Evaluating the Cloud Architecture of AToMPM

Jonathan Corley[1], Eugene Syriani[2] and Huseyin Ergin[1]
[1]*Department of Computer Science, University of Alabama, Tuscaloosa, AL, U.S.A.*
[2]*Department of Computer Science, University of Montreal, Montreal, Canada*
corle001@crimson.ua.edu, syriani@iro.umontreal.ca, hergin@crimson.ua.edu

Keywords: Multi-view, Multi-user, Collaboration, Architecture, Volume Testing, Stress Testing.

Abstract: In model-driven engineering, stakeholders work on models in order to design, transform, simulate, and analyze systems. Complex systems typically involve many stakeholder groups working in a coordinated manner on different aspects of a system. Therefore, there is a need for collaborative platforms to allow modelers to work together. Previously, we introduced the cloud-based multi-user tool AToMPM, designed to address the challenges for building a collaborative platform for modeling. This paper presents on the multi-user, multi-view architecture of AToMPM and an initial evaluation of its performance and scalability.

1 INTRODUCTION

Complex systems engineering typically involves many stakeholder groups working in a coordinated manner on different aspects of a system. In a model-driven engineering (MDE) context, engineers express their models in different domain-specific languages (DSLs) to work with abstractions expressed in domain-specific terms (Combemale et al., 2014). Recently, there has been a growing trend toward collaborative environments especially those utilizing browser-based interfaces (e.g., Google Docs or Trello[1]). Additionally, this trend can be seen in software development tools, such as Eclipse Che[2]. Cloud-based technology to support collaboration has also started to gain interest in the MDE community (Paige et al., 2014). Example tools are AToMPM (Syriani et al., 2013), cloud-based multi-user web interface for modeling and transformations, and WebGME (Maróti et al., 2014), a web-based collaborative modeling environment.

These collaborative environments bring new concerns. Modeling tools, frameworks, and language workbenches typically consider all developed artifacts as models. Although users believe they are working on distinct models, there is a single underlying model (Atkinson and Stoll, 2008) that ensures consistency among the users. Users effectively operate on a view, projection of the model, that shows a portion of the model in its own concrete syntax representation.

This paper focuses on the multi-user capabilities of AToMPM. In previous work (Corley et al., 2016), we presented in detail the four collaboration scenarios that the multi-view modeling enhancement of AToMPM supports: (1) two users are working on the exact same artifact simultaneously; (2) two users are working on different parts of the same artifact; (3) two users with different expertise are working on distinct artifacts that, together, compose the overall system; and (4) one user is working on a view that projects two models, while another user is working on a view of one of these models. These form the functional requirements of our architecture. In contrast, this paper focuses on the non-functional requirements of the architecture described in Section 2.

In order to ensure consistency and synchronization among the artifacts produced by each stakeholder, we favor a cloud-based environment. In previous work (Corley and Syriani, 2014), we presented an early prototype of the multi-view modeling architecture of AToMPM. In this paper, we present an updated version of the architecture that provides modeling as a service (i.e., multi-user, multi-view modeling storage and processing capability) to any client able to maintain a connection with the system (e.g., browser-based clients, mobile and tablet clients, and traditional desktop clients). Moreover, the primary contribution of this paper is an evaluation of this architecture with regards to performance as the number of users increases, described in Section 4.

[1]docs.google.com, www.trello.com
[2]https://eclipse.org/che/

2 NON-FUNCTIONAL REQUIREMENTS

The primary guiding concern driving this work is to create a system that can support collaboration and modeling where it is most natural to the users.

2.1 Responsiveness

A primary concern for our architecture is to provide an acceptable level of responsiveness. Most operations on models in AToMPM, such as model transformations and larger distributed processes and storage, are based on CRUD (create, read, update, delete) operations, it is imperative to ensure these operations are processed in a minimal amount of time. A user across a session will make many individual CRUD operations, thus these operations should not present a notable delay in the users process.

2.1.1 Models and Views in AToMPM

Two modelers working within a shared artifact to collaborate on a larger project may not always share the same focus, working with subsets of the same model. Alternatively, modelers may desire to visualize a model for varying purposes. AToMPM utilizes views to provide each modeler with the capability to refine the scope and style of visualization when working on shared artifacts. A specific view may contain only a portion of a model or utilize a distinct visualization.

In AToMPM, models are separated into two categories: abstract models and views. This distinction allows modelers to work with portions of a model and use distinct representation, arrangement, and sizing for the elements of their view; AToMPM exclusively supports a graphical syntax for models. Thus, stakeholders are able to share models, allowing modelers with varying skills to work collaboratively on the same model. An abstract model is the abstract syntax of a model conforming to the metamodel of a given DSL. Conceptually, a view is a projection of an abstract model onto a DSL that uses the most appropriate representation of a subset of the model's elements for the needs of the expert modeler working on that part of the model. A view references a set of elements from the abstract model. The view also maintains a list of other models necessary to use the specific view, including a concrete syntax model. By varying the concrete syntax between views, different users may view the same model in their own notations (e.g., European vs. American notations for electrical circuit diagrams). Finally, the view includes a mapping of model elements to concrete syntax information, since these mappings are specific to a view of the model. These mappings might include the absolute position, rotation angle, and size scaling of the element.

2.2 Managing Conflicting Requests

As the number of concurrent users accessing a shared resource increases, conflicting requests become inevitable. Managing conflicting requests is a primary concern for systems enabling shared access to resources. However, the architecture is designed to process requests based on order of arrival: every request will either succeed or fail based on the conditions of the model at the time it is processed. We chose this opportunistic method of conflict management to allow automatic resolution of all requests and improve responsiveness. The architecture focuses upon resolving all messages with minimal processing to preserve responsiveness. This focus on responsiveness also works to reduce the possibility of conflicting requests. Decreasing time spent processing and providing updates to the user more quickly both decrease the window in which users may provide conflicting requests. If the time to process a request and then update all clients is one second, then a conflicting request must occur within one second of the prior request.

However, the client systems may still detect failures or even lost updates (a scenario where a second update overwrites a prior update before the second user is made aware of the first update), because the architecture provides continuous updates to all relevant users as each request is made by any user. When a user creates an element, all users receive an update stating the element has been created. The client system may take advantage of this stream of updates to identify conflicting requests and provide more advanced and costly methods of conflict management, but we did not implement any yet. In case of conflicts that cannot be automatically resolved, the client system enables users to interact with each other directly via an instant chat window, so they can manage such conflicts themselves.

3 MODELING AS A SERVICE

In this section, we present the architecture of AToMPM. This architecture has been designed to resolve the challenges of providing an efficient multi-user, multi-view modeling environment. The components of this architecture are client systems, Modelverse Kernel (MvK), and a set of controllers coordinating the other two components. The three com-

ponents communicate by exchanging change logs encoded in JSON that contain all the necessary information to perform a task, or undo it. The design provides modeling as a service through a distributed network of controllers managing the concerns of a multi-user, multi-view modeling system, being independent from the implementation of the client.

The Modelverse (Van Mierlo et al., 2014) is a centralized repository to store and manipulate models and provides an API, the MvK, that processes all modeling actions; e.g., primitive requests on and execution of models that are stored in the Modelverse. AToMPM includes a default web client, but the underlying architecture supports a wide variety of clients by exposing a simple API with connections and transmissions made using 0MQ[3], an open source socket messaging library with support for a large variety of languages and platforms.

3.1 Controllers

The controllers of the architecture encompass a series of distinct controllers that ensure consistency among clients, view(s), and model(s). A user can perform CRUD operations, load a model, add or remove elements from a view, execute a model or set of models, and more. These requests are sent into the system through a message forwarder that routes messages to the proper controller for processing. Reciprocally, a response forwarder returns responses to clients. This pattern enables clients and controllers to subscribe to messages regarding any number of specific views of models without needing to connect to the dynamically spawned controllers that manages the relevant views and abstract models.

3.1.1 Model Controller

The model controller provides a layer above the MvK that incorporates multi-user, multi-view modeling. A model controller manages a given model and all views of that model, and there exists one model controller per active abstract model. In this way, the processing of concurrent user requests is distributed based on the artifacy being used. However, all requests for a given model are managed by a single controller ensuring consistency for the model and the associated views. Whenever a model operation alters a portion of the abstract model, the resulting changelog is published to all that have subscribed to a view that references the updated portion of the model. The view operations are guaranteed to have no side effect on the model and thus do not need to update users of other views. Load and read requests are performed directly at the model controller, which are resolved locally using cached information. For all other operations, the model controller coordinates the processing of primitive operations within the MvK. Model controllers ensure that a sequence of primitive operations is valid and separates models from views, whereas they are treated uniformly within the MvK.

To enable automatic resolution of conflicting messages, the model controller guarantees one message must complete successfully (win) and other conflicting messages will fail. Incoming client changelogs are queued to be processed using a FIFO strategy. Thus, operations that originate from clients sending a changelog may fail based on the results of a previous message. By allowing the earlier message to win, the system does not need to process any rollback operations due to conflicting messages. This scenario of conflicting messages is mitigated in practice by the publishing update messages to ensure all users maintain a consistent copy of the model. Thus, even if multiple users send conflicting messages simultaneously, the users will be notified of the resolution for all messages. Furthermore, because users are notified of all updates immediately, the likelihood of generating conflicting messages is minimized.

3.1.2 Supercontroller

Model controllers manage all model and view related operations. For each model, there exists exactly one model controller. However, the system may have any number of models. If the system maintained these model controllers at all times, a significant amount of resources would be wasted on model controllers that are not in use. To mitigate this issue, we spawn model controllers as needed on a distinct processor. Controlling which model controllers are active and assigning the model controllers to resources is the primary responsibility of the supercontroller. The supercontroller ensures there exists at most a single model controller for a given abstract model and its related view(s), and it monitors all client changelogs entering the system. When a client changelog is encountered that requires a model controller not yet spawned, the supercontroller manages spawning the model controller. Additionally, the supercontroller ensures all client changelogs received during the spawning process are queued and forwarded after the model controller is sucessfully spawned. Any client changelog intended for an active model controller is ignored by the supercontroller.

[3]http://zeromq.org/

3.1.3 Node Controllers

When a model controller is spawned, it is assigned to a node controller. Node controllers are abstract representations of available hardware computing resources (e.g., nodes in a cluster, CPUs, cores), and provide an interface to these resources for the supercontroller.

4 EVALUATION

This section presents an empirical evaluation of the architecture, focusing on the Controllers, targeting two key concerns: communication time for transmitting requests and processing time for handling requests transmitted. AToMPM is evaluated through a series of experiments where the size of models and number of concurrent users are varied independently. These experiments exercise the system in a wide variety of expected conditions including worst case scenarios to evaluate the performance of the system. This serves as volume and stress testing of the MVC architecture of AToMPM. We conducted three experiments to answer the following research questions: (**RQ1**) How does the architecture scale regarding the number of concurrent users? (**RQ2**) How effective is the architecture in responding to a user request as the size of the model increases?

4.1 Setup

The supercontroller was running on a dual-core machine (3.33 GHz, 4GB RAM) and the MvK on a quad-core machine (3.1 GHz, 8GB RAM). Two dual-core machines (3.33 GHz, 4GB RAM) and a quad-core machine (2.4 GHz, 8Gb RAM) were simulating users to have true concurrency. The node controllers were running on three dual-core machines (3.33 GHz, 4GB RAM) and a quad-core machine (3.1 GHz, 8GB RAM).

This resulted in a load of 50 users and 50 model controllers per machine for each experiment. The experiment setup preloaded all users, model controllers, and models in the MvK before measuring. This warmup phase put the system in an average expected case; i.e., how the system would perform after initial requests for each model controller. The startup time for a model controller is significant at approximately 2-3 seconds to spawn its process and some additional time spent building the cache of elements for the model and initial view. For all experiments, we used the same basic Petri net model with varying numbers of places (duplicated when necessary to have multiple models). Note that all views referenced every element of the associated abstract model. Thus, we measured the worst case scenario for performance.

4.2 Experiments

Each of the following experiments varied the number of concurrent users and model size independently. The variable for number of concurrent users is $U = 1, 25, \ldots, 200$ with increments of 25. The variable for model size is $N = 1, 50, \ldots, 200$ with increments of 50. Here N represents the number of places (i.e., elements) in the Petri nets model. We conducted three experiments. Each experiment varied the resources shared by the users. However, all users send a series of load, create, and delete requests. Load is a read all request sent to obtain an initial version of the model. Create is the most expensive single element operation. Delete is the least expensive single element operation. The load requests have a list of N dictionaries for a model of size N with each dictionary containing the relevant information for a specific model element. Each request is repeated 10 times by each user to minimize random machine and network effects on transmission time. Furthermore, all transmission were performed on a LAN to minimize network latency and routing effects on transmission time. In **Experiment 1**, each user connects to the same view of the same model. In **Experiment 2**, each user connects to a distinct view of the same model. In **Experiment 3**, each user connects to a distinct view of a distinct model.

Experiments 1 and 2 are designed to simulate collaboration scenarios 1 and 2. However, experiment 3 is not a precise simulation of scenario 3. Experiment 3 simulates multiples users accessing distinct models (where each model has its own view), but the models are not related, as described in collaboration scenario 3. Experiment 3 presents a near-simulation of the scenario to provide an initial evaluation of the scaling as the number of concurrent users increases. Designing a set of related models scaling from 1-200 models in the set is not feasible and would likely present additional factors impacting scaling. These experiments are designed primarily to focus on the performance scaling of the system as the number of concurrent users is increased and how the specifics of a collaboration scenario impact the performance scaling.

4.3 Results

Before discussing the three experiments, we evaluate the Controllers in a best case scenario without network overhead in the communication with the MvK and with a single user accessing an existing model controller (i.e., the model controller had already been

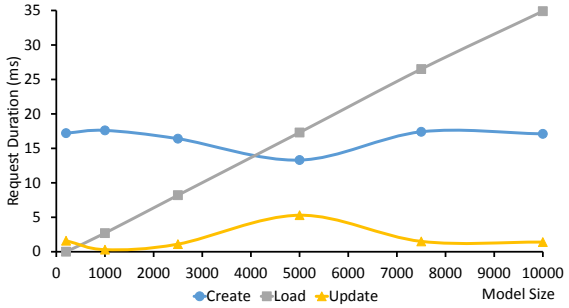


Figure 1: Create, Load, and Delete requests on a model controller with local MvK.

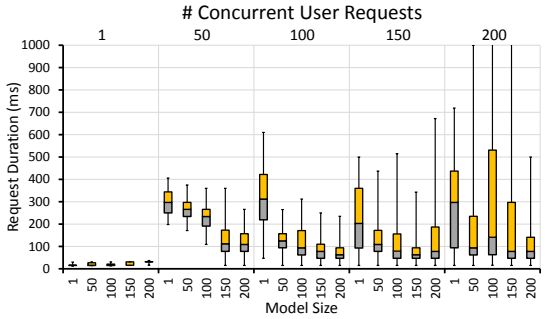
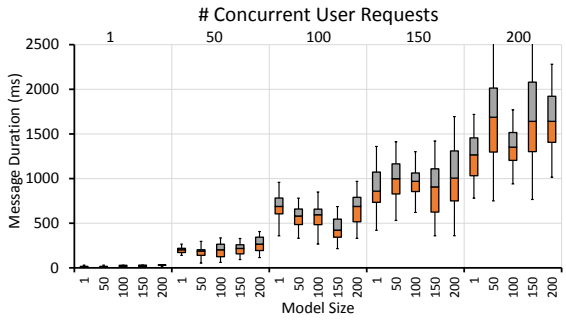
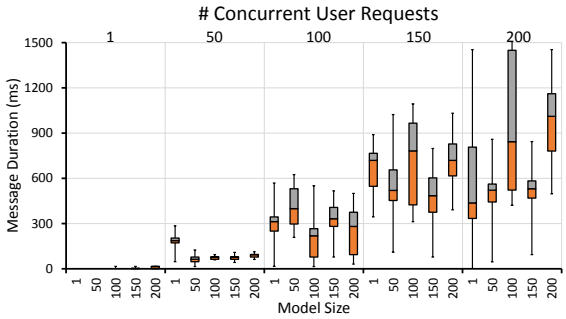


Figure 2: Results for experiment 1: Create Request.



(a) Create request



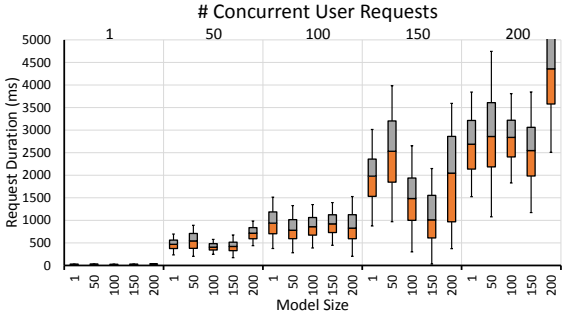
(b) Load request

Figure 3: Results for experiment 2.

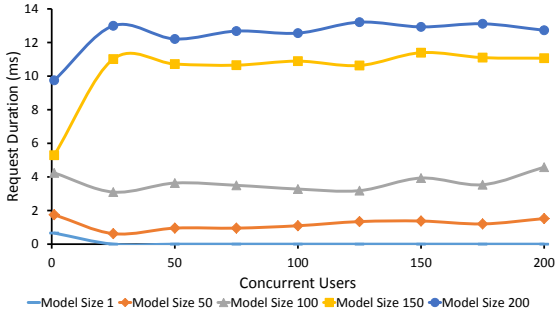
spawned and loaded its local caches before the user request). The results of the best case scenario (presented in Figure 1) establishes a baseline of processing capability for the system. The time to load a view and the related model pair depends on the size of the model. We measured approximately 3-4ms per 1,000 elements loaded, but this time will vary slightly depending on the performance capability of the computing resources (we used a machine with 3.33 GHz processor speed and 4GB total RAM). Both create and delete requests took a relatively constant amount of time regardless of model size. Minor variances were noted, but with variances from the norm being no more than 3-5ms these variances can be attributed to random variance in system performance. Create requests are not affected by the size of the model, but are slower than delete requests. The time to perform a load request is linear with respect to the size of the model, because load requires accessing each element in the model. Also, unlike the other three operations, load bypasses the MvK using a cache in the model controller.

Experiment 1. demonstrates an approximately constant response time except at 1 user for all operations. We only show the create operations in Figure 2 since delete and load performed similarly to create. Smaller model sizes have a lot of variance, because the network communication is predominant. The most important result of experiment 1 is the large ranges for most runs. All runs contain approximately 1,500 data points which are typically uniformly distributed over a wide standard deviation. The average coefficient of variance (CoV) across all runs was 63.5% and the median CoV was 46.6%. The large variation is due to the fact that all messages are being sent at approximately the same time, similar to a distributed denial-of-service. This results in messages being queued at the model controller and MvK which must be processed sequentially. The variance grows more significant as the number of users increases: i.e., the number of messages received concurrently increase. Experiment 1 and 2 share this quality due to their design: all users are on the same abstract model, thus both experiments will be processed by a single model controller. Nevertheless, all clients receive their response within less than a second. We expect relatively small numbers of users (e.g., less than 10) per Model Controller. Thus, we have designed for consistency when multiple users access the same model, but we have not focused on distributing the processing within a model controller to efficiently manage large numbers of users concurrently collaborating on a single model.

Experiment 2. demonstrates a linear increase as the number of users increases, but again model size has little to no impact (depicted in Figure 3). Load operations, which ignore the MvK, scale similarly albeit



(a) Create request



(b) Load request

Figure 4: Results for experiment 3.

slightly improved compared to the create requests. Thus, the bottleneck lies in how the model controller is managing multiple concurrent views.

Experiment 3. is designed to test the scalability in the expected case where users are more evenly distributed across model controllers. In Figure 4, load is constant with respect to the number of concurrent users. However, the other operations perform slowly, taking at least a second. The main reason for that is the current implementation of the MvK. Although the Controllers are a distributed system, the MvK queues all request and processes them sequentially. We expect improved performance of these operations once the MvK is moved to a distributed design.

4.4 Discussion

In the following, we discuss the implications of our results with regard to our two identified research questions.

RQ1. Experiment 1 for all request types and experiment 3 for load demonstrate promising scaling with regards to increased number of concurrent users. However, experiment 2 identifies that many users accessing distinct views poses a concern for scalability with performance at $U = 200$ approaching 2 seconds for a create request. Nevertheless, the most significant concern is the limitation of the current MvK as demonstrated by comparing the create and load results for experiment 3. Create scales linearly reaching response times in excess of 5 seconds while load requests do not scale with number of users in experiment 3. The results indicate a strong need for distributed processing in the MvK, and strong promise for the overall performance in experiment 3; i.e., our expected average case. Overall, we feel the system performs well with obvious deficiencies when many users ($U > 50$) employ distinct views of the same system. Also, the system is unable to leverage the full potential of the model controllers due to the current performance bottle neck of the MvK. In conclusion, the Controllers portion of the architecture can efficiently handle multiple users sending concurrent requests, but unusually large numbers of users on one or more views of a model will adversely affect scalability.

RQ2. Our results for the main experiments do not indicate scaling as model size varies. However, due to limitation of the MvK, we were unable to test reasonably large models during the main experiments. Currently, the MvK stores active models within memory and fails to swap out models as the limits of RAM are reached. Thus, when we perform an experiment with many users (up to 200 in these experiements) accessing distinct models and views of significant size (i.e., moderate and large scale models), the system will crash due to reaching memory bounds. Due to the limitations of the MvK, we refer to the results in our best case scenario (i.e., using a single model controller with a localhost connection to the MvK) as presented in Figure 1. These results demonstrate that create and delete requests are not influenced by the size of the model. This is expected since create and delete requests target a single element and are not influenced by model size. However, load is affected by the model size, because it reads all the elements in both the abstract model and the view. In conclusion, we find that the system scales as expected with regards to model size, but further work is needed to investigate potential interactions of the variables N and U (i.e., repeating the main experiments with large scale models to ensure our results hold true as the number of concurrent users increases).

4.5 Threats to Validity

The first threat to validity regards the metamodel. The performance results have only been measured on similar Petri net models. Different formalisms may influence the results, in particular those exposing complex inter-object relationships or heavy objects: i.e., many attributes or large attributes. Another threat to valid-

ity is that we have not measured results in scenarios using large scale models. Our results demonstrate the performance scaling as size of the model is increased, but do not demonstrate the interaction of simultaneously scaling the model size and concurrent number of users. We were unable to test this scenario due to limitations at the MvK level. Thus, we must leave investigating the interaction of these two variables for future work. We identify a threat to validity regarding the experimental setup. The method for distributing controllers among available machines has an influence on the overall performance. We assume an ideal distribution of model controllers among machines. In practice, the distribution of these controllers may not be ideal, but the exact distribution depends on the specfics of the load balancing algorithm and factors specific to a given scenario. Depending on the load balancing strategy these factors might include order of spawn for model controllers, size of model/views on a given model controller, or frequency of requests to a given model controller. The experiments presented here provide an initial evaluation identifying promisiong areas and areas in need of further improvement in the current architecture. We leave evaluating the complex task of load balancing in our architecture to future work.

5 RELATED WORK

We only discuss related work focusing on multi-user modeling environments. Nevertheless, there are also related work in web diagramming environments[4], distributed databases (Özsu and Valduriez, 2011), and distributed simulation (Zhang et al., 2010).

Modern modeling tools are primarily native desktop applications, e.g., MetaEdit+ (Kelly et al., 1996). AToMPM is one of the first web-based collaboration tool for MDE. It takes advantage of the ever increasing capabilities of web technologies to provide a purely in-browser interface for multi-paradigm modeling activities. Nevertheless, Clooca (Hiya et al., 2013) is a web-based modeling environment that lets the user create DSLs and code generators. However, it does not offer any collaboration support.

Recently, Maroti et al. proposed WebGME, a web-based collaborative modeling version of GME (Maróti et al., 2014). It offers a collaboration where each user can share the same model and work on it. In contrast with the Modelverse, WebGME relies on a branching scheme (similar to the GIT version control system) to manage the actions of different users on the same model. Thus, WebGME supports scenarios 1 and 2, but not in an always-online environment, unlike in AToMPM where operations immediately resolve.

The GEMOC Initiative (Combemale et al., 2014) produced GEMOC Studio, a modeling system within the Eclipse ecosystem. Users compose multiple DSLs into a cohesive system with well-defined interaction of the various models (Combemale et al., 2013). GEMOC Studio provides a solution for scenario 3 similar to the Controllers, but does not handle concurrent users.

Gallardo et al. proposed a 3-phase framework to create collaborative modeling tools based on Eclipse (Gallardo et al., 2012). The difference between creating a regular DSL and a collaborative modeling tool in their system is the addition of the technological framework, adding collaboration support to the DSL. Naming it "TurnTakingTool", multiple users are able to modify an existing model by utilizing a turn-based system. The framework helps the user to create the DSL as a native eclipse plug-in with concurrency controls, graphical syntax and multiple user support.

Basciani et al. proposed MDEForge (Basciani et al., 2014), a web-based modeling platform. MDEForge offers a set of services to the end user by a rest api, including transformation, model, metamodel editing. Although, the idea seems elegant, the system was not ready at the time of writing this paper. The authors, also, mention adding the collaborative capabilities of the platform in the future.

6 CONCLUSION

In this paper, we presented the cloud architecture of AToMPM providing modeling as a service and designed to support collaborative modeling scenarios. We presented a preliminary evaluation focusing on how scaling the number of concurrent users impacts the response time for create, delete, or load requests. The results reveal that the controllers in the architecture are efficient with regards to number of concurrent users, but the MvK currently presents a bottleneck. We plan to explore a strategy supporting distributed processing and storage for low-level modeling operations within the MvK. and then investigate load balancing strategies to manage distributing processing and storage among distributed nodes with distinct performance and storage capabilities.

[4]https://cacoo.com/

REFERENCES

Atkinson, C. and Stoll, D. (2008). Orthographic Modeling Environment. In *Fundamental Approaches to Software Engineering*, volume 4961 of *LNCS*, pages 93–96. Springer.

Basciani, F. et al. (2014). MDEForge: an extensible Web-based modeling platform. In *(Paige et al. 2014)*, pages 66–75.

Combemale, B. et al. (2013). Reifying Concurrency for Executable Metamodeling. In *Software Language Engineering*, volume 8225 of *LNCS*, pages 365–384. Springer.

Combemale, B. et al. (2014). Globalizing Modeling Languages. *Computer*, pages 68–71.

Corley, J. et al. (2016). Cloud-based Multi-View Modeling Environments. In *Modern Software Engineering Methodologies for Mobile and Cloud Environments*. IGI Global.

Corley, J. and Syriani, E. (2014). A Cloud Architecture for an Extensible Multi-Paradigm Modeling Environment. In *MODELS 2014 Poster Session and the ACM SRC*, pages 6–10.

Gallardo, J., Bravo, C., and Redondo, M. A. (2012). A model-driven development method for collaborative modeling tools. *Journal of Network and Computer Applications*, 35(3):1086–1105.

Hiya, S., Hisazumi, K., Fukuda, A., and Nakanishi, T. (2013). clooca : Web based tool for Domain Specific Modeling. In *MODELS'13: Invited Talks, Demos, Posters, and ACM SRC*, volume 1115, pages 31–35. CEUR-WS.org.

Kelly, S., Lyytinen, K., and Rossi, M. (1996). MetaEdit+ A fully configurable multi-user and multi-tool CASE and CAME environment. In *Conference on Advanced Information Systems Engineering*, volume 1080 of *LNCS*, pages 1–21. Springer.

Maróti, M. et al. (2014). Next Generation (Meta)Modeling: Web- and Cloud-based Collaborative Tool Infrastructure. In *Multi-Paradigm Modeling*, volume 1237, pages 41–60. CEUR-WS.org.

Özsu, M. T. and Valduriez, P. (2011). *Principles of distributed database systems*. Springer Science & Business Media.

Paige, R. F. et al., editors (2014). *Proceedings of the 2nd International Workshop on Model-Driven Engineering on and for the Cloud*, volume 1242. CEUR-WS.org.

Syriani, E. et al. (2013). AToMPM: A Web-based Modeling Environment. In *MODELS'13: Invited Talks, Demos, Posters, and ACM SRC*, volume 1115. CEUR-WS.org.

Van Mierlo, S., Barroca, B., Vangheluwe, H., Syriani, E., and Kühne, T. (2014). Multi-Level Modelling in the Modelverse. In *Multi-Level Modelling*, volume 1286, pages 83–92. CEUR-WS.org.

Zhang, H. et al. (2010). A model-driven approach to multidisciplinary collaborative simulation for virtual product development. *Advanced Engineering Informatics*, 24(2):167 – 179. Enabling Technologies for Collaborative Design.