrams to the underlying database, and creates it. Furthermore, its construction also delivers a RESTful API for accessing data from the generated database. Its central objective is to offer undergraduate students in IT courses a standalone tool capable of facilitating the process of building databases and accessing their data through a RESTful API. Unlike our approach, the authors intentionally avoid the complexities associated with production level by focusing on educational purposes.

In turn, Driessen [8] proposes something similar to our work regarding using UML Class Diagrams as first-class artifacts, whether in the software development phase or the execution phase. However, this is done so that, when the software is executed, the data models are interpreted to generate the domain data class code—a hybrid approach. In this sense, as a condition for adopting this approach, the programming language must allow dynamic binding between the representation of new data classes, or changes to them, and the latest binary codes injected into the application for these classes at run time. In this sense, we chose not to create dependencies between the platform's services and the programming language used to build them.

In code generation, Torres et al. [27] address the challenge of persistence modeling in Model-Driven Engineering (MDE). Its main contribution is prioritizing the use of UML diagrams as the primary modeling language to serve technical and non-technical users, simplifying the persistence specification and facilitating communication between developers and domain experts. Furthermore, the approach integrates formal methods to check model consistency and ensure data integrity, addressing concerns about the reliability and correctness of the generated code. Another paper, *Data integration and interoperability: Towards a model-driven and pattern-oriented approach* [21], presents a model-driven, standards-centric approach to data integration and interoperability, prioritizing the representation of domain data through Entity-Relationship or UML Class Diagrams. While data persistence concerns remain outside its scope, it defines reusable design patterns to effectively integrate diverse data representations into legacy systems. Despite its relevance in using models to integrate different systems, it is supported by code generation. Distinctively from these two approaches, our work is entirely built using the model interpretation.

Finally, we also highlight the work of [12] who defines a metamodel to represent domain data in an IS, for its manipulation by end users, as well as the automated generation of a RESTful Web API to manipulate it. This is done to support the implementation and operation of the backend of applications in the context of he LCP service. However, although the work centrally implies the use of models to collaborate with the execution of the LCP platform, it does not offer a discussion on the aspects related to the persistence service itself, or its mode of operation.

In summary, we seek to move away from lesser-known techniques among industry practitioners by making them interested in using our data persistence service at a production level in a multi-tenant single-service instance manner; we decoupled the platform's services from the programming language used to build them; we use a fully interpretative approach to achieve flexibility from the users' perspective and, potentially, better results in terms of code and computing resource reuse.

VI. FINAL REMARKS

This paper has exploited the potential of MDE approaches in the context of LCPs, specifically focusing on using runtime model interpretation for data persistence services. We provide experimental results that suggest the technical feasibility of this approach to interpreting data models at runtime. We have presented an alternative LCP architecture that leverages a model interpretation approach to manage data access and persistence across multi-tenant software services, doing so more flexibly than automated or manual code generation approaches. The proposed architecture utilizes a subset of UML Class Diagrams as the primary modeling tool, enabling users to capture and represent data hierarchies with basic IT training. Given that the load test results yielded numbers that are comparable to the performance of solutions involving conventional domain model encoding (through manual or automatic generation), coupled with superior performance in terms of resource allocation and sharing, we deem it relevant to delve deeper into the research.

We have also detailed the implementation of our LCP prototype, presenting some components that, together, interpret models, translate them into database-specific instructions, and manage data access and persistence.

Although our work highlights the potential of MDE in LCPs using a data model interpretation approach, we do not exploit aspects related to scalability and security, as well as the limitations imposed by the need for isolation of request executions from different clients. Furthermore, investigating the use of more expressive data models, as well as the use of process models, is necessary to improve the capabilities of this LCP. Another relevant aspect involves the investigation of error handling mechanisms more suitable for distributed and multi-tenant architectures like the one presented in this platform.

DISCLOSURE STATEMENT

The first and third authors have shares in the software development and operation company Ycodify, which acts as a Low Code services platform. The second author declares that he has no competing interests.

ACKNOWLEDGMENTS

This work was supported by Coordenação de Aperfeiçoamento de Pessoal de Nível Superior—Brazil (CAPES)—Finance Code 001.

REFERENCES

[1] Ahmed, Usama, Imran Raza, and Syed Asad Hussain. 2019. “Trust evaluation in cross-cloud federation: Survey and requirement analysis.” ACM Computing Surveys (CSUR) 52 (1):1–37.

[2] Atkinson, Colin, and Thomas Kuhne. 2015. “In defence of deep modelling.” Information and Software Technology 64:36–51.

[3] Blair, Gordon, Nelly Bencomo, and Robert B. France. 2009. “Models@ run.time.” Computer 42 (10): 22–27. https://doi.org/10.1109/MC.2009.326.

[4] Bock, Alexander C, and Ulrich Frank. 2021. “Low-code platform.” Business & Information Systems Engineering 63: 733–740.

[5] Bubalo, Alan, and Nikola Tankovi ´c. 2023. “Modeling low-code databases with executable UML.” Human Systems Engineering and Design (IHSED 2023): Future Trends and Applications 112 (112).

[6] Ciccozzi, Federico, Ivano Malavolta, and Bran Selic. 2019. “Execution of UML models: a systematic review of research and practice.” Software & Systems Modeling 18: 2313–2360. https://doi.org/https://doi.org/10.1007/s10270-018-0675-4.

[7] Colantoni, Alessandro, Antonio Garmendia, Luca Berardinelli, Manuel Wimmer, and Johannes Brauer. 2021. “Leveraging Model-Driven Technologies for JSON Artefacts: The Shipyard Case Study.” In 2021 ACM/IEEE 24th International Conference on Model Driven Engineering Languages and Systems (MODELS), 250–260. IEEE.

[8] Driessen, Ralph. 2020. “UML class models as first-class citizen: Metadata at design-time and run-time.” Leiden University. Leiden Institute of Advanced Computer Science (LIACS) 1–42.

[9] Elshan, Edona, Ernestine Dickhaut, and Philipp Alexander Ebel. 2023. “An investigation of why low code platforms provide answers and new challenges.” In Proceedings of the 56th Hawaii International Conference on System Sciences.

[10] Galhardo, Pedro, and Alberto Rodrigues da Silva. 2022. “Combining Rigorous Requirements Specifications with Low-Code Platforms to Rapid Development Software Business Applications.” Applied Sciences 12 (19). https://doi.org/https://doi.org10.3390/app12199556, https://www.mdpi.com/2076-3417/12/19/9556.

[11] Henderson-Sellers, Brian, and Cesar Gonzalez-Perez. 2005. “The rationale of powertype-based metamodelling to underpin software development methodologies.” In Conferences in research and practice in information technology series, 7–16. http://hdl.handle.net/10453/1937.

[12] Hili, Nicolas, and Raquel Ara ´ujo de Oliveira. 2022. “A light-weight low-code platform for back-end automation.” In Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings, 837–846.

[13] Hintsch, Johannes, Daniel Staegemann, Matthias Volk, and Klaus Turowski. 2021. “Low-code development platform usage: towards bringing citizen development and enterprise it into harmony.” ACIS. https://aisel.aisnet.org/acis2021/11.

[14] Horvath, Benedek, Akos Horvath, and Manuel Wimmer. 2020. “Towards the next generation of reactive model transformations on low-code platforms: three research lines.” MODELS'20, New York, NY, USA. Association for Computing Machinery.

[15] Indamutsa, Arsene, Davide Di Ruscio, and Alfonso Pierantonio. 2021. “A Low-Code Development Environment to Orchestrate Model Management Services.” In Advances in Production Management Systems. Artificial Intelligence for Sustainable and Resilient Production Systems, edited by Alexandre Dolgui, Alain Bernard, David Lemoine, Gregor von Cieminski, and David Romero, Cham, 342–350. Springer International Publishing.

[16] Kegel, Karl, and Ing Sebastian G¨otz. 2022. “Deep Models@ Run. time for User-Driven Flexible Systems.”

[17] Kass, Sebastian, Susanne Strahringer, and Markus Westner. 2023. “Practitioners' Perceptions on the Adoption of Low Code Development Platforms.” IEEE Access 11: 29009–29034. https://doi.org/10.1109/ACCESS.2023.3258539.

[18] Martinez, Eder, and Louis Pfister. 2023. “Benefits and limitations of using low-code development to support digitalization in the construction industry.” Automation in Construction 152: 104909. https://doi.org/https://doi.org/10.1016/j.autcon.2023.104909, https://www.sciencedirect.com/science/article/pii/S0926580523001693.

[19] Neto, Josino Rodrigues, Vinicius Cardoso Garcia, Andreza Leite de Alencar, Julio Cesar Damasceno, Rodrigo Elia Assad, and Fernando Trinta. 2012. “Software as a Service: Desenvolvendo Aplicacoes Multi-tenancy com Alto Grau de Reuso.” Sociedade Brasileira de Computacao .

[20] Peng, Dunlu, Lidong Cao, and Wenjie Xu. 2011. “Using JSON for data exchanging in web service applications.” Journal of Computational Information Systems 7 (16): 5883–5890.

[21] Petrasch, Roland J, and Richard R Petrasch. 2022. “Data integration and interoperability: Towards a model-driven and pattern-oriented approach.” Modelling 3 (1): 105–126.

[22] Philippe, Jolan, Helene Coullon, Massimo Tisi, and Gerson Sunye. 2020. “Towards transparent combination of model management exe-

cution strategies for low-code development platforms." In Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings, MODELS '20, New York, NY, USA. Association for Computing Machinery. https://doi.org/10.1145/3417990.3420206.

[23] Pinho, Daniel, Ademar Aguiar, and Vasco Amaral. 2023. "What about the usability in low-code platforms? A systematic literature review." Journal of Computer Languages 74: 101185. https://doi.org/https://doi.org/10.1016/j.cola.2022.101185, https://www.sciencedirect.com/science/article/pii/S259011842200082X.

[24] Rymer, John R, Rob Koplowitz, Christopher Mines, Sara Sjoblom, and Christine Turley. 2019. "The Forrester wave: low-code development platforms For AD&D professionals, Q1 2019." Forrester Report, Forrester.

[25] Song, Hui, Gang Huang, Franck Chauvel, and Yanshun Sun. 2010. "Applying MDE Tools at Runtime: Experiments upon Runtime Models." In Proceedings of the 5th International Workshop on Models at Run Time, edited by Nelly Becomo, Gordon Blair, and Franck Fleurey, Oslo, Norway, October. To be published, https://inria.hal.science/inria-00560785.

[26] Tisi, Massimo, Jean-Marie Mottu, Dimitrios S. Kolovos, Juan de Lara, Esther M Guerra, Davide Di Ruscio, Alfonso Pierantonio, and Manuel Wimmer. 2019. "Lowcomote: Training the Next Generation of Experts in Scalable Low-Code Engineering Platforms." In STAF 2019 Co-Located Events Joint Proceedings: 1st Junior Researcher Community Event, 2nd International Workshop on Model-Driven Engineering for Design-Runtime Interaction in Complex Systems, and 1st Research Project Showcase Workshop co-located with Software Technologies: Applications and Foundations (STAF 2019), CEUR Workshop Proceedings (CEUR-WS.org), Eindhoven, Netherlands, July. https://hal.science/hal-02363416.

[27] Torres, Alexandre, Renata Galante, and Marcelo S. Pimenta. 2011. "A synergistic model-driven approach for persistence modeling with UML." Journal of Systems and Software 84 (6): 942–957. https://doi.org/https://doi.org/10.1016/j.jss.2011.01.027, https://www.sciencedirect.com/science/article/pii/S0164121211000197.

[28] Vincent, Paul, Kimihiko Iijima, Mark Driver, Jason Wong, and Yefim Natis. 2019. "Magic quadrant for enterprise low-code application platforms." Gartner report .

[29] Yao, Mengdi, Donglin Chen, and Jennifer Shang. 2019. "Optimal overbooking policy for cloud service providers: Profit and service quality." IEEE Access 7: 96132–96147.

FIRST A. AUTHOR Julio Gustavo Costa was born in Natal, Brazil. He holds a Bachelor's degree in Computer Engineering from the Federal University of Rio Grande do Norte (UFRN), Brazil, in 2003, and a Master's degree in Systems and Computing from the same university in 2020.

He worked at i3C as a software analyst and developer from 2003 to 2007. He worked as a systems and projects analyst at the Municipal Government of Parnamirim, RN, Brazil, between 2006 and 2017.

Since 2020, he has been pursuing a Ph.D. in the area of Systems and Computing at the Graduate Program in Electrical and Computer Engineering/UFRN.

Since 2022, he has been a partner at Ycodify, a software development company, and serves as a software architect for the process modeling and automation platform.

SECOND B. AUTHOR Luiz M. G. Gonçalves holds a Doctorate in Systems and Computer Engineering from COPPE-UFRJ, Brazil, in 1999, which included a two years study at the Laboratory for Perceptual Robotics of UMASS, Amherst, USA.

He is a full Professor at the Computer Engineering Department of UFRN, Brazil. He has done research in several aspects of Graphics Processing including fields such as Robotics Vision (main interest), Computer Graphics, GIS, Geometric Modelling, Computer Animation, Image Processing, Computer Vision, and Robotics in Education.

THIRD C. AUTHOR Samuel Xavier-de-Souza was born in Natal, Brazil. He holds a Computer Engineer degree from Universidade Federal do Rio Grande do Norte-UFRN, Brazil, 2000, and a Ph.D. in Electrical Engineering from Katholieke Universiteit Leuven, Belgium, 2007.

He worked for IMEC, Belgium, as a Software/Hardware Engineer and for the Flemish Supercomputing Center, Belgium, as a High-Performance Computing Consultant.

In 2009, he joined the Department of Computer Engineering and Automation at UFRN, where he currently holds the position of Associate Professor. He is a founder and Director of the High-Performance Computing Center-NPAD at UFRN. In 2016, he became a Royal Society-Newton Advanced Fellow in 2016 for his research on Software Energy. In 2019, he became an IEEE Senior Member. His current research interests are high-performance and energy-efficient computing, scaleable and efficient parallel systems, scaleability profiling, visualization, analysis tools, parallel algorithms, parallel architectures, and their applications.