ShareLatex on the Edge: Evaluation of the Hybrid Core/Edge Deployment of a Microservices-based Application

Genc Tato, Marin Bertier, Etienne Rivière, Cédric Tedeschi

ShareLatex on the Edge: Evaluation of the Hybrid Core/Edge Deployment of a Microservices-based Application

Genc Tato*, Marin Bertier*, Etienne Rivière° and Cédric Tedeschi*
* Univ Rennes, Inria, CNRS, IRISA, France ○ UCLouvain, Belgium

ABSTRACT

Collaborative web applications benefit from good responsiveness. This can be difficult to achieve with deployments on core data centers subject to high network latencies. Hybrid deployments using a mix of core and edge resources closer to end users are a promising alternative. Many challenges are associated with hybrid deployments of applications, starting from their decomposition into components able to be replicated dynamically onto edge resources to the management and consistency of these components' state.

We report on our experience with the hybrid deployment of ShareLatex, a legacy collaborative web application. We show how its design based on the use of microservices and resource-oriented APIs allow for an efficient modular decomposition. We detail how we adapted the application configuration for a hybrid deployment with no modification to its source code. Our experiments using a fleet of emulated users show that the use of a hybrid deployment for this legacy collaborative application can decrease user-perceived application latencies for common operations at the cost of increasing them for operations involving core/edge coordination traffic.

1 INTRODUCTION

The demand for low-latency internet services generates great interest towards using edge resources as a complement to traditional "core" cloud resources in data centers. Edge computing resources may offer more limited capacities or reliability guarantees than core resources, but are geographically spread closer to the end users, providing smaller network latencies [12].

A great diversity of applications can benefit from *hybrid* core/edge deployments [9, 18]. We are interested in this paper in collaborative edition environments, web-based applications that allow geographically distributed users to concurrently edit a document. Examples include SaaS applications such as Google Documents, Microsoft Word online, Nuclino and ShareLatex. User-perceived latencies are very important for these applications. Low delays between the action of a user and the visibility of its result by other users reduce the risk of conflicts and improve the global user experience.

While edge servers have been used to improve latencies by offloading localized computation or caching static content [20], the impact of core/edge deployments for dynamic interactive applications, such as collaborative environments, is still relatively unclear [6], in particular for *legacy*, off-the-shelf applications.

The ability of a legacy collaborative application to be deployed over both core and edge resources strongly depends on its software architecture. Monolithic applications linking to a unique database are not a good fit for hybrid deployments, for the same reasons they are not easily scaled horizontally in a single datacenter. A hybrid core/edge deployment requires instead the application to have a modular architecture, in order to be able to move or clone some of its constituents from core to edge resources, based on performance, reliability and durability considerations. The design principle of *microservices* [16] has gained a strong momentum for building of large web applications in cloud environments. Under this model, the application is split in a collection of independent single-purpose services whose implementation is independent from that of other microservices. Each microservice may use a different solution for managing its state and be independently scaled horizontally. Interaction between services typically happen through resource-oriented HTTP APIs following REST principles [13].

Microservices-based applications naturally appear to be good candidates for hybrid core/edge deployments. For availability reasons, microservices that handle durable, long-term information should remain in the core, while other microservices can be delegated or cloned to the edge. Deciding which microservice(s) should, or should not, be deployed at the edge is not simple. User-perceived latencies are typically a result of multiple interactions between microservices, and delays between edge and core resources may accumulate and result in poorer performance than in the initial core deployment. Furthermore, while stateless microservices are easier to clone to the edge, stateful microservices require special care, as their state may need to be kept consistent across multiple copies, which can yield more costs than benefits.

Contributions. We report on our experience of porting a legacy collaborative application for a hybrid core/edge deployment, and present the lessons learnt in the process. Our target application is ShareLatex, an open-source collaborative editing tool for LaTeX documents. Our goal is to assess whether porting such an existing complex application for a hybrid deployment is feasible with appropriate configuration but without changes to its code.

Our contributions are the following. We discuss criteria for edge deployment of microservices, including aspects linked to state management and consistency, and apply these criteria to the ShareLatex application (section 3). We describe the mechanisms that support this hybrid deployment with no modification to the application source code (section 4). We detail our benchmarking toolset featuring simulated behaviors of users collectively editing LaTeX documents. We experimentally demonstrate the performance improvement in terms of user-perceived latencies for some operations on joint documents and the tradeoffs that exist for other operations due to the hybrid, multi-site resulting deployment (section 5).

2 RELATED WORK

An overview of the field of edge and fog computing can be found in a survey by Li et al. [12]. Our work targets the evaluation of the hybrid core/edge deployment of a *legacy* application. A complementary line of work proposes new programming and software engineering models for building new applications in edge/cloud environments. This includes component-based approaches such as Jolie [15], solutions based on eventually-consistent convergent data types (CRDTs) such as LASP [14] or development frameworks for stream processing such as Steel [17].

Fesehaye et al. [8] consider a deployment scenario in which an overlay of *cloudlets* (similar to the type of edge resources we consider in this paper) supports applications including file edition. This application is not interactive: Users download the file from its host cloudlet, edit it locally and re-upload it. The authors study the impact of document location on user-observed performance. This is complementary to the problem we consider, and we similarly highlight in our evaluation the impact of hosting part of the state of our target application on one edge resource or another.

Clinch et al. [7] evaluate the impact of using cloudlets on the *display appropriation* of mobile applications users, which depends on the responsiveness of the application. The study considers in particular a simple interactive game and evaluates users' perception of its responsiveness for different deployment models. The authors conclude that while there are lower latencies the user perception is not significantly impacted by the use of close-by cloudlet resources.

Báguena et al. [5] study the responsiveness of mobile apps under core, edge and hybrid core/edge deployments. Their model considers the limited resources and lower latency available at the edge. It proposes a split between services for hybrid placement. It does not consider however, the interaction patterns between these services and the impact they may have on the performance of the application, and it does not provide guidelines for splitting an existing application into services able to benefit from hybrid deployments.

Aderaldo et al. [4] discuss the evaluation of microservices-based applications in general, and note the lack of empirical study on the performance of these applications. Our study aims at filling this gap for the particular case of hybrid core/edge deployments. Sieve [19] also considers ShareLatex as a target application, for the problem of efficiently collecting monitoring information from microservices. The monitoring allowed by Sieve could be the basis for implementing future automated core/edge deployments policies.

3 BACKGROUND

We review microservices principles and define categories of stateful microservices based on their ability to be replicated to the edge. We then describe our use-case application, ShareLatex.

Microservices. Microservices [16] are an evolution of service-oriented architectures currently very popular for building large-scale cloud-based applications. They favor the use of a collection of small and feature-focused software entities over software monoliths. Microservices are advantageous for agile software development, as each service can be implemented, tested and evolved independently. Interaction between microservices typically happen through APIs following the resource-centric REST over HTTP [13].

While microservices principles where not designed specifically for edge deployments, they offer a number of advantages for this purpose. Due to the statelessness of REST, interactions between services support dynamic relocation. This feature is already leveraged for elastic scaling. In order to support hybrid core/edge deployments, it is desirable to support the creation of *replicas* of services to one (or several) of the edge sites. This ability primarily depends on whether the service is stateful (i.e. it maintains state across different calls) or stateless (i.e. it may maintain temporary state for an ongoing request, but no state is kept between requests). Stateless services are easily replicable. Replicated stateful services are however concerned with consistency, as simply creating a clone of their existing database to another site may lead to diverging and irreconcilable states. Fortunately, we can observe that in many cases replicating or splitting a stateful service and its database to the edge may only have a limited impact on application correctness. Identifying such services in a legacy application is actually key to deciding which can or cannot benefit from a deployment to the edge. We define four cases, illustrated by Figure 1.

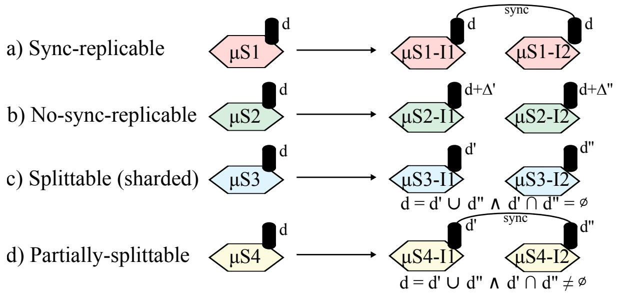

Figure 1: Four categories of microservices based on the impact of their replication on application correctness.

- *Sync-replicable* services require the full current state to be available on all instances of the service in order to maintain correct application behavior. They also require that this state be kept strongly consistent across replicas. Replicating such services generally leads to poor performance/cost tradeoffs, unless they have read-mostly accesses and are able to employ a strong consistency model that allows local reads, such as sequential consistency.
- *No-sync-replicable* services require a copy of the full service state for a new (edge) replica, but can preserve application behavior without enforcing strong consistency with the original (core) replica. The cost for replicating such services therefore only depends on the initial cost of copying the state from the core to the edge.
- *Splittable* services can provide semantically-equivalent service with only a *subset* of the original full service state. As partial states (shards) are disjoint between services instances, there is no need for consistency enforcement. The cost of replication in this case is the cost of splitting and outsourcing one of the shards to the edge.
- *Partially-splittable* services are a hybrid between sync-replicable and splittable services where only *part of the state* has to be shared and maintained strongly consistent between replicas. Replicating these services to the edge without falling back to the sync-replicable mode is however difficult. It typically requires modifying their implementation to redirect accesses to shared and disjoint states to different databases, or using geo-replicated databases that can leverage the partial split information [10, 11].

The *no-sync-replicable*, *splittable*, and *partially-splittable* assume a preferential attachment from users to a particular site, when accessing the service for a specific *resource*. It is necessary to deterministically redirect calls to this specific site. When there is no modification to the application code, this redirection has to rely on externally-visible information. This constraints is often met by microservices in the form of the URI resource name in the HTTP REST calls. This strengthens their potential for edge placement.

Use case application: ShareLatex. Our goal is to study the hybrid deployment of a legacy collaborative editing application, with no modification to its source code. We target an application based on microservices. We select ShareLatex, a collaborative tool popular in

Service	Description	Category	Cr.	LS	Fr.	Service	Description	Category	Cr.	LS	Fr.
1. docstore	CRUD ops on tex files	Splittable	✓		✓	8. track-changes	History of changes	Splittable		✓	✓
2. filestore	CRUD ops on binary files	Splittable	✓		✓	9. real-time	Websocket server	Stateless		✓	✓
3. clsi	Compile project	No-sync-replicable			✓	10. notifications	Notifications between users	Sync-replicable			
4. contacts	Manage contacts	Splittable				11. document-updater	Maintain consistent document state	Splittable		✓	✓
5. spelling	Spell checking	Splittable			✓	12. web	User interface and service hub	Sync-replicable	✓	✓	✓
6. chat	Chat service	Splittable			✓	Redis (db)	DB (Pub/Sub) for dynamic data				
7. tags	Folders, tags	Splittable				MongoDB (db)	DB for internal static data				

Table 1: ShareLatex services, their categories and additional attributes: critical(Cr.), latency-sensitive(LS), Frequent(Fr.)

research and teaching communities. It enables concurrent real-time modifications by various users on the same LaTeX document.

Collaborators in ShareLatex are likely to come from the same location. Typical examples could be a student and her supervisors, or researchers across local institutions working on a joint paper. Responsiveness is an important aspect for this application, as oftentimes editors can work collaboratively on the same parts of the document, and need to get updates and notifications from other users in near-real-time.

ShareLatex features 12 microservices and 2 databases, listed in Table 1. Their interactions are shown in Figure 2. We observe that the interaction is roughly centralized: the *web* service acts as hub for all other services and as the interface to the user (therefore playing the role of an API gateway). We also see that the *web*, *track-changes*, *document-updater* and *real-time* services are all interconnected through a *Redis* database, which is in charge of storing the real-time state of a document under edition (list of modifications, cursor positions, etc). This state is saved periodically in a long-term storage for documents with a call to the *docstore* service.

We should note that microservices principles are best practices in nature. They are not necessarily strictly and consistently enforced across entire applications. This is the case for ShareLatex. Service decoupling and state isolation are key principles in microservices, but we can observe that in this application the *Redis* service, acting as a container for the database of the same name, is used as a shared state for various other services. This inevitably affects the possibilities we have for replicating and deploying these services to the edge. Our goal is to study the impact of hybrid deployment for an unmodified, legacy application. We choose therefore to work with the application as it stands, and leave the evaluation of "pure microservices" applications to future work. We note that the lack of availability of such application for benchmarking has been pointed out by other authors before us [4].

4 HYBRID CORE/EDGE DEPLOYMENT

We seek to determine a *split* of the application, that is a set of replication possibilities for the microservices and a suggested *placement* on core and edge resources. We leverage the mapping into service categories defined in Section 3. Categories for all services are listed in Table 1. No service in ShareLatex is partially-splittable.

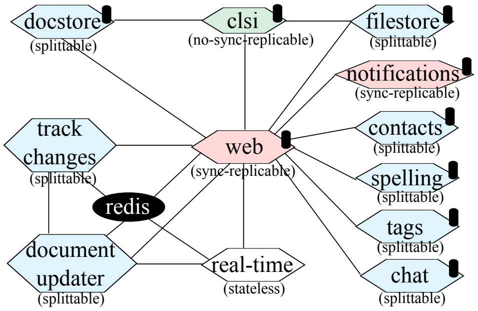

Figure 2: ShareLatex microservices architecture and their replication categories as defined in Figure 1.

Most ShareLatex microservices are splittable services. In particular, we observe that we can split their database either using the project identifier present in the REST calls URIs (services *document-updater*, *track-changes*, *docstore*, *filestore*, *chat*) or by the user identifier available in the same way (services *contacts*, *tags*, *spelling*). Two services, *web* and *notifications* are sync-replicable as their state cannot be replicated without full synchronization. The *clsi* service is no-sync-replicable: replicating it with no further consistency between replicas does not impact correctness. This service handles the compilation of LaTeX sources. For performance, it maintains a cache of the project elements including images and indexes, but loosing this state only result in these being fetched again from the *docstore* and *filestore* services.

All services could be potentially replicated from the core to one or more edge sites, but it arguably makes more sense for no-sync-replicable and splittable ones. There are however other non-functional aspects that must be considered for deciding on an appropriate placement. We identify three key aspects: *criticality*, *latency-sensitivity* and *frequency*. Critical microservices are essential for the resiliency and availability of the application. Data loss in this case must be avoided, requiring reliable hardware typically available only in the core, and this is where these services should stay. Services maintaining state but performing periodic checkpoints to another critical service can often be considered non-critical. Latency-sensitive services typically benefit highly from being replicated to the edge, especially when they can leverage the locality between users. Similarly, frequently-used services are more

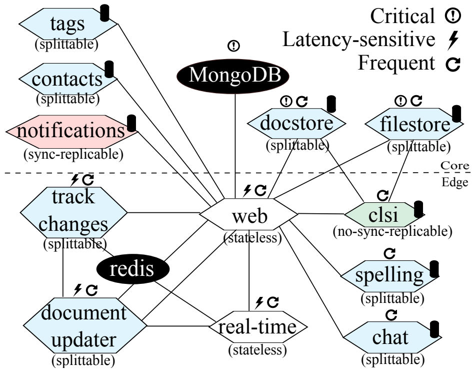

Figure 3: Our split and placement. Critical services in the core, latency-sensitive and frequent services at the edge.

likely to impact user experience, and this impact pays off more compared to the cost of enabling their replication.

Service replication categories suggest which services *may* to be deployed at the edge, and additional non-functional aspects indicate if they *should* be. Table 1 lists this information for all services.

We start by discussing a potentially contradicting case for the *web* microservice. This service has a sync-replicable state, meaning it is costly to replicate to the edge(s), but it is also latency-sensitive. Previously, we observed that it is closely coupled to other latency-sensitive services. From the service interaction graph, we can deduct that these services will benefit little from being deployed at the edge if they have to interact with the core, or we would have to implement strongly consistent replication defeating our objective of not modifying the services source code. Our suggested workaround is to decouple the state from the *web* service and host it separately from the service implementation. We deploy the *MongoDB* database of *web* in the core. The now stateless service can be deployed at the edge and benefit from low-latency interactions with other services through the *Redis* service. This is not an optimal solution as database accesses are now significantly less efficient. However, it is the only one which does not require any intervention on the application.

After considering the different service attributes and the workaround, we propose the split shown in Figure 3. We keep in the core services that are critical (*docstore*, *filestore*), sync-replicable (*notifications*) and infrequent (*tags*, *contacts*). We deploy at the edge latency-sensitive services (*web*, *document-updater*, *real-time*, *track-changes*) and frequent services (*spelling*, *clsi*, *chat*). Note that there remain instances of all services deployed to the edge in the core. These are not used by edge users but are useful for other users who connect directly to the core.

Implementing redirections. Splittable services now have instances on multiple edge sites. Each instance holds a disjoint subset of the original service state. Requests must deterministically reach the appropriate service instance. This instance can be in the core, or at one of the edge sites (not necessarily the closest to the client).

In ShareLatex, we create instances of splittable services storing the state of projects (currently edited documents). The instance storing the state for a specific project is selected at the site that is the closest to the user that creates this project. Note that we assume in this feasibility study that the location of a specific project (resource for this split service) is globally known on all core and edge sites, but remains unknown to the clients who simply connect to their closest edge site or to the core. We leave runtime service resolution and dynamic relocation for our immediate future work.

ShareLatex is an application designed for a single-site deployment. Services implementations are oblivious of any state split and do not implement the redirection of calls to instances on multiple sites. We can however implement this redirection at the level of REST calls between services, thanks to the resource-centric nature of REST and to the use of HTTP URIs for services endpoints. All user calls are intercepted by a reverse-proxy (`nginx` [3]) which redirects them to the local *web* service. The HTTP request to a splittable service contains a resource identifier that can be used to decide on a redirection to an edge location. Resources that are not subject to such a redirection are simply directed to the core by default. We modify the configuration of `nginx` to redirect calls to the correct site through a rule based on the project identifier: For any known mapping between a project identifier and some edge site, the call is redirected transparently.

5 EVALUATION

We evaluate the performance of the hybrid deployment of ShareLatex. We use 3 server-grade nodes from Grid'5000 [1], each with 2 Intel Xeon E5-2630 v3 CPUs and 128 GB of RAM. Our experiments do not focus on scalability or resource saturation. As a result, we use the same machines at low utilization rates for emulating core and edge sites as well as emulated users. We emulate WAN latencies between sites using the `tc` (traffic control) tool. We use a 50 ms roundtrip latency between the core and any edge site, and a 70 ms roundtrip latency between two edge sites. Users are considered as being close to one of the edge sites, to which they connect with a 5 ms roundtrip latency. We package the split ShareLatex application of Figure 3 and its support services (e.g. `nginx` and its configuration) as containers deployed using the Docker CE platform.

Emulating real users. We focus on user-visible application responsiveness. We are therefore interested in delays between actions by one user and the visibility by other users inside the actual web application running in a browser. This latency is a conjunction of multiple factors that cannot be fully modeled by measuring API-level latencies. We emulate a set of users using `Locust` [2], a load testing tool that allows programmatically describing the behavior of users as a list of actions, under specific scenarios and occurrence frequencies. Actions are HTTP or WebSocket requests. For example, opening a project requires an HTTP call to the project URL and establishing a Websocket connection. This connection is then used to write content to the project, which is another action which has a high frequency as it is the most common interactive one. `Locust` simulates a number of concurrent users by executing actions and scenarios in separate pseudo-threads (*greenlets*). Our user behavior is similar to the one used by Sieve [19].

To measure end-to-end user-perceived application latencies, we collect timestamps for each action start by some user, and corresponding events occurring on the application interface of other users (e.g. a cursor position change and its visibility).

Centralized vs. hybrid deployment. We start by comparing perceived latencies between a centralized deployment and a hybrid deployment using a single edge site. We use 5 concurrent emulated users who open and modify the same shared projects. Figure 4

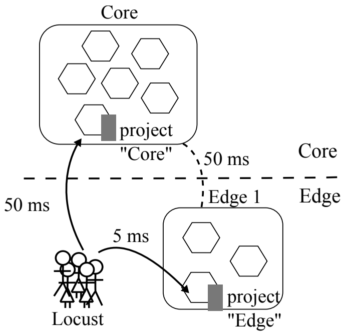

Figure 4: Centralized vs. hybrid deployment - exp. setup.

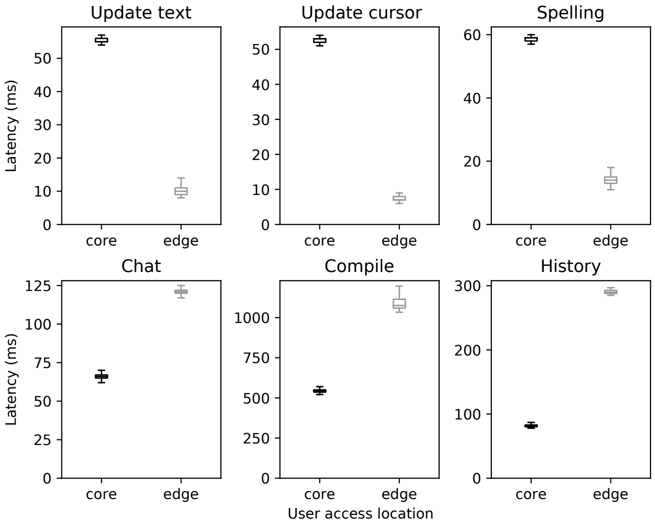

Figure 5: Hybrid deployment yields lower latencies for most frequent actions (first row).

shows the setup. Users first share a project fully hosted in the core. This represents our baseline. Measurements for a different set of actions are taken for a period of 10 minutes. Users then switch to working with another project hosted on the edge site, and run the same measurements for another 10 minutes.

We present results for the six following common actions: writing, moving the cursor, spell checking, sending chat messages, compiling the project and displaying the history of modifications. Figure 5 shows the distribution of perceived latencies for each of them. In the first row we observe that for common actions such as writing, cursor update and spell checking, users take advantage of reduced latencies when connecting to the project when hosted on the local edge site. These actions are indeed handled in our split by services (or services shards) that are all replicated to the edge. The other three actions, present in the second row, are negatively impacted by the hybrid deployment and the hosting of the project on the edge. The main services in charge of these actions are *chat*, *clsi* (for compiling) and *track-changes* (for history). These services have replicas running on the edge site in the hybrid deployment, but their interactions require communication with services instances remaining in the core, which results in additional latencies. For instance, although the document is compiled at the edge, the base document is retrieved first from the *docstore* service, which we kept in the core for availability reasons. In other cases, such as sending a chat message, the added latency comes due to the *web* service which needs to access its remote database to check if the user has the right permission to perform the task and does not cache this information. Our goal is to keep the application unmodified but implementing a fix would be relatively easy.

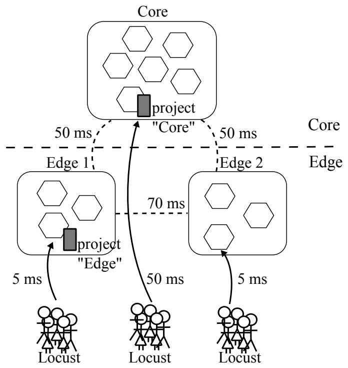

Figure 6: Impact of using redirections - exp. setup.

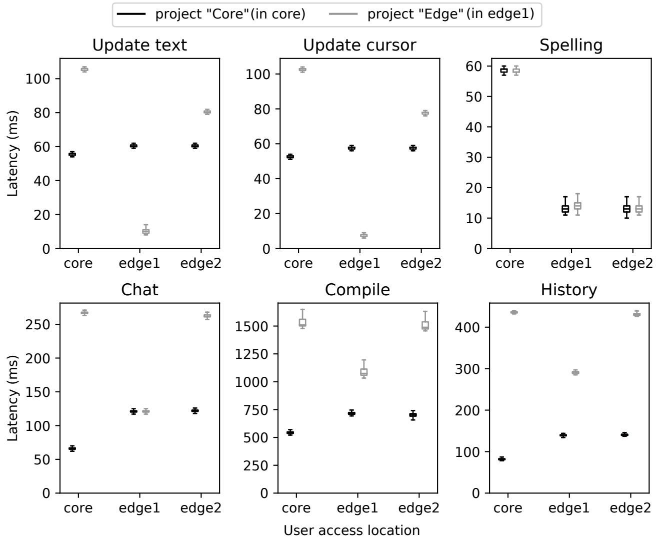

Figure 7: Impact of redirection and user access points.

Impact of redirections. In a second experiment, we evaluate the impact of using redirections when the project accessed by users is not managed on the site (core or edge) they are connecting to. We consider the scenario with two edge sites, Edge 1 and Edge 2, shown by Figure 6. We distinguish three user groups based on the site they connect to: Edge 1 users, Edge 2 users and users connecting directly to the core. Two separate projects are hosted on Edge 1 and in the core – we refer to them as project “Edge” and project “Core”. We emulate using Locust 5 users who simultaneously access and perform actions on the same project. These 5 users are in the same location, and we alternate between the 3 groups and 2 projects for 6 possible configurations.

Figure 7 shows the distributions of user-perceived latencies for the six operations we used previously, and for the six considered configurations. Perceived latencies for updating the text or the cursor position for the “Core” project are similar for all access locations, with a slight increase for users connecting through one of the edge sites. This small increase highlights the low cost of the unique redirection implemented by nginx. For the "Edge" project hosted on Edge 1, latencies for these two operations are greatly reduced for local accesses (when users connect to the Edge 1 site) while accesses through the other sites is impacted, mostly due to the addition of network latencies between sites, as the redirection operation itself remains of low cost. Observed latencies are in line with the emulated network configuration: 50 ms to the core and 50 ms more from the core to Edge 1 in the first case, or 5 ms to Edge 2 and 70 ms from Edge 2 to Edge 1 in the second case. Interestingly, we can observe that the *spelling* operation benefits from locality for users connected to both Edge 1 and Edge 2. The reason is that the *spelling* service does not depend on a particular project instance or identifier: there is no need to redirect the call to a particular site if an instance of the service is deployed locally.

The second row of Figure 7 presents latencies distributions for operations that always involve interactions with services instances in the core. The cost of redirecting some of the calls to services in the core is expected to have a negative impact when accessing a project located in the edge, as we have shown in our previous experiment. We observe indeed that these 3 actions on the "Core" project have a lower latency than the same actions on the "Edge" project. Furthermore, for these three actions applied to the "Edge" project, we do not observe the better performance for users connecting to Edge 2 compared to those connecting to the core, that we could observe for the actions of the top row. The reason for this is that the additional latency of redirection paid by users connecting to the core is evened out by the use of calls to local core services that follow. Users connecting to Edge 1 have better latencies for the "Edge" project as they avoid the price of redirection for the initial call to the service, but still do not perform as well as when the project is hosted directly in the core. The calls to the *chat* service yields the same delays whether or not the project is hosted on the edge or core site for users connecting to Edge 1: in both cases there is a redirection to the corresponding service in the core. For other groups of users, we observe the cumulative costs of the redirections to this same service.

In general, we observe from our experiments that the hybrid core/edge deployment results in a compromise between the gain of performance of operations that involve services that can be replicated (or split) to the edge site, and the loss of performance that is observed for operations requiring interaction with services staying in the core.

6 CONCLUSION AND FUTURE WORK

ShareLatex is a legacy application originally designed for single-site deployment but using a modular implementation based on microservices. We have described the lessons learnt in adapting this application configuration for a hybrid core/edge deployment. We identified that the state management and service localization are the two critical aspect for deciding on replication of a microservice to the edge, in addition to considerations on costs and reliability. These allow us to propose and evaluate a split of the application that only required changes to its configuration and runtime environment, but no modification to its source code. This was made possible, to a large extent, by the microservices-based design and the resource-oriented REST API used between these microservices and allowing redirections. Our evaluation has shown that some common operations benefit from hosting the project by services replicated to a close-by edge site, while others see a drop in performance due to the inevitable interactions with services that had to stay in the cloud for reliability and persistence reasons.

This study motivates our future work towards automated hybrid core/edge deployments. Automated deployment would require automatic service discovery and redirection at runtime with strong consistency requirements. Stateful service types from our taxonomy and additional criteria may be determined by the programmer, but runtime decisions about which service to replicate or move from one site to the other require appropriate measurements and decision-making algorithm, possibly powered by machine learning.

REFERENCES

[1] [n. d.]. Grid5000. https://www.grid5000.fr/.

[2] [n. d.]. Locust - An open source load testing tool. https://www.locust.io.

[3] [n. d.]. NGINX - Load Balancer, Web Server & Reverse Proxy. https://www.nginx.com/.

[4] Carlos M. Aderaldo, Nabor C. Mendonça, Claus Pahl, and Pooyan Jamshidi. 2017. Benchmark Requirements for Microservices Architecture Research. In *1st Intl. Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering (ECASE)*.

[5] M. Báguena, G. Samaras, A. Pamboris, M. L. Sichitiu, P. Pietzuch, and P. Manzoni. 2016. Towards enabling hyper-responsive mobile apps through network edge assistance. In *13th IEEE Annual Consumer Comm. Net. Conf. (CCNC)*.

[6] Junguk Cho, Karthikeyan Sundaresan, Rajesh Mahindra, Jacobus Van der Merwe, and Sampath Rangarajan. 2016. ACACIA: Context-aware Edge Computing for Continuous Interactive Applications over Mobile Networks. In *12th Intl. on Conference on Emerging Networking EXperiments and Technologies (CoNEXT)*.

[7] S. Clinch, J. Harkes, A. Friday, N. Davies, and M. Satyanarayanan. 2012. How close is close enough? Understanding the role of cloudlets in supporting display appropriation by mobile users. In *IEEE Intl. Conference on Pervasive Computing and Communications (PerCom)*.

[8] D. Fesehaye, Y. Gao, K. Nahrstedt, and G. Wang. 2012. Impact of Cloudlets on Interactive Mobile Cloud Applications. In *IEEE 16th Intl. Enterprise Distributed Object Computing Conference (EDOC)*.

[9] Pedro Garcia Lopez, Alberto Montresor, Dick Epema, Anwitaman Datta, Teruo Higashino, Adriana Iamnitchi, Marinho Barcellos, Pascal Felber, and Etienne Riviere. 2015. Edge-centric Computing: Vision and Challenges. *SIGCOMM Comput. Commun. Rev.* 45, 5 (Sept. 2015).

[10] Raluca Halalai, Pierre Sutra, Etienne Rivière, and Pascal Felber. 2014. ZooFence: Principled Service Partitioning and Application to the ZooKeeper Coordination Service. In *33rd IEEE Intl. Symposium on Reliable Distributed Systems (SRDS)*.

[11] Kfir Lev-Ari, Edward Bortnikov, Idit Keidar, and Alexander Shraer. 2016. Modular Composition of Coordination Services. In *2016 Usenix Annual Technical Conference (ATC)*.

[12] Chao Li, Yushu Xue, Jing Wang, Weigong Zhang, and Tao Li. 2018. Edge-Oriented Computing Paradigms: A Survey on Architecture Design and System Management. *ACM Comput. Surv.* 51, 2 (April 2018).

[13] Mark Masse. 2011. *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces.* " O'Reilly Media, Inc.".

[14] Christopher Meiklejohn and Peter Van Roy. 2015. Lasp: A Language for Distributed, Eventually Consistent Computations with CRDTs. In *1st Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC)*.

[15] Fabrizio Montesi, Claudio Guidi, and Gianluigi Zavattaro. 2014. Service-Oriented Programming with Jolie. In *Web Services Foundations*, Athman Bouguettaya, Quan Z. Sheng, and Florian Daniel (Eds.). Springer, 81–107.

[16] Sam Newman. 2015. *Building microservices: designing fine-grained systems.* " O'Reilly Media, Inc.".

[17] Shadi A. Noghabi, John Kolb, Peter Bodik, and Eduardo Cuervo. 2018. Steel: Simplified Development and Deployment of Edge-Cloud Applications. In *10th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*.

[18] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu. 2016. Edge Computing: Vision and Challenges. *IEEE Internet of Things Journal* 3, 5.

[19] Jörg Thalheim, Antonio Rodrigues, Istemi Ekin Akkus, Pramod Bhatotia, Ruichuan Chen, Bimal Viswanath, Lei Jiao, and Christof Fetzer. 2017. Sieve: Actionable Insights from Monitored Metrics in Distributed Systems. In *18th ACM/IFIP/USENIX Middleware Conference*.

[20] S. Wang, X. Zhang, Y. Zhang, L. Wang, J. Yang, and W. Wang. 2017. A Survey on Mobile Edge Networks: Convergence of Computing, Caching and Communications. *IEEE Access* 5 (2017).