Experiences of Models@run-time with EMF and CDO

Daniel Seybold  
Institute of Information Resource Management, Ulm University, Germany  
daniel.seybold@uni-ulm.de

Jörg Domaschka  
Institute of Information Resource Management, Ulm University, Germany  
joerg.domaschka@uni-ulm.de

Alessandro Rossini  
Department of Networked Systems and Services, SINTEF, Oslo, Norway  
athanasios.tsitsipas@uni-ulm.de

Christopher B. Hauser  
Institute of Information Resource Management, Ulm University, Germany  
christopher.hauser@uni-ulm.de

Frank Griesinger  
Institute of Information Resource Management, Ulm University, Germany  
frank.griesinger@uni-ulm.de

Athanasios Tsitsipas  
Institute of Information Resource Management, Ulm University, Germany  
athanasios.tsitsipas@uni-ulm.de

Abstract

Model-driven engineering promotes models and model transformations as the primary assets in software development. The models@run-time approach provides an abstract representation of a system at run-time, whereby changes in the model and the system are continuously reflected in each other. In this paper, we report on more than three years of experience with realising models@run-time in scalable cloud scenarios using a technology stack consisting of the Eclipse Modeling Framework (EMF) and Connected Data Objects (CDO). We establish requirements for the three roles domain-specific language (DSL) designer, developer, and operator, and compare them against the capabilities of this technology stack. Our assessment is that EMF and CDO are well-suited for DSL designers, but less recommendable for developers and even less suited for operators. For these roles, we experienced a steep learning curve and several lacking features that hinder the implementation of models@run-time in scalable cloud scenarios. Moreover, we experienced performance limitations in write-heavy scenarios with an increasing amount of stored elements. While we do not discourage the use of EMF and CDO for such scenarios, we recommend that its adoption for similar use cases is carefully evaluated until this technology stack has realised our wish list of advanced features.

1. Introduction

Model-driven engineering (MDE) aims to improve the productivity, quality, and cost-effectiveness of software development by shifting the paradigm from code-centric to model-centric. In MDE, models and model transformations are the main artefacts of the development process. Often, models are specified using general-purpose languages such as the Unified Modeling Language (UML) [12]. Yet, to fully unfold the potential of MDE, models are specified using domain-specific languages (DSLs) tailored to a specific domain of concern, e.g., the Cloud Application Modelling and Execution Language (CAMEL) [19, 21].

Models@run-time [3] is an architectural pattern for dynamically-adaptive systems that leverages on models at both design-time and run-time. In particular, models@run-time provides an abstract representation of the underlying running system, whereby a modification to the model is enacted on-demand in the system, and a change in the system is automatically reflected in the model.

Categories and Subject Descriptors  D.2.4 [Software Engineering]: Software/Program Verification; D.2.9 [Software Engineering]: Management

General Terms  Documentation, Management, Experimentation

Keywords  model-driven engineering, models@run-time, model repository, eclipse modeling framework, connected data objects

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

SLE’16, October 31 – November 1, 2016, Amsterdam, Netherlands  © 2016 ACM. 978-1-4503-4447-0/16/10...$15.00  http://dx.doi.org/10.1145/2997364.2997380

http://camel-dsl.org
The combination of the Eclipse Modeling Framework (EMF\(^2\)) and Connected Data Objects (CDO\(^3\)) facilitates the development of models@run-time and represents the primary focus of this paper. EMF is the de-facto standard in MDE and consists of a framework for (meta)modelling and code generation. EMF provides Ecore, a core metamodel that allows Ecore models to be specified. CDO is part of EMF, and provides a persistence framework that works natively with Ecore models and their instances.

In this paper, we report on more than three years of experience using EMF and CDO to realise models@run-time in the three cloud research projects PaaSage\(^4\), CACTOS\(^5\)\([16]\) and CloudSocket\(^6\). In particular, we provide a three-fold contribution: (i) We identify three main roles in models@run-time, assign them specific tasks in the life-cycle of models@run-time, and derive requirements for tools in order to support models@run-time. (ii) For each role, we reflect on our experiences of more than three years of designing DSLs with, developing for, and operating EMF and CDO in scalable cloud scenarios. (iii) Based on these experiences, we derive a wish list for the EMF and CDO developers on how to extend the tools.

The reminder of the paper is organised as follows. Section 2 introduces a models@run-time scenario in the cloud, derived from three research projects; identifies the related roles; and derives their respective requirements. Section 3 sketches the technology stack consisting of EMF and CDO. Section 4 describes our experiences of implementing models@run-time for each role as well as our experiences of performance. Section 5 discusses our experiences and presents the lessons learnt and a wish list of advanced features. Section 6 discusses related work, before Section 7 concludes the paper and outlines future work.

2. Models@run-time for the Cloud

We acquired our experiences in the PaaSage, CACTOS, and CloudSocket projects, which share the adoption of models@run-time to support the execution of cloud applications.

PaaSage delivers a platform to support the modelling and execution of cloud applications on multiple clouds together with an accompanying methodology to allow model-based provisioning, deployment, and adaptation of these applications independently of the existing underlying cloud infrastructures.

CACTOS provides a toolset for data centre operators, which facilitates dealing with the increasing scale, heterogeneity, and complexity in cloud data centre infrastructures. CACTOS relies on a model-based representation of the physical and virtual infrastructure and derives self-adaptive optimisations based on these models.

CloudSocket uses concept models and semantics to align domain-specific business processes with executable workflows. These are enabled in a multi-cloud environment addressing business needs by the Business Process as a Service (BPaaS) paradigm.

The PaaSage, CACTOS, and CloudSocket projects have in common: (i) a centralised models@run-time environment to allow the programmatic manipulation of models; (ii) a centralised actuator to enact changes by reasoners; (iii) probes deployed on multiple nodes, virtual machines, and workflow engines to collect metrics at multiple layers of the cloud stack; and (iv) a technological stack consisting of EMF and CDO.

From these commonalities, we derive a models@run-time scenario in the cloud.

2.1 Models@run-time Scenario in the Cloud

Figure 1 sketches the models@run-time architecture of our scenario in the cloud: A reasoner reads the current model in the CDO Server (step 1) and produces a target model (step 2). Then, an actuator computes the difference between the current model and the target model, and enacts the adaptation by only modifying the parts of the application necessary to account for the difference (step 3). Finally, the actual ap-
2.2 Models@run-time Roles

While working with models@run-time, we observed that the same responsibilities were assigned to groups of individuals in all three projects. This led to the definition of the following three roles: **DSL designer**, **developer**, and **operator**. While the DSL designer is well-known in MDE, the others have barely been considered by other authors, especially in the context of models@run-time. Yet, in practice, they are crucial for realising system integration and operation.

Please note that the typical role of the **modeller**, who specifies and manipulates models in a general-purpose or domain-specific language, is beyond the scope of this paper. This is because we are concerned with the run-time manipulation of models performed by software components such as reasoners, rather than the design-time manipulation of models performed by humans. The interested reader may consult [10] for an empirical study of the state of the practice and acceptance of MDE, which focuses on the modeller.

The **DSL designer** describes the modelling concepts, their properties and their relations, and rules for combining these concepts to specify valid models in a DSL. Usually, this is realised by a metamodel along with attached constraints.

The DSL designer requires a friendly **user interface** (UI) to specify metamodels. Moreover, the DSL designer requires another UI to specify and attach **constraints** to metamodels, which ensure the **validity** of models at both design- and run-time. Moreover, the DSL designer requires support for the **versioning** and co-evolution of metamodels, which enables an iterative development process. Finally, the DSL designer requires that the **number and complexity** of metamodels and models is not limited by technical constraints.

The **developer** writes business code that connects third-party systems with the model repository. In particular, the developer provides actuators that transform model changes into actions performed on the underlying system, as well as the probes that gather data from the underlying system and feed them into the model.

The developer relies on an **API** to interact with the repository. To avoid constant polling for tracking changes, the developer requires **push notifications** from the repository. Moreover, to enable **concurrent access** to models, the developer requires a mechanism to ensure the **consistency** of the repository, ideally enhanced with mechanisms that ensure the **validity** of models. Finally, when accessing the repository via the API, the developer requires that **growing workloads** shall not lead to the starvation of queries.

The **operator** runs the infrastructure including the model repository, the back-end services, the actuators, and the probes. The operator wires all entities, enabling their communication, and enacts the security and integrity of the repository by providing adequate access control as well as backup mechanisms.

The operator is dependent on **up-to-date documentation** in order to configure the system. Moreover, the operator requires a mechanism for **monitoring** the performance of the repository in order to analyse and fix performance bottlenecks, e.g., by scaling out. Furthermore, the operator requires **fine-grained security access** in order to ensure system integrity. In addition, the operator requires **durability** of stored data in order to cope with long-running services. Finally, the operator is supported by an administrative **user interface**.

2.3 Requirements Towards Models@run-time

In order to evaluate a models@run-time environment, we introduce requirements based on the descriptions of the roles. Table 1 lists those consolidated requirements and shows their relation with roles. **Transaction** allows applying atomic updates to and consistent reads from the repository as well as concurrent access. **Validation** allows verification of the well-formedness of the models. **Versioning** supports accessing the history of changes of metamodels and models. In addition, it enables migration of the models in the repository to newer versions of their metamodel. The semantics of **scalability** is three-fold: For the DSL designer, it allows defining metamodels of arbitrary size; for the developer, it allows using models of arbitrary size; whereas for the operator, it enables extending the repository at run-time with new storage and processing power to provide a sufficient storage size and to serve enough requests. **Access control** allows specifying and enforcing access rights to different parts of the repository in a sufficiently fine-grained manner. **Durability** allows for storing models in a persistent way that survives crashes and server out-takes. **Notification** allows the installation of callback handlers that indicate changes in the models. **Usage interfaces** summarise all ways of interacting with the repository, ranging from defining metamodels to accessing and modifying existing models to the administration of the repository, e.g., for registering metamodels or changing access rights. Besides the requirements towards models@run-time, the overall performance experience, especially in a large scale set-up, is of interest.

3. Technology Stack

In this section, we describe the features of the technology stack we adopted in the context of self-adaptive, cloud applications. The stack description is not intended to be exhaustive, but only aims to present the necessary background for the reasoning in the remainder of this paper. We start with an overview of EMF followed by a sketch of CDO and its features.
| Working area      | DSL designer metamodels | Developer models | Operator system |
|------------------|-------------------------|-----------------|----------------|
| Transaction      | consistency, concurrency|                 |                |
| Validation       | constraints             | stability       |                |
| Versioning       | evolution               | history         |                |
| Scalability      | metamodel size          | model size      | load capability|
| Access control   |                         | security        |                |
| Durability       |                         | data persistence|                |
| Notification     |                         | change propagation|              |
| Usage interfaces | creation                | access models   | administration |

Table 1. Requirements towards models at run-time per role

Figure 2. CDO platform architecture in a self-adaptive cloud context

3.1 Eclipse Modeling Framework (EMF)

In EMF, metamodels are Ecore models that conform to the Ecore metamodel (cf. Figure 2). The Ecore metamodel, in turn, is an Ecore model that conforms to itself (i.e., it is reflexive). Ecore models and their instances can be either manipulated manually through UIs or programmatically through Java APIs.

EMF provides frameworks such as Eclipse OCL\(^8\) to validate models against Object Constraint Language (OCL)\(^9\) constraints attached to the metamodel. It also provides additional frameworks such as the Graphical Modeling Framework (GMF\(^10\)) and Xtext\(^11\) to define concrete (graphical or textual) syntaxes and to realise integrated development environments (IDEs) for DSLs.

Finally, EMF comes with three persistence providers: XML Metadata Interchange (XMI)\(^12\), EMFStore\(^13\), and CDO. This paper only considers CDO as it is the most mature persistence provider\(^14\) for distributed applications.

3.2 Connected Data Objects (CDO)

CDO is a semi-automated persistence framework that works natively with Ecore models and their instances. It abstracts from the specific underlying database and automatically takes care of the bi-directional mapping of models to (relational) data.

\(^8\)http://wiki.eclipse.org/OCL
\(^9\)https://www.eclipse.org/modeling/gmp/
\(^10\)https://eclipse.org/GMF
\(^11\)https://eclipse.org/Xtext/
\(^12\)http://www.eclipse.org/emfstore/

The CDO Server represents the model repository comprising the actual server and a database as storage back-end as depicted in Figure 1. It supports several database management systems (DBMS), including relational databases such as MySQL, but also NoSQL databases such as MongoDB.

3.2.1 Client-side Aspects

The Java API represents models as sets of CDO objects. CDO objects accessed by the client remain logically attached to their pendant in the repository. This mechanism enables CDO to support fully-fledged lazy loading and notifications of changes.

CDO revisions allow accessing and reproducing the history of changes of CDO objects. CDO provides the concept of transactions and views to deal with concurrent access to the repository. A transaction has to be used for write operations, while views are supposed to be applied for read-only access.

Based on CDO objects and CDO revisions, CDO supports the concept of optimistic versioning\(^15\). This means that clients first change the state of objects locally before resynchronizing the changes with the server, e.g., through committing a transaction. In addition to its Java API, CDO offers the query language interfaces Structured Query Language (SQL), Object Constraint Language (OCL), and Hibernate Query Language (HQL) to retrieve and manipulate content.

\(^15\)http://www.eclipse.org/emfstore/
3.2.2 Server-side Aspects

The server implementation is based on OSGi. This modular platform promises an extensible architecture where additional features such as security can be enabled by adding the respective bundles to the server core.

The mapping from CDO logic to DBMS invocations is realised by database store bundles. Each store comes with dedicated capabilities and supports a limited number of DBMS. For instance, the DB store can be associated with any DBMS supported by a JDBC driver and the MongoDB store uses the document-oriented MongoDB for persistence.

3.3 CDO Features Towards Models@run-time

Based on the outlined conceptual and technical capabilities, CDO offers a feature set that satisfies the design-time and run-time requirements of the self-adaptive cloud applications as described in Section 2.3. Table 2 shows the models@run-time requirements and the respective CDO feature that claims to fulfil the requirements.

4. Experiences

In this section, we summarise our experiences with EMF and CDO in the context of models@run-time for cloud applications. For each of the roles, we compare EMF and CDO’s capabilities from Section 3.3 against the role-specific requirements defined in Section 2.3. The section concludes with a performance evaluation in Section 4.4.

4.1 Experiences as a DSL Designer

EMF and CDO were developed with MDE in mind. Therefore, they were able to satisfy many of the requirements from a DSL designer.

On the positive side, we found EMF and its IDE sufficiently intuitive and effective for the specification of metamodels. Moreover, we did not experience any problems with the scalability of metamodels. Furthermore, OCL proved to be sufficiently expressive to specify complex constraints for capturing (part of) the semantics. Its integration in both EMF and CDO effectively supported the validation of models at both design-time and run-time.

On the negative side, we experienced a steep learning curve for OCL. Besides, we were forced to manually co-evolve Ecore models along with attached OCL constraints—a time-consuming and error-prone task. In addition, we suffered from limited support for model versioning and co-evolution. Hence, we were forced to manually migrate models to the updated metamodels—again, a cumbersome and error-prone task.

4.2 Experiences as a Developer

From a developer’s perspective, the interfaces offered by APIs were sufficient, but not well documented or out-dated. In addition, all three projects faced restrictions in using third-party libraries with CDO due to the binding with EMF, e.g., restrictions in serialising/deserialising CDO Java Objects to/from a JSON representation.

Scalability was not problematic: the size of the models had no impact on functionality. However, the interoperability with the model repository was affected in terms of performance and latency (cf. Section 4.4). The lazy loading increased the performance when working on small parts of the models. However, when working on the whole model, lazy loading had a negative impact on performance and could not be used in wide-area networks due to high latency.

Similarly, the transaction support offered by CDO eased the realisation of consistency. Yet, the type of DBMS used as back-end actually impacted the semantics of transactions and the behaviour of the server. This required adapting the client code to the configuration of the server. Besides, the optimistic versioning of CDO required a large portion of error handling code in the client application to handle exceptions thrown when concurrent updates occurred.

The validation of models enhanced the stability of the overall system. Yet, it also required the developer to deal with possible constraint violations in the code, which added additional complexity to the actual application. Finally, an update of the metamodel required the rewrite and adaptation of all client applications as the concept of data transfer objects is lacking and models are directly exposed to clients.

While none of the projects utilised the history and the branching feature offered by CDO, they made use of the notification mechanism. Due to marginal documentation for the different CDO notification concepts, i.e., Adapter and IListener, we switched to a polling approach. While polling is normally considered suboptimal, it helped us to improve the stability of the code and hence, of the overall system.

4.3 Experiences as an Operator

We experienced operating the CDO Server to be a cumbersome and error-prone task also due to marginal documentation. In all three projects, we had to manually export the CDO Server with additional bundles out of Eclipse in order to ensure that persistence and security mechanisms were in place. The interested reader may consult our blog post for technical details. This required in-depth knowledge of OSGi and Eclipse, and to some extent limited the automation of the server administration, e.g., using DevOps tools. Administrative tools for the operator were limited to the basic Eclipse

11https://wiki.eclipse.org/CDO/DB/MySQL
12https://wiki.eclipse.org/Net4j/DB/MySQL
13https://download.eclipse.org/modeling/emf/cdo/drops/R20140610-0212/help/org.eclipse.emf.cdo.doc/html/reference/StoreFeatures.html
14http://www.cactosfp7.eu/2015/04/03/cactos-blog-setting-secure-cdo-server/
| Working area      | Relation to CDO feature                                                                 |
|------------------|-----------------------------------------------------------------------------------------|
| Transaction      | Transactional manipulations of the models persisted in the repository.                   |
| Validation       | Automatic checking of the conformance between the models persisted in the repository against their metamodel along with the attached constraints. |
| Versioning       | Optimistic versioning [20] of the models persisted in the repository, where each client of the repository has a local (or working) copy of the models. |
| Scalability       | Arbitrary size of the models persisted in the repository, exploiting mechanisms such as caching and lazy loading. |
| Access control   | Role-based access control to the models persisted in the repository, supporting the controlled access to (parts of) models. |
| Durability        | Multiple store implementations with different databases.                                  |
| Notification      | Automatic notification of clients about changes to the state of the models persisted in the repository. |
| Usage interfaces | Eclipse-based UI and API for direct and programmatic manipulation of models, respectively. |

**Table 2. Requirements mapped to CDO features**

UI, which offers basic access to the currently stored models. A shell or scripting interface was not available. The only administrator support was a configuration file which, however, only exposes a subset of the necessary configuration options.

In our experience, we could not find a different way to change the database store except rebuilding the CDO Server. Similarly, we could only realise access control by building a custom version including the Security Manager.

The durability of a set-up, just like consistency, was heavily dependent on the DBMS back-end. Crashes of the CDO Server and connected clients occurred when developers used features not provided by the current configuration (e.g., transactions). In those cases, restarting the server as a means of recovery often did not work and in a few cases led to complete data loss. All data was also lost when new metamodels were installed, which required wiping the storage and rebuilding the structure of the database.

Regarding scalability, the CDO Server in our usage scenarios proved to be a performance bottleneck when many probes and actuators were installed and/or a high access frequency was used. The operators suffered from the lack of built-in tools to analyse the performance and the lack of configuration guidelines. In all cases, the operators had to dig into code to analyse performance bottlenecks and provide possible optimisations. While vertical scaling is physically limited, horizontal scaling was conceptually possible when using a scalable database such as MongoDB. Also, horizontal scaling at CDO level, i.e., using multiple CDO Servers was an option. Experimenting with both approaches is subject to ongoing evaluations. Yet, in both cases we are challenged by sparse documentation and often outdated documentation.

### 4.4 Performance Experiences

While working with CDO, we experienced a satisfying performance during the development and testing phases, but struggled in production set-ups. Mainly for large-scale and long-running applications, the performance behaviour of

**Table 3. Configuration of environment**

CDO was hard to predict and yielded surprising results. This section aims to illustrate some of these results through a set of experiments.

#### 4.4.1 Experiment Set-up and Methodology

The environment consisted of a server and a client deployed on two hosts (cf. Table 3). Both hosts resided in the same local area network with a maximum bandwidth of 1 Gb/s. The round trip time (rtt) between the server and client for 100 packets was on average 0.737 ms with a deviation of 0.120 ms. MySQL was used as a database server. The CDO Server and database back-end were deployed on the same host implemented as a virtual machine in an OpenStack

https://www.mysql.com/
cloud environment. CDO used the DB store and hence the JDBC driver to connect to the database via TCP. The CDO Server was built with security features and access control enabled. Apart from the DB store configuration, the CDO Server used the default configuration settings (e.g., versioning enabled).

All the experiments were run from a single client machine with only a single thread, hence there was no concurrent access to the CDO Server. The connection to the server was opened once at the beginning of each experiment and then reused throughout the experiment. This allowed CDO to enable caching on the client-side. We then measured the time on the client-side, if not stated differently. Hence, all the results presented in the following serve as a lower bound for performance. The following results are derived from three independent runs and hence show an average value with deviations.

The use of CDO in models@run-time scenarios meant that the system used both read and write operations with read operations dominating. Nevertheless, in all three projects we also experienced a constant load on write operations leading to growing models. This is mainly caused by the fact that (parts of) monitoring information is stored directly in CDO models.

4.4.2 Write Performance

To validate the behaviour with an increasing amount of elements, the write experiment repeatedly opened a transaction, added 100 new elements, and committed the transaction. This sequence was repeated 100 times so that at the end of the experiment 10,000 new elements were written to the CDO Server. Between two transactions, the thread waited several seconds to enable an interleaving read experiment.

As new elements, we took optimisation plans from the CACTOS project18. Each plan contained a root step, which in turn contained two optimisation actions. All elements had a status property, while plans had a date property in addition. Actions referred to a fixed model in a different CDO resource. Besides, plans were standalone objects without additional payload data.

Figure 3 shows the results of this experiment. It contains the time required for opening a transaction (purple line) and for committing the 100 new elements (green line). While the time for opening a transaction was \(O(1)\), the time required for committing new elements depended heavily on \(n\), the number of elements already stored in the server. In particular, it grew with \(O(n)\). In the worst case, it took almost seven seconds to add 100 new elements.

4.4.3 Read Performance with Growing Model Size

The first read experiment elaborated the behaviour of reading the most recent models from the CDO Server while the server stored an increasing number of elements. The experiment interleaved with the write experiment as follows: whenever the write experiment slept after creating new elements, the first read experiment read 100 randomly chosen elements from the server in a view (i.e., read-only transaction). This way the benchmarks never interfered or overlapped with each other.

Figure 4 shows the results of the experiment. The diagram shows that the time needed for reading 100 elements was independent from the number of elements stored and after a warm-up phase drops to a constant value.

4.4.4 Read Performance with Growing Read Size

The second read experiment elaborated the behaviour of reading a growing number of the most recent elements from a CDO Server storing 10,000 elements. In the first iteration, 100 elements were read. Then, 99 further iterations were run and in each iteration, 100 additional elements were read. Hence, the last iteration read all elements in the CDO Server.

Figure 5 shows the results of the experiment. Not surprisingly, the time consumed for reading a linearly increasing number of elements from the CDO Server increased linearly. Hence, on average the time needed for reading an individual element remained constant.

18 https://sdqweb.ipd.kit.edu/eclipse/cactus/optimisationplan/nightly/
4.4.5 Resource Utilisation in Experiment Set-up

While running the three experiments, the server and client were continuously monitored. The monitored metrics for the CDO related processes were CPU utilisation, memory consumption, and the used network bandwidth, separately for the client and the server-side. Analysing the monitoring data, the CPU and memory utilisation on both the client- and server-side was reasonable in all experiments: the load was rather stable, and never exceeded a noteworthy threshold of less than 10% CPU and 13% memory (see Figures 6 and 7). Yet, while executing the write performance experiment, the network utilisation on the server-side showed a similarity to the time required for committing new elements to the CDO Server.

4.4.6 Summary

The experiments showed that the read performance experienced widely depended on the number of elements read. The write performance for adding new elements, however, decreased with the size of the model, which was limiting. Although CDO offers several configuration parameters for performance tweaking, none of the configurations notably influenced the write performance. The utilisation metrics showed a clear relation with the network bandwidth on the server-side. Improving the performance of the CDO Server in write heavy use cases is subject to ongoing investigations.

5. Discussion

This section discusses the consequences of the experiences described in Section 4. It starts with a concluding summary, then presents the lessons learnt, and finally presents a wish list of desired features in CDO that eases the life of developers and operators in particular.

5.1 Experience Summary

Our experiences show that EMF and CDO have been developed for MDE. In short, it is better suited for use, the more abstract a user’s role is and the less automation is

https://wiki.eclipse.org/CDO/Tweaking_Performance
needed. A DSL designer sees a rich technology stack in EMF and CDO. The specification, validation, and versioning of metamodels and models is well supported by the Eclipse IDE. Based on the extensibility of the Eclipse IDE specific tools can also be integrated, e.g., OCL or Xtext.

For developers, CDO offers basic features that fully support implementing actuators and probes in a models@runtime environment. Yet, the programming model requires getting used to. Even experienced (Java) programmers have to adapt their programming style to constraints imposed by CDO. For instance, the way CDO treats the equality and identity of objects is contrary to how the Java collections framework deals with these aspects. In our experience, the sparse and often outdated documentation does not sufficiently guide inexperienced CDO developers. From an architectural point of view, data transfer objects [4] are missing. This lack of abstraction layer leads to a tight coupling between CDO-specific and actual application code in client applications. The fact that CDO-specific code in client applications depends directly on the set-up of the CDO Server is sub-optimal from a management point of view, as applying changes to the server configuration may influence client applications.

Among all the three roles, operators are the least pleased with CDO. This is probably due to the fact that they are furthest away from the DSL designer. Also, operators are not directly foreseen in the MDE paradigm. While CDO offers mandatory features such as access control and support for different databases, the set-up, monitoring, and operation of a CDO Server is a widely manual task that is time-consuming and error-prone; often complicated by the marginal documentation. Operators in all three projects are missing administrative tools and find performance issues difficult to resolve.

5.2 Lessons Learnt
Not all negative experiences in the three projects stem from the technical domain: Operators suffer from the behaviour of DSL designers and developers. Here, we experienced that the workarounds these two groups introduce to cope with shortcomings and to realise advanced features, fall at the operators’ feet.

This is due to the fact that many of the workarounds lead to a frequent update of the metamodels or require a frequent reconfiguration of the database. The easy access to metamodels and their high abstraction level tempts DSL designers in particular to do quick updates to the metamodel. As a consequence, this imposes a load on the developers who have to adapt their code to the new models and particularly on the operators who not only have to reconfigure the CDO Server, but also to redeploy all probes and actuators to support the new metamodels. Here, automated deployment provides considerable help. Yet, this is complicated because of the way CDO is built and operated (cf. Section4.3).

Due to organisational shortcomings with frequently updated models, CACTOS and CloudSocket suffered from data loss on a regular basis. PaaSage, in contrast, established a rigid release cycle of both metamodels and code dependent on the metamodels that avoided this trap.

The experiments show that the read performance mostly depends on the number of elements read. The write performance for adding new elements, however, grows with the size of the model. This requires specific counter measures to be taken by the system architect. First, CDO views should be used instead of transactions whenever possible. Second, each client in the system should have a performance configuration[20] that takes into account this client’s way of access and operation frequency to the repository. Third, whenever possible, models should be designed such that they have a fixed size and changes are realised by updating existing elements instead of creating new elements. As mentioned above, the realisation of all these steps requires rigidity from a management and quality assurance point of view.

5.3 Desired Features
For the future, we expect more advanced features for both EMF and CDO that truly enable its use in production.

Crucial to this and a more wide-spread uptake of the technology is comprehensive and up-to-date documentation. This helps developers to use the correct and up-to-date APIs, and operators to pick the right database and right configuration for their particular use case. Operators (and system administrators) also benefit from more support tailored to their user group. Here, the provisioning of sophisticated UIs plays a major role: They can help with installing, updating, and managing (new) metamodels; they support the configuration and entire set-up of the CDO Server; they can help with changing database back-ends or scaling out the CDO deployment. Finally, they support the monitoring of CDO, the hardware, and the system usage and provide suggestions on how to improve the configuration. The provisioning of pre-built binaries helps the integration of the CDO Server and client bundles in automated build chains. Moreover, it eases the deployment and update not only of the server, but of the entire system.

Native support for the co-evolution of metamodels and attached constraints as well as the co-evolution of metamodels and models [11] is absolutely necessary when working with rapidly evolving DSLs. Solving this problem would also give EMF and CDO—and consequently MDE—a competitive advantage over the use of mere database mechanisms (e.g., tables) or object-relational mappers, which both suffer from the same problem but have not been able to provide a satisfying solution.

For developers, a more generic abstraction layer would ease integrating CDO with other parts of the system. In particular, the provisioning of a layer with DTO (data transfer
objects) functionality could decouple the release cycles of metamodels and client code.

6. Related Work

Other authors approve our view that CDO is well-suited as a model repository from a DSL designer’s perspective due to its support for persistence, collaborative work, and versioning [5]. Yet, due to the lack of support of scalable models it is proposed to shift the scaling responsibility to the DSL designer. Other authors identify that using relational DBMS as storage back-ends limits the performance and scalability of CDO [11]. A discussion and comparison of new storage solutions concludes that NoSQL-based repositories overcome these limitations [1].

A roadmap towards scalability in MDE is presented in [9]. The authors suggest further research in model query languages, scalable model repositories and model transformations. CDO as the state is highlighted for supporting collaboration based on transactions and views, but the lack of mature support for conflict management and merging is noted. As an alternative, Morsa is pointed out with a focus on scalable persistence but without access control, version management or security [6].

The Morsa approach [6] targets the scalability of the model repository. CDO is referred to as the most mature model repository for EMF. Yet the scalability of CDO is criticised, especially for large models (> 600 MB). Hence, a new model repository is introduced, focusing on performance and scalability instead of consistency. Based on the EMF resource API and MongoDB as storage back-end, Morsa promises better scalability than CDO. Yet [6] does not evaluate CDO with MongoDB against their approach.

Neo4EMF [2] is a similar approach to improve scalability. Instead of using MongoDB, they propose the graph-based Neo4j database. CDO is referred to as the de facto standard model repository for EMF based on its features, however operator-related requirements such as monitoring or horizontal scalability are missing.

EMF in combination with CDO has been examined in various scenarios including models@run-time. CDO for models@run-time is discussed by [7]. The requirements for models@run-time are outlined and correlated to CDO and further model repositories. The authors conclude that CDO only partially fits the needs of models@run-time, but that there is currently no available solution that fits all requirements. Therefore, they propose the Kevoree Modeling Framework, which provides a toolset to model, structure, and reason about data at design- and run-time, while introducing as little overhead as possible. CDO in the cloud context is considered by [11] with a focus on co-evolving metamodels and models. In this context, the lacking support of CDO for model migration is further elaborated, leading to a solution proposal.

Finally, [18] reports on a number of collaborative repositories in MDE, including repositories that are based on technologies different from EMF and CDO, although the emphasis is not on models@run-time.

7. Conclusions and Future Work

In this paper, we reported on more than three years of experience with realising and operating models@run-time in cloud scenarios using a technology stack consisting of EMF and CDO. First, we identified three main roles in models@run-time, namely DSL designer, developer and operator. For each role, we assigned the specific tasks in the life-cycle of models@run-time, and derived requirements for tools to support models@run-time. Next, we introduced the technological stack of EMF and CDO. We claimed that EMF and CDO basically support the models@run-time paradigm. Yet, they do not support all three roles to the same extent and show performance limitations. CDO is well-suited for DSL designers and we only experienced minor divergences with our requirements. In contrast, the use of CDO may be cumbersome for developers and operators. For both roles we experienced a steep learning curve and several lacking features that hindered the implementation of models@run-time in scalable cloud scenarios. Based on our experiences, we derived a wish list for the EMF and CDO developers on how to extend the tools.

Concluding, we do not discourage the use of EMF and CDO for our scenario, but we recommend that its adoption for similar use cases is carefully evaluated and that general best practices in software engineering are adopted. In particular, changes to metamodels and their impact on the models@run-time environment require the establishment of strict project management and rigid release and update cycles.

Future work will focus on a quantitative evaluation of the experiences, in particular with respect to an extended performance evaluation. Based on artificial and actual load, we will produce a performance model that helps to identify bottlenecks and scaling options beyond the performance experiments presented in Section 4.4. Ideally, the results of the extended performance evaluation will lead to an operator guide for the CDO Server for productively used and large scale setups.

Acknowledgments

The research leading to these results has received funding from the European Commission’s Seventh Framework Programme under grant agreement numbers 317715 (PaaSage) and 610711 (CACTOS), and the Horizon 2020 framework programme under grant agreement numbers 644690 (Cloud-Socket) and 645372 (ARCADIA).
References

[1] K. Barmpis and D. S. Kolovos. Comparative Analysis of Data Persistence Technologies for Large-scale Models. In *XM 2012: Extreme Modeling Workshop*, pages 33–38. ACM, 2012. ISBN 978-1-4503-1804-4. doi: 10.1145/2467307.2467314.

[2] A. Benelallam, A. Gómez, G. Sunyé, M. Tisi, and D. Launay. Neo4EMF, A Scalable Persistence Layer for EMF Models. In J. Cabot and J. Rubin, editors, *ECMFA 2014: 10th European Conference on Modelling Foundations and Applications*, volume 8569 of *Lecture Notes in Computer Science*, pages 230–241. Springer, 2014. ISBN 978-3-319-09194-5. doi: 10.1007/978-3-319-09195-2_15.

[3] G. Blair, N. Bencomo, and R. France. Models@run.time. *IEEE Computer*, 42(10):22–27, 2009. doi: 10.1109/MC.2009.326.

[4] W. Crawford and J. Kaplan. *J2EE Design Patterns*. O’Reilly & Associates, Inc., 2003. ISBN 978-0596004279.

[5] J. Domaschka, C. B. Hauser, and B. Erb. Reliability and Availability Properties of Distributed Database Systems. In M. Reichert, S. Rinderle-Ma, and G. Grossmann, editors, *EDIC 2014: 18th IEEE International Enterprise Distributed Object Computing Conference*, pages 226–233. IEEE Computer Society, 2014. ISBN 978-1-4799-5470-4. doi: 10.1109/EDOC.2014.38.

[6] J. Espinazo-Pagán, J. S. Cuadrado, and J. G. Molina. Morsa: A Scalable Approach for Persisting and Accessing Large Models. In J. Whittle, T. Clark, and T. Kühne, editors, *MODELS 2011: 14th International Conference on Model Driven Engineering Languages and Systems*, volume 6981 of *Lecture Notes in Computer Science*, pages 77–92. Springer, 2011. ISBN 978-3-642-24484-1. doi: 10.1007/978-3-642-24485-8_7.

[7] F. Fouquet, G. Nain, B. Morin, E. Daubert, O. Barais, N. Plouzeau, and J.-M. Jézéquel. An Eclipse Modelling Framework Alternative to Meet the Models@Runtime Requirements. In R. B. France, J. Kazmeier, R. Breu, and C. Atkinson, editors, *MODELS 2012: 15th International Conference on Model Driven Engineering Languages and Systems*, volume 7590 of *Lecture Notes in Computer Science*, pages 87–101. Springer, 2012. ISBN 978-3-642-33665-2. doi: 10.1007/978-3-642-33666-9_7.

[8] D. S. Kolovos, R. F. Paige, and F. Polack. The Grand Challenge of Scalability for Model Driven Engineering. In M. R. V. Chaudron, editor, *Models in Software Engineering, Workshops and Symposia at MODELS 2008*, volume 5421 of *Lecture Notes in Computer Science*, pages 48–53. Springer, 2009. ISBN 978-3-642-01647-9. doi: 10.1007/978-3-642-01648-6_5.

[9] D. S. Kolovos, L. M. Rose, N. D. Matragkas, R. F. Paige, E. Guerra, J. S. Cuadrado, J. de Lara, I. Ráth, D. Varró, M. Tisi, and J. Cabot. A research roadmap towards achieving scalability in model driven engineering. In D. D. Ruscio, D. S. Kolovos, and N. Matragkas, editors, *BigMDE 2013: 1st Workshop on Scalability in Model Driven Engineering*, page 2. ACM, 2013. ISBN 978-1-4503-2165-5. doi: 10.1145/2487766.2487768.

[10] P. Mohagheghi, W. Gilani, A. Stefanescu, and M. A. Fernández. An empirical study of the state of the practice and acceptance of model-driven engineering in four industrial cases. *Empirical Software Engineering*, 18(1):89–116, 2013. doi: 10.1007/s10664-012-9196-x.

[11] N. Nikolov, A. Rossini, and K. Kritikos. Integration of DSLs and Migration of Models: A Case Study in the Cloud Computing Domain. *Procedia Computer Science*, 68:53–66, 2015. ISSN 1877-0509. doi: 10.1016/j.procs.2015.09.223. 1st International Conference on Cloud Forward: From Distributed to Complete Computing.

[12] Object Management Group. *Unified Modeling Language Specification*, August 2011. [http://www.omg.org/spec/UML/2.4.1/](http://www.omg.org/spec/UML/2.4.1/).

[13] Object Management Group. *Object Constraint Language*, February 2014. [http://www.omg.org/spec/OCL/2.4/](http://www.omg.org/spec/OCL/2.4/).

[14] OSGi Alliance. OSGi Core. (6.0.0), 2014. [http://www.osgi.org/download/r6.osgi.core-6.0.0.pdf](http://www.osgi.org/download/r6.osgi.core-6.0.0.pdf).

[15] P. Östberg, H. Groenda, S. Wesner, J. Byrne, D. S. Nikolooulos, C. Sheridan, J. Krzywda, A. Ali-Eldin, J. Tordsson, E. Elmroth, C. Stier, K. Krogmann, J. Domaschka, C. B. Hauser, P. J. Byrne, S. Svorobej, B. McCollum, Z. C. Papazachos, D. Whigham, S. Ruth, and D. Paurevic. The CACTOS Vision of Context-Aware Cloud Topology Optimization and Simulation. In *CloudCom 2014: 6th IEEE International Conference on Cloud Computing Technology and Science*, pages 26–31. IEEE Computer Society, 2014. ISBN 978-1-4799-4093-6. doi: 10.1109/CloudCom.2014.62. URL [http://dx.doi.org/10.1109/CloudCom.2014.62](http://dx.doi.org/10.1109/CloudCom.2014.62).

[16] A. Pantaleev and A. Rountev. Identifying Data Transfer Objects in EJB Applications. In *WODA@ICSE 2007: 5th International Workshop on Dynamic Analysis*, page 5. IEEE Computer Society, 2007. ISBN 0-7695-2963-1. doi: 10.1109/WODA.2007.6.

[17] J. D. Rocco, D. D. Ruscio, L. Iovino, and A. Pierantonio. Collaborative Repositories in Model-Driven Engineering. *IEEE Software*, 32(3):28–34, 2015. doi: 10.1109/MS.2015.61.

[18] A. Rossini. Cloud Application Modelling and Execution Language (CAMEL) and the PaaSage Workflow. In A. Celestí and P. Leitner, editors, *Advances in Service-Oriented and Cloud Computing—Workshops of ESOCC 2015*, volume 567 of *Communications in Computer and Information Science*, pages 437–439. Springer, 2016. ISBN 978-3-319-33313-7. doi: 10.1007/978-3-319-33313-7.

[19] A. Rossini. Cloud Application Modelling and Execution Language (CAMEL) and the PaaSage Workflow. In A. Celestí and P. Leitner, editors, *Advances in Service-Oriented and Cloud Computing—Workshops of ESOCC 2015*, volume 567 of *Communications in Computer and Information Science*, pages 437–439. Springer, 2016. ISBN 978-3-319-33313-7. doi: 10.1007/978-3-319-33313-7.

[20] A. Rossini, A. Rutle, Y. Lamo, and U. Wolter. A formalisation of the copy-modify-merge approach to version control in MDE. *Journal of Logic and Algebraic Programming*, 79(7): 636–658, 2010. doi: 10.1016/j.jlap.2009.10.003.

[21] A. Rossini, K. Kritikos, N. Nikolov, J. Domaschka, F. Griesinger, D. Seybold, and D. Romero. D2.1.3—CAMEL Documentation. PaaSage project deliverable, October 2015.