



**HAL**  
open science

## Split and migrate: Resource-driven placement and discovery of microservices at the edge

Genc Tato, Marin Bertier, Etienne Rivière, Cédric Tedeschi

### ► To cite this version:

Genc Tato, Marin Bertier, Etienne Rivière, Cédric Tedeschi. Split and migrate: Resource-driven placement and discovery of microservices at the edge. OPODIS 2019: 23rd International Conference On Principles Of Distributed Systems, Dec 2019, Neuchâtel, Switzerland. pp.1-16, 10.4230/LIPIcs.OPODIS.2019.29 . hal-02401933

**HAL Id: hal-02401933**

**<https://inria.hal.science/hal-02401933>**

Submitted on 10 Dec 2019

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# Split and migrate: Resource-driven placement and discovery of microservices at the edge

Genc Tato

Univ Rennes, Inria, CNRS, IRISA, France

Marin Bertier

Univ Rennes, Inria, CNRS, IRISA

INSA Rennes, France

Etienne Rivière

UCLouvain, Belgium

Cédric Tedeschi

Univ Rennes, Inria, CNRS, IRISA, France

---

## Abstract

Microservices architectures combine the use of fine-grained and independently-scalable services with lightweight communication protocols, such as REST calls over HTTP. Microservices bring flexibility to the development and deployment of application back-ends in the cloud.

Applications such as collaborative editing tools require frequent interactions between the front-end running on users' machines and a back-end formed of multiple microservices. User-perceived latencies depend on their connection to microservices, but also on the interaction patterns between these services and their databases. Placing services at the edge of the network, closer to the users, is necessary to reduce user-perceived latencies. It is however difficult to decide on the placement of *complete* stateful microservices at one specific core or edge location without trading between a latency reduction for some users and a latency increase for the others.

We present how to dynamically deploy microservices on a combination of core and edge resources to systematically reduce user-perceived latencies. Our approach enables the split of stateful microservices, and the placement of the resulting splits on appropriate core and edge sites. Koala, a decentralized and resource-driven service discovery middleware, enables REST calls to reach and use the appropriate split, with only minimal changes to a legacy microservices application. Locality awareness using network coordinates further enables to automatically migrate services split and follow the location of the users. We confirm the effectiveness of our approach with a full prototype and an application to ShareLatex, a microservices-based collaborative editing application.

**2012 ACM Subject Classification** Information systems → Distributed storage; Information systems → Service discovery and interfaces; Computer systems organization → Cloud computing

**Keywords and phrases** Distributed applications, Microservices, State management, Edge computing

**Digital Object Identifier** 10.4230/LIPIcs.OPODIS.2019.29

## 1 Introduction

Modern interactive applications combine a front-end running on client devices (e.g. in their web browser) with a back-end in the cloud. Collaborative *editing* applications, in which multiple users concurrently make changes to the same document, such as Google Docs, Microsoft Office 365, and ShareLatex, are good examples of such interactive applications. Quality of experience for users of such applications depends on low latencies between an action of one client and its visibility by other clients.

A solution to enable fast request-response latencies between the front-end and the back-end of a collaborative application is to deploy part of the back-end at the edge, i.e. on computing resources that are closer and accessible with low latencies from the front-end.



© Genc Tato, Marin Bertier, Etienne Rivière and Cédric Tedeschi;  
licensed under Creative Commons License CC-BY

23rd International Conference on Principles of Distributed Systems (OPODIS 2019).

Editors: Pascal Felber, Roy Friedman, Seth Gilbert, and Avery Miller; Article No. 29; pp. 29:1–29:16

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

45 It is often challenging to adapt an application to make use of edge resources. Software  
46 monoliths typically require massive re-engineering to support a deployment on multiple sites,  
47 as they base the collaboration between their constituents on shared memory or common  
48 databases. Service-Oriented Architectures (SOAs) on the other hand present desirable  
49 features for this adaptation, by splitting the features of the application into independent  
50 services and decoupling service location and naming.

51 Microservices are a popular approach to SOAs [9, 35] adopted by many large-scale  
52 companies [15, 17]. Features of the back-end are handled by fine-grained services commu-  
53 nicating through lightweight protocols, such as publish/subscribe or event stores [8]. The  
54 most common form of interaction between microservices is the use of point-to-point calls to  
55 Representational State Transfer (REST) APIs provided over HTTP.

56 We are interested in this work in the adaptation of microservices applications towards a  
57 joint deployment on core resources, e.g. in some cloud datacenter, and edge resources, e.g.  
58 at micro-clouds located in the same metropolitan-area network as the clients. Our objective  
59 is to reduce latencies between user actions and their visibility by other users.

60 We target collaborative editing applications based on microservices. We demonstrated in  
61 our previous work [25] that ShareLatex, an open source and microservices-based application  
62 for collaboratively editing  $\text{\LaTeX}$  documents, could benefit from reduced user-perceived  
63 latencies thanks to a *static* core/edge deployment of its microservices. This previous work  
64 considers however the placement of *entire* services onto different sites, which may lead to  
65 trading latency reduction for some users for latency increases for the others. It also does not  
66 consider the adaptation of this placement based on the actual location of the application  
67 users.

## 68 Contributions

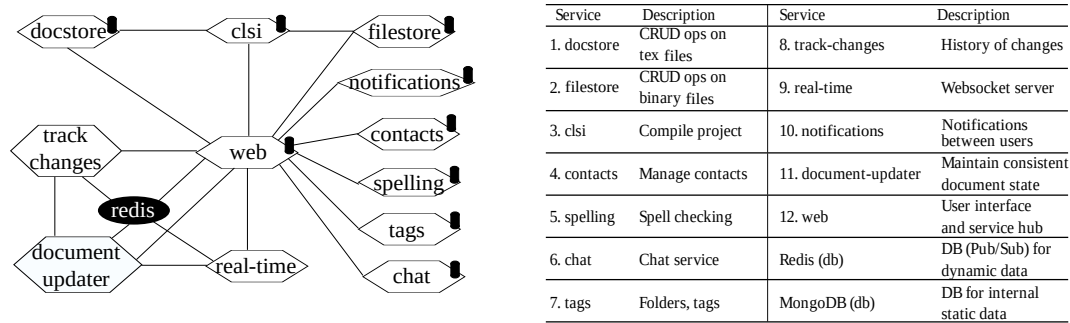
69 We consider in this paper the *dynamic* placement and migration of microservices in core/edge  
70 deployments. We leverage the use in modern microservices applications of resource-centric  
71 REST APIs and NoSQL databases partitioned by a single primary key. This allows us to  
72 *split* microservices, and create independent instances responsible for a partition of the original  
73 service’s data. These splits, deployed at different edge locations, can then handle requests  
74 for specific partitions of the service data, accessed by close-by users. We demonstrate our  
75 ideas with ShareLatex (§2).

76 Our first contribution is the support for splitting and multi-site placement of microservices.  
77 We detail how the state of a microservice can be partitioned, and how the resulting splits  
78 can be dynamically deployed on different core and edge sites (§3).

79 Our second contribution is the middleware support for the decentralized and dynamic  
80 *discovery* of microservice splits. We build on Koala [26], a lightweight Distributed Hash  
81 Table (DHT) for decentralized cloud infrastructures. We enable the transparent redirection  
82 of calls based on resource identifiers present in HTTP Uniform Resource Identifiers (URIs),  
83 also supporting the *relocation* of microservices splits. This allows adapting compatible legacy  
84 microservices applications for hybrid core/edge deployments with minimal effort (§4).

85 Our third contribution is a locality-driven policy conducting the *creation* and *migration*  
86 of microservices splits between the core and the edge, and between edge sites themselves,  
87 allowing to seamlessly adapt to the location of the users. This policy estimates latencies  
88 using network coordinates [13], enabling the automatic selection of the most appropriate site  
89 for the services splits used by a group of collaborative users, with the goal of achieving better  
90 response times (§5).

91 We demonstrate our ideas on the ShareLatex application, using a representative core-edge



■ **Figure 1** ShareLatex architecture (left) and list of constituents (right).

92 network topology and measuring the impact of latencies at the level of the application front-  
 93 end. Our results indicate that Koala and redirection layers induce only minimal overheads,  
 94 while the dynamic placement of microservices splits enables users in different regions to  
 95 access the same application with greatly reduced latencies (§6).

96 Finally, we present related work (§7) and conclude (§8).

## 97 2 ShareLatex and its core/edge deployment

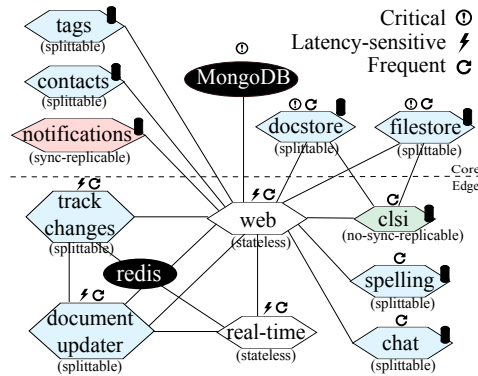
98 ShareLatex is a collaborative application allowing users (e.g. students, researchers or writers  
 99 of technical documentation) to concurrently edit a  $\text{\LaTeX}$  project. It features a web-based  
 100 editor with spellchecking and auto-completion, facilities for compiling and producing papers,  
 101 and tools for the collaboration between writers, such as an address book and a chat service.

102 Responsiveness is a key element of the perceived quality of service in ShareLatex. For  
 103 instance, a team of researchers could collaborate on the writing of algorithm pseudocode.  
 104 Changes made by one researcher must be visible with no undue delay by the others, and  
 105 changes must propagate as fast as possible to the reference document stored in the back-end  
 106 to avoid concurrency hazards.

107 The ShareLatex back-end uses 12 microservices and a database, **Redis**, shared by four of  
 108 them (Figure 1).<sup>1</sup> The **web** provides the front-end to the client browser and acts as an API  
 109 gateway to other services. User actions (cursor changes, new text, etc.) are propagated by  
 110 **web** to the **real-time** service using WebSockets. The **real-time** service then sends them to  
 111 **document-updater** which is responsible for maintaining a consistent order of modifications.  
 112 This dynamic state of the project is stored in **Redis**, and periodic snapshots are pushed  
 113 to the **docstore** (text files) and **filestore** (binary files). Figure 1 details the ShareLatex  
 114 architecture and its services.

115 Core servers are typically hosted in a centralized data center, while edge servers are  
 116 distributed and closer to the users. In our previous work [25], we demonstrated that  
 117 ShareLatex can benefit from a *static* placement of its services on a combination of core and edge  
 118 servers, closer to clients collaborating on a document. We build on our previous contribution,  
 119 which requires only minimalistic modifications to the configuration and deployment scripts  
 120 of ShareLatex, and no changes to the application code. The most significant modification

<sup>1</sup> Note that using a shared database does not fully comply with the microservices architectural pattern, where all state should be encapsulated in services. Yet, such compromises with the norm are found in many microservices-based applications. We prefer taking them into account rather than heavily modifying the legacy application code.



■ **Figure 2** Static ShareLatex deployment as suggested in [25].

121 performed in our previous work is the disassembly of the `web` service implementation from  
 122 its database. This was necessary as `web` acts as an API gateway and *must* be deployed at  
 123 the edge, but it also features a global database of information about users, which is queried  
 124 infrequently. These queries can be done remotely to a database in the core, with minimal  
 125 performance penalty.

126 The static core and edge placement of services of Figure 2 follows the recommendations  
 127 argued in our previous work [25]: `web`, `real-time`, `document-updater` and `Redis` should be  
 128 deployed on an edge site. Due to the coupling of `track-changes` with `Redis`, this service  
 129 must be deployed alongside to avoid remote calls, even if it does not influence perceived  
 130 latencies as much. The `clsi`, `spelling` and `chat` services can also be deployed at the edge,  
 131 with a moderate but positive impact on perceived latencies. This placement resulted in lower  
 132 latencies for operations impacting the most the user experience, at the cost of increasing  
 133 latencies for operations that require interactions between services at the edge and services  
 134 remaining in the core.

### 135 3 Splitting microservices

136 While some microservices may be *stateless*, most of them need to internally store and query  
 137 data. A stateful microservice is typically implemented as a business-logic tier combined  
 138 with a database. The choice of the appropriate database is specific to each microservice,  
 139 leading to what is sometimes called a *polyglot* architecture. Figure 1 represents the presence  
 140 of a database inside each service using a small black database symbol. In the unmodified  
 141 ShareLatex, only `real-time` is a stateless service. All other services are stateful, including  
 142 `document-updater` and `track-changes` which use the common `Redis` database. With the  
 143 decoupling of `web` from its database (as depicted in Figure 2), this service is also stateless  
 144 and uses remote calls to a `MongoDB` service.

145 A key property of SOA and therefore of microservices is the ability to independently *scale*  
 146 *in* and *out* the business-logic tier and the database [17]. For the former, new *instances* may  
 147 be created and deleted *on the fly*, e.g. using deployment middleware such as Kubernetes [7]  
 148 and a scaling policy [28]. Elastic scaling is difficult to realize with relational databases, and  
 149 microservices state may grow to large sizes requiring the ability to scale out storage to a  
 150 large number of servers. NoSQL options with such horizontal scaling abilities are therefore a  
 151 favored choice in many microservices applications.

152 NoSQL databases such as key/value stores or document stores, partition the data using a  
 153 unique primary key. We observe that very often, accesses to the database by the business-logic

154 tier for a query only read and write a *limited* and *identifiable* subset of keys. The identification  
155 of this subset typically depends on the characteristics of the query, and in particular on  
156 its *object*. It results that the state of the service, i.e. the content of the database, may be  
157 *partitioned* in such a way that keys that are accessed together for any future service requests  
158 belong to the same *partition*. This enables in turn the possibility to create multiple instances  
159 of the service, each equipped with one of the partitions. We call these services hosting  
160 independent partitions of the database *service splits*. A service that supports splitting is a  
161 *splittable* service.

162 Not all services are splittable. Some may require operations (e.g., Map/Reduce queries,  
163 scans, etc.) that operate on the entire content of the database. In some cases, it is not  
164 possible to identify a mapping between requests characteristics and partitions, e.g. when calls  
165 may use object keys generated at runtime or read from the database itself. These services  
166 are therefore only *replicable*: It is only possible to make *complete* copies of the service and its  
167 state. When these copies must be kept in sync for the well-functioning of the application, the  
168 service is *sync-replicable*. When operating on divergent copies does not impact, or impacts  
169 only marginally, the well-functioning of the application, provided that users systematically  
170 use the same copy, the service is *no-sync-replicable*.

171 The analysis of ShareLatex code results in the following categorization of services, also  
172 reflected in Figure 2.<sup>2</sup> The `notifications` service is *sync-replicable*, while `clsi`, handling  
173 the compilation, is *no-sync-replicable*: compilations across projects do not require consistent  
174 updates. The `web` service was initially *sync-replicable*, but the decoupling of its database  
175 makes it stateless. All other stateful services –a majority of them– are *splittable*. This means  
176 that their state (content of the services databases, but also the content of the shared `Redis`  
177 database) can be partitioned, and that partitions can be deterministically identified for any  
178 query. The object of the query, that allows identifying the partition of service state, and  
179 therefore the appropriate service split, is the specific writing *project* that the user is editing.  
180 In other words, the state of ShareLatex splittable services at the bottom of Figure 2 can be  
181 partitioned based on the project identifier, resulting in splits able to handle requests for a  
182 specific subset of projects. Such splits can then be deployed at the edge, and serve requests  
183 from close-by users accessing one of these projects.

184 The implementation of splitting requires support from the database embedded in splittable  
185 microservices, to be able to bulk load and store data partitions between an existing service  
186 and a newly created split. This support depends on the database API but does not pose  
187 implementation difficulties. For ShareLatex, we built minimalistic APIs enabling this for the  
188 `Redis` and `MongoDB` databases.

189 Our goal is to support the *dynamic* creation of service splits and their deployment  
190 over a combination of core and edge resources. This requires both appropriate *middleware*  
191 *support mechanisms* enabling the discovery and redirection of calls between microservices in  
192 a transparent manner, and appropriate *adaptation policies* to decide at runtime when and  
193 where to create splits, and when and where to migrate an existing split if its current location  
194 is not optimal. We cover these two aspects in the two following sections.

---

<sup>2</sup> This identification of services classes and partitions was performed manually, but did not represent a particularly difficult task in the case of ShareLatex. Automated or semi-automated service class identification and partitioning are beyond the scope of this paper, but we intend to explore these directions in our future work.

## 195 **4** Discovering and redirecting to microservice splits

196 We now present the mechanisms that support the dynamic deployment of service splits on  
197 multiple sites. Our focus in this section is on the proper functioning of the system during  
198 and after service splitting and migration operations. We present the policies triggering these  
199 operations in the next section.

200 Our support middleware serves two purposes: Firstly, it enables the *discovery* of services  
201 and splits, and the live modification of their placement (§4.1). Secondly, it enables the  
202 *redirection* of point-to-point calls between source and destination services, ensuring that the  
203 core service or its appropriate split is reached (§4.2).

### 204 **4.1** Discovery of microservice splits with Koala

205 Each service is initially associated with one instance in the core (the *core service*), responsible  
206 for its full state. Split and migrate operations dynamically update the list of splits for each  
207 service. Service discovery, therefore, requires the maintenance of an *index* of existing services,  
208 together with their current lists of splits. Every such split is associated with a list of object  
209 identifiers, for which this split is the only one able to process queries. This index must remain  
210 *strongly consistent*: At any point in time, there must be a single core service or split that can  
211 answer a query for a given object, and it must be impossible for two clients of the service  
212 under the same object to use different splits concurrently.

213 Service registries based on replicated databases updated using consensus (e.g., using  
214 etcd [11] or ZooKeeper [18]) are adapted for datacenter deployments with low network  
215 latencies. In our target context of distributed sites, centralizing the index would result in  
216 unacceptable overheads. We favor instead a decentralized design, supporting the caching  
217 and lazy revocation of split-to-site associations. This service is distributed, with an instance  
218 running at the core and at each of the edge sites.

219 Service discovery requests contain the name of the service, and for splittable services, the  
220 identifier of the query *object*. For ShareLatex splittable services, this object is the *project*  
221 *identifier*, that allows identifying the appropriate service state partition. Service discovery  
222 requests can be addressed to any of the sites.

223 The service index is implemented as a Distributed Hash Table (DHT), in which each  
224 node stores a subset of the index, partitioned using consistent hashing. Index elements are  
225 accessed using a primary key. Each node is responsible for a *range* of these keys. An overlay  
226 enables requests to deterministically reach the responsible node using greedy routing (each  
227 node in the path selects amongst the nodes it knows the closest to the destination). Typical  
228 DHT designs actively maintain all overlay links through the exchange of explicit overlay  
229 construction messages. In this work, we rely on Koala [27], a DHT that creates overlay links  
230 in a lazy manner, by piggybacking overlay construction messages over existing application  
231 traffic. This design choice enables to create more overlay links for routes in the overlay that  
232 are more frequently used for index reading requests, and minimize maintenance costs for  
233 seldom-used links. This is beneficial for workloads that are highly local, which is expected  
234 from service requests in one single application and to a relatively limited number of services  
235 (e.g. up to a few hundred).

### 236 **Indexing**

237 We keep two global indexes in Koala, an index of Objects, and an index of Splits. Figure 3  
238 shows an example of the local subset of these indexes maintained by one Koala node. A Koala

Object ID	Location	Responsibility	Split group
<b>Object 1</b>	local	YES	[Service 1 - Split 1, Service 2 - Split 1]
<b>Object 2</b>	local	6-8	[Service 1 - Split 1]
<b>Object 3</b>	5-2	YES	-

Service name	Split ID	Location	Responsibility	IP	Port
<b>Service 1</b>	<b>Split 1</b>	local	YES	x.x.25.1	3001
<b>Service 2</b>	<b>Split 1</b>	local	6-8	x.x.25.2	3002
<b>Service 3</b>	<b>Split 1</b>	9-7	YES		
<b>Service 3</b>	<b>Split 2</b>	5-2	YES		

■ **Figure 3** Indexes stored at some Koala DHT node: Objects table (left) and Splits table (right). Primary keys are in boldface.

node is *responsible* for maintaining the authoritative and strongly consistent entry for a number of index items, falling in its key responsibility range. It also maintains *local* resources, objects and splits, that are *hosted* on the corresponding edge site. A Koala node may have local resources for which it is not responsible or be responsible for resources that are not local. This design enables the creation of resources on a different node than the one that the DHT assigns for the corresponding entry index, while maintaining a single node in charge of this index entry and allowing atomic modifications. Lookups follow multiple hops in the overlay, until the *responsible* node is found, leading to one last hop to the node where the entry is *local* (if different). Nodes hosting locally a resource access it without involving the responsible node.

## Discovery

A local split can only be reached by proxying through the local Koala instance.<sup>3</sup> The discovery of the appropriate site for an incoming service request proceeds in two phases. First, the Objects table in the DHT is queried to establish whether there exists a split of at least one service under that object. This information is stored in the *split group* for that object. If there is no entry for the object, or if there is no entry for the specific service in the split group, the request must go to the core. Second, the Koala node responsible for the split is located using the Splits table, using both the service name and split number as the key. This requires reaching the Koala node that is *responsible* for that key and then reaching the Koala node where that split is *local*.

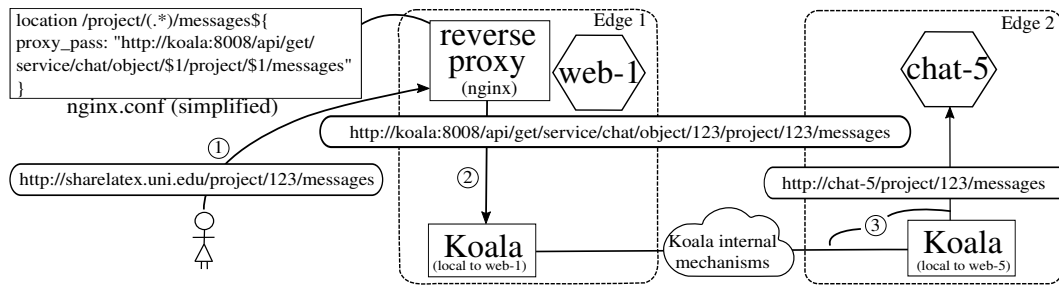
For instance, on the node whose local subset of the index is represented by Figure 3, a request to Service 2 for Object 1 will be immediately sent to Service 2's Split 1 hosted locally. A request for Object 3 will be redirected in one hop to Koala node of identifier 5-2, to read its split group. A request for Object 4, not present in the local state, requires a multi-hop routing in the Koala overlay to request its service group.

## Caching

Looking up service discovery entries in the DHT for *every* service call is too expensive. We implement *caching*: results of index lookup are kept at the local Koala node and reused. Stale cache entries are discarded in a lazy fashion. We leverage the fact that all requests must go through the *local* Koala node, e.g. on the edge site where the split actually runs. After the migration to a new site, queries based on stale cached information will arrive at the Koala node at the *previous* local location of the split. This node simply informs the origin, which invalidates related cache entries and falls back to a regular lookup.

<sup>3</sup> Allowing uncontrolled connections from outside of the edge site might not be possible due to network constraints, or not desirable for security reasons. The local Koala node acts, therefore, as an API gateway for all local service splits.





■ **Figure 4** Example of REST call redirections in ShareLatex.

## 272 Migration

273 The *migration* of an existing split, or the creation of a new split, follows four phases. Firstly,  
 274 an instance of the service is bootstrapped if none already exists at the destination edge site,  
 275 or it is selected among existing instances, but it does not hold state or service requests.  
 276 Secondly, a new entry in the Splits table is created to announce the existence of the new  
 277 split. It does not contain a location yet. The split group for all corresponding objects is  
 278 updated to indicate the temporary unavailability of the split. Service requests will block at  
 279 the lookup request stage, and back off for a random time duration. Thirdly, the new instance  
 280 receives the partition of the data from the source service or split. Finally, the Koala entry for  
 281 the split is updated to reflect the location of the new *local* site for that split, and the split  
 282 groups for all corresponding objects are updated. This allows request services to resume,  
 283 using the new split location.

## 284 4.2 Transparent redirection of REST service calls

285 Modifying legacy microservices applications to directly make use of Koala APIs to discover  
 286 and call services and splits would require an important effort. Instead, we leverage the  
 287 fact that the objects of queries are accessible in the URIs of REST service calls. Indeed,  
 288 REST being a resource-centric approach to designing interfaces, calls are made, typically  
 289 over HTTP, to an explicit resource given in the request URI. We implement the transparent  
 290 redirection of calls by extracting the object from this URI. Then, the local Koala node  
 291 queries for the existence of a split for that object and the requested service. The request URI  
 292 is transformed using rewriting rules to reach either the original core service, or the Koala  
 293 node on the edge site where the split runs.

294 The implementation of the redirection is as follows. It is illustrated for a call in ShareLatex  
 295 in Figure 4. We use the high-performance web server `nginx` as a reverse proxy for calls from,  
 296 and to, local services. In ShareLatex, this includes the `web` service that serves as an API  
 297 gateway for the user frontend. The reverse proxy translates the original request from the  
 298 unmodified ShareLatex, to a request to the local Koala node. The discovery process detailed  
 299 before establishes that there exists a split for that service that must serve the request. In the  
 300 example of Figure 4, the `web` service on the Edge 1 site calls the `chat` service. The object  
 301 “123”, the project identifier, is extracted from the call URI. Koala then determines that the  
 302 service split is on the Edge 2 site. The request is redirected to `chat` service in that site,  
 303 where the call is handled by Koala.

## 5 Splits creation and migration policy

The creation of service splits and their migration between sites obey an adaptation *policy*. This policy must determine *what* service to split, *when* these split decisions are made and *where* to (re)deploy the splits. Its goal is to ensure that user-perceived latencies in the application are minimized.

### What service to split?

The first aspect of the policy is application-dependent and results from the analysis of the interactions between its microservices. A set of *splittable* services, and not necessarily all of them, must be tagged for a preferential deployment at the edge. This aspect of the ShareLatex policy builds upon our previous results [25] (§2). Microservices that lie in the bottom part of Figure 2 are tagged for edge deployment. All other services always remain in the core.

### When should splits happen?

There are two situations where a split may be formed: When a new object is created, and when latencies to the core are too high. The first option is sufficient for the ShareLatex policy: The creation of a new project leads to the immediate creation of all corresponding splits.

### Where should splits go?

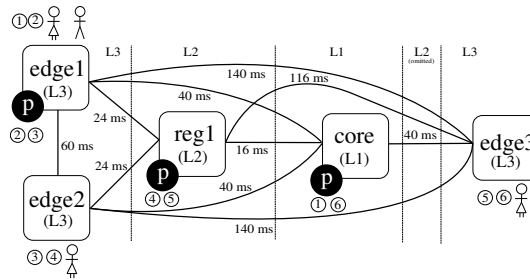
This aspect of the policy is twofold: Firstly, we must ensure that splits are created on a site (core or edge) close to the first user of the corresponding object. Secondly, we must adapt this placement when the chosen site is no longer the most adequate for the *current set* of users of that object.

This requires the ability to evaluate network latencies. Active probing of latencies (e.g. using ICMP packets) is impractical and unscalable. We combine two mechanisms to enable probe-less estimations. Firstly, we enforce that users always connect to the *geographically closest* site.<sup>4</sup> The location of a client is that of its connection site. Secondly, latencies between sites are estimated using Network Coordinates (NCs). We use Vivaldi [13] to compute NCs. Each site is represented by a  $d$ -dimensional point. These points positions evolve following a process similar to a spring-mass relaxation, based on observed latencies for *actual* message exchanges, and Euclidean distances eventually approximate latencies.

The ShareLatex policy enforces that the initial version of an object, and the corresponding splits, be hosted by the connection site of the first user. Each site collects for its local splits, a history of the NCs of the sites forwarding client calls. Periodically (every 5 minutes, or 100 requests, whichever comes first, in our implementation), the policy determines whether migration of the splits for each hosted object is necessary. Several users access a project, from different sites and with different frequencies. The ideal location of the splits for that project can be represented as a point in the NCs space. We define this point as the Center of Mass (CoM) for that object. It is the geometric average of the connection sites' NCs, weighted by the number of accesses from their clients. If there exists a site whose NC is closer to the CoM, the policy triggers a migration of all splits for that object to this new site.

---

<sup>4</sup> The list of core and edges sites IP is publicly known. Clients use an IP-to-location service (e.g. [www.iplocation.net](http://www.iplocation.net)) and choose the geographically closest site.



■ **Figure 5** Topology and first experiment setup.

## 344 6 Evaluation

345 We evaluate the split and migrate principles with a full prototype, combining Koala, `nginx`  
 346 reverse proxies, Docker CE for bootstrapping containers on the core and edge sites, and  
 347 ShareLatex as the application.

348 Our evaluation aims at answering the following research questions: (i) Is the approach  
 349 able to reduce perceived latencies for users of the application? (ii) Can the policy successfully  
 350 migrate splits between edge sites when users' locations change? (iii) Is the overhead of using  
 351 Koala and proxying acceptable?

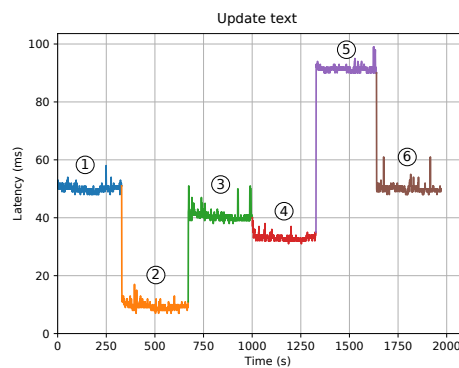
352 We consider the three-layer (L1-L3) hierarchical topology shown in Figure 5. Its char-  
 353 acteristics are derived from information obtained from an Internet Service Provider in the  
 354 EU [23]. Layer L1 consists of the *core* site, L2 of regional sites (*reg1*) and L3 of edge sites  
 355 (*edge1*, *edge2* and *edge3*). We deploy each site on a node of the Grid'5000 [5] testbed. Each  
 356 node features 2 Intel Xeon E5-2630 v3 CPUs and 128GB of RAM. We emulate latencies  
 357 between sites using the `tc` (traffic control) tool. Note that *reg1* is treated as an edge site,  
 358 and that we ignore latencies between users and sites, and model their mobility by enforcing  
 359 that they connect to a specific (closest) site. We use Network Coordinates (NCs) in  $d = 2$   
 360 dimensions for ease of presentation, although a higher dimensionality (e.g.  $d = 5$ ) would  
 361 yield better estimations. Latencies are measured at the level of the instrumented ShareLatex  
 362 frontend. We emulate the activity of users using the `Locust` [1] load testing tool, which allows  
 363 describing programmatically the behavior of users as a list of actions and their respective  
 364 occurrence frequencies.

### 365 6.1 Adaptation and split migrations for moving users

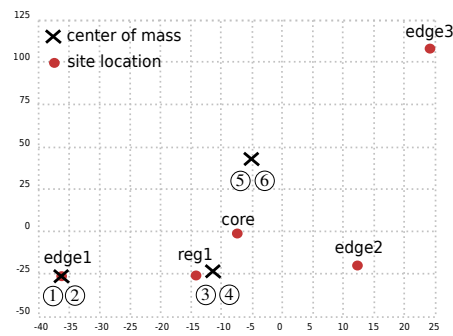
366 Our first experiment evaluates the ability of our approach to adapt the location of the splits  
 367 for single a ShareLatex project, and the impact this has on latencies. We consider a project  
 368  $p$  shared by two equally active users, one stationary and one who changes her location  
 369 continuously. Each user performs one operation every second, adding a new character to the  
 370 text. The user-perceived latency is measured from the moment the text is updated by one  
 371 user to the moment the update appears in the screen of the other user.

372 Figure 5 presents the experiment setup. Figure 6 presents the evolution of the average  
 373 perceived latency for the two users, and Figure 7 presents the evolution of the CoM of the  
 374 project. Circled numbers in all figures show the sequence of operations.

375 We follow three phases. In each phase, users are assigned to connection sites, and we  
 376 observe the triggering and impact of the adaptation and resulting split migration decisions.  
 377 Initially, both users are closer to *edge1* and therefore connect to that site. The latency for  
 378 updating the text (50 ms) is roughly the RTT between *edge1* and *core*, plus the processing



■ **Figure 6** Evolution of *text update latencies* when migrating splits to follow a project CoM.



■ **Figure 7** Evolution of Network Coordinates and CoMs when migrating splits.

Project	p1	p2	p3	p4	p5	p6	p7	p8	p9	p10
Users	u1, u5, u6	u2	u1, u3	u1, u4	u5, u6	u6	u4, u7	u5, u8, u9	u5, u8, u9	u8, u10
User locations	e1, e2, e2	e1	e1, e1	e1, e1	e2, e2	e2	e1, e2	e2, e3, e3	e2, e3, e3	e3, e3
Ideal site(s)	e2, r1, e1	e1	e1	e1	e2	e2	r1, e1, e2	e3, core, e2	e3, core, e2	e3

■ **Table 1** Distribution of projects, users and ideal site placements.

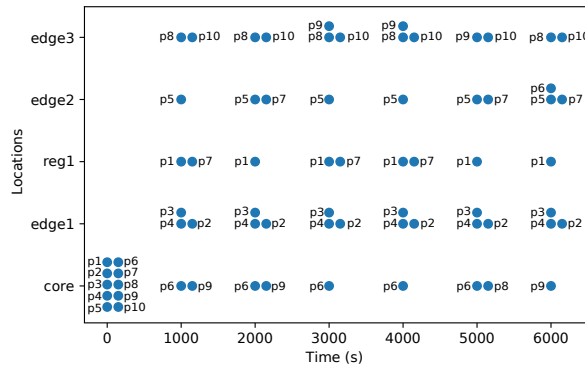
379 time, of 40 ms and 10 ms respectively (① in Figure 6). Given that all requests for project  $p$   
 380 originate from the Koala instance on *edge1*, that location is also the CoM (① in Figure 7),  
 381 and therefore the policy decides to split and migrate all tagged services to this site (②). The  
 382 latency drops to slightly over the processing time. In a second phase, we move one of the  
 383 users to *edge2* while the service splits for the project are still in *edge1* (③). This results in  
 384 an increase in latencies. When it next triggers, the adaptation policy decides to migrate  
 385 the service splits to *reg1* which is closer to the new CoM for the project (④). In the third  
 386 phase, we move the user of *edge2* and connect it to *edge3* (⑤). The service splits are still  
 387 in *reg1*, which results in high latencies. Again, the adaptation policy triggers and orders  
 388 the migration of splits to the closest site to the CoM (⑥). The *core* happens to be the best  
 389 compromise to serve the two users connected to *edge1* and *edge3*. This experiment shows  
 390 that the policy is effective in splitting and migrating a single project according to its user  
 391 locations, for a positive impact on perceived latencies.

## 392 6.2 Evolution of splits distributions

393 This second experiment shows how the split and migrate principles allow shifting the load  
 394 from the core servers to edge servers while following the location of the most active users  
 395 in a *collection* of ShareLatex projects. All services are initially only in *core*. We consider  
 396 10 users and 10 projects. Each project is edited by 1, 2 or 3 users. The two first lines of  
 397 Table 1 show the mapping between users and projects. The third line indicates the (static)  
 398 user locations for each project.

399 We model the activity of users to represent work sessions. During one hour and a half,  
 400 every user randomly picks one of their assigned projects and edits it for a random duration  
 401 of 2 to 10 minutes. The project CoM evolves to follow the location(s) of the currently active  
 402 user(s). The fourth line of Table 1 indicates the possible ideal location(s) for the project  
 403 splits, calculated offline.

404 We monitor the location of the service splits for the different projects, taking snapshots



■ **Figure 8** Evolution of splits placements.

405 every 1,000 seconds. We run this experiment until the projects with a single ideal site  
 406 placement reach this destination. Figure 8 presents these snapshots and the location of the  
 407 service slices for the 10 projects. Projects whose ideal site is unique, such as *p2-p5* and  
 408 *p10*, have the corresponding service slices migrated to these sites correctly and immediately.  
 409 Projects with multiple ideal sites see their slices periodically migrate between these sites,  
 410 following the currently active user(s). For instance, splits for *p7* move between *reg1* and  
 411 *edge2*, while splits for *p8* and *p9* move between *edge3* and *core*. The final site is highlighted  
 412 in boldface in Table 1. This experiment shows that the split and migrate mechanisms and  
 413 the adaptation policy for ShareLatex allow dynamically moving microservices close to the  
 414 users, based on the used resources (*projects* in ShareLatex).

### 415 6.3 Overheads of Koala and redirections

416 In this final experiment we evaluate the costs and overheads of the mechanisms enabling  
 417 transparent call redirections. To isolate the overhead we compare a centralized setting where  
 418 everything is deployed in the *core*, corresponding to the original ShareLatex model, with a  
 419 one-edge-site setting where requests are redirected from this edge site to the core by Koala.  
 420 Figure 9 presents this setup. We use a 50 ms latency between edge and core sites.

421 In both settings, the service split that responds to the user request is in *core*. In the  
 422 centralized setting the request is first sent to the **web** core service and then forwarded to the  
 423 right service directly, while in the second setting the request goes first through the local **web**  
 424 split. This proxies the request to the Koala instance on *edge1*, which in turn forwards it to  
 425 the Koala instance in *core* who then calls the service.

426 We distinguish three kinds of requests, two HTTP REST calls and one WebSocket request.  
 427 For the REST calls, we consider a call to **tags**, for which splitting is disallowed (①), and a  
 428 call to **chat**, which is splittable using the project identifier as the object (②). The WebSocket  
 429 request updates the text (writing) ③. It is also a project-specific request and must reach the  
 430 corresponding split of the **document-updater** service.

431 We expect a slightly higher overhead for redirections to split services compared to non-split  
 432 ones. For non-split services, a single interaction with Koala is required (follow ①). For split  
 433 services, two interactions are necessary: one to locate the object and one to redirect to the  
 434 correct split (follow ② and ③).

435 The operation latencies times of the three requests with and without the redirection are  
 436 shown in Figure 10. We consider two cases for the redirection: without and with caching.  
 437 When the cache is disabled, lookups on the Koala DHT can require multiple hops between

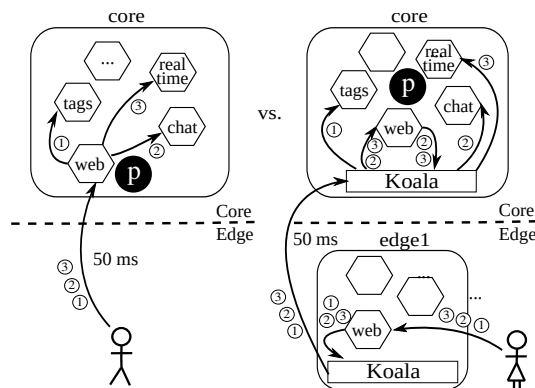


Figure 9 Setup for the experiment evaluating the overheads of Koala and redirections.

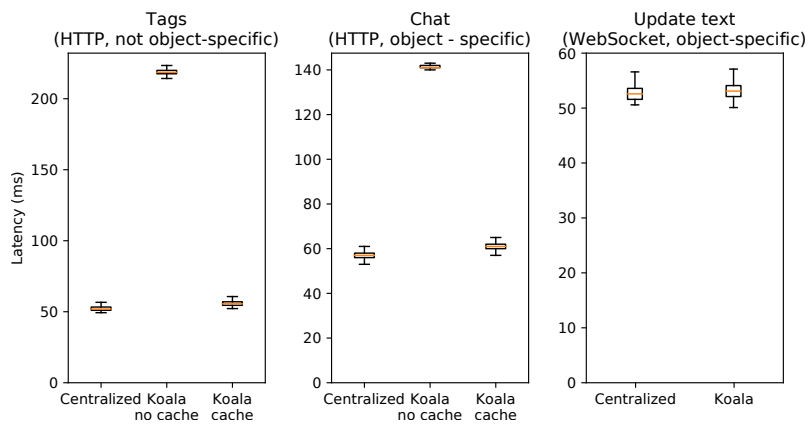


Figure 10 Evaluation of the overheads of Koala and redirections.

438 sites and incur a significant and unpredictable penalty. With caching, this penalty is only  
 439 paid for the first access or after a migration invalidates the cached information. WebSocket  
 440 requests occur on an established connection, therefore caching does not apply.

441 Figure 10 presents the distribution of latencies for the three operations and for 500  
 442 requests each. We observe a similar performance between the centralized setting and the  
 443 setup using caching. The median overhead of proxying through the local edge site is  $\approx 3$  ms  
 444 for the non-split service and  $\approx 4$  ms for the split one. For WebSockets operations this  
 445 difference is smaller,  $\approx 1$  ms, which can be explained by the fact that this protocol is more  
 446 lightweight than HTTP. Disabling caching leads to significant overheads as every operation  
 447 leads to lookups in the DHT, bouncing between the core and edge Koala instances. This  
 448 experiment shows that the latency impact of proxying through the edge is likely to be  
 449 negligible compared to the gain of using locally-deployed services splits.

## 450 7 Related work

451 Previous research advocates to revisit the SOA paradigm for supporting service-based  
 452 applications deployed in edge cloud platforms [19]: In light of the increase of the number of  
 453 services at the edge able to answer a specific query, service registration must take into account  
 454 spatial coverage, and service discovery must take locality into account. Our contributions  
 455 are a step in that direction.

456 The placement of applications on fog platforms has been an active research topic in the

457 recent years. One target domain is IoT applications where data collected from connected  
458 objects must be processed on nearby resources [24, 34]. Stream processing is another  
459 application that benefits from deployments on a combination of core and edge resources. It  
460 explicits its communication patterns (i.e., the directed acyclic graph linking stream processing  
461 operators), which can be leveraged for optimal placement on edge resources [12]. The Balanced  
462 RePartitioning (BRP) [4] algorithm targets generic distributed cloud applications and devises  
463 online algorithms which find a good trade-off between communication and migration costs.

464 Our work is linked with the concept of *mobile edge clouds*, where users move and  
465 connect to nearby resources dynamically [30]. When the mobility of users is modeled using  
466 Markov stochastic decision processes, analytical frameworks allow devising close-to-optimal  
467 algorithms for automating service placement [31]. Other approaches advocate the use of  
468 genetic algorithms to gradually refine an allocation of services to the edge [33].

469 We note that all of the aforementioned work considers the placement (and in some cases  
470 the migration) of *full* instances of services. We are not aware of solutions proposing to split  
471 stateful microservices and support resource-based discovery. State splitting is used, in a  
472 different context, for the elastic scaling of publish/subscribe middleware [6].

473 Research on collaborative edition has focused on enabling correctness and performance,  
474 including in the presence of network issues. The Jupiter protocol [21, 32] and the RGA  
475 protocol [22] implement a *replicated list object* abstraction and define how to propagate  
476 updates to achieve convergence [3]. Our work is complementary: The responsiveness of  
477 replicated list object algorithms (i.e. the time between an update and its visibility at the  
478 other clients) is sensitive to the latency between client nodes and a coordination server.

479 Service discovery middleware solutions for data centers typically rely on strongly consistent,  
480 fully replicated stores maintaining the complete index of services instances and of their  
481 locations. SmartStack [2], used for example by the Synapse [29] microservices platform,  
482 is based on Apache ZooKeeper [18]. Similarly to Koala, Synapse instances provide local  
483 proxies to services, but each maintains a full copy of the index while Koala relies on a  
484 DHT and caching for scalability. Kubernetes [10] leverages etcd [11] for service discovery.  
485 Recent work [14] suggests to add support for network coordinates [13] to route requests  
486 based on network locality. Yet, service selection decision remains a centralized process unlike  
487 with Koala where it can happen at the edge. Eureka [20] is also centralized but introduces  
488 the notion of *read clusters* that can serve requests closer to the clients. Unlike lazy cache  
489 management in Koala, read clusters must be explicitly synchronized when the service index  
490 changes. *Write clusters* can also be replicated, but are only eventually consistent, which  
491 makes them ill-suited for implementing consistent service migration. Finally, Consul [16]  
492 supports deployment to multiple data centers, and use network coordinates for location-aware  
493 selection. Consul only uses consensus-based synchronization within each individual data  
494 center. Updates propagate lazily between data centers using gossip, preventing consistent  
495 service relocation *across* data centers.

## 496 **8 Conclusion**

497 We presented how microservices could be dynamically deployed on a combination of core  
498 and edge resources. Our approach leverages the possibility to *split* microservices for which  
499 partitions of the data can be used to answer subsets of service requests independently. The  
500 Koala middleware enables to transparently redirect requests to the appropriate split based  
501 on object information available in REST calls URIs. Migration policies enable a dynamic  
502 placement of microservices splits on edge sites, and as our evaluation with the ShareLatex  
503 application shows, allow following the users and reduce perceived latencies.

504 This work opens interesting perspectives that we intend to consider in our future work.  
505 First, we wish to explore the automation of the identification of splittable microservices, and  
506 the use of static and dynamic analysis techniques to infer the relation between objects and  
507 state partitions. Second, we intend to extend support middleware to support redirections  
508 with other forms of communication, such as publish/subscribe or event sourcing [8]. Finally,  
509 we would like to build tools to automatize the identification of placement policies based on  
510 dynamic observations of communications between microservices.

### 511 Acknowledgments

512 We thank the anonymous reviewers for their comments. This work was partially funded by  
513 the Belgian FNRS project DAPOCA (33694591) and partly supported by the Inria Project  
514 Lab program Discovery (<http://beyondtheclouds.github.io/>).

### 515 References

---

- 516 1 Locust: An open source load testing tool. <https://www.locust.io>.
- 517 2 Airbnb. SmartStack Service Discovery in the Cloud. <https://bit.ly/2SAvRHn>.
- 518 3 Hagit Attiya, Sebastian Burckhardt, Alexey Gotsman, Adam Morrison, Hongseok Yang, and  
519 Marek Zawirski. Specification and complexity of collaborative text editing. In *ACM Symposium*  
520 *on Principles of Distributed Computing*, PODC. ACM, 2016.
- 521 4 Chen Avin, Andreas Loukas, Maciej Pacut, and Stefan Schmid. Online balanced repartitioning.  
522 In *International Symposium on Distributed Computing*, DISC. Springer, 2016.
- 523 5 Daniel Balouek, Alexandra Carpen Amarie, Ghislain Charrier, Frédéric Desprez, Emmanuel  
524 Jeannot, Emmanuel Jeanvoine, Adrien Lèbre, David Margery, Nicolas Niclausse, Lucas  
525 Nussbaum, Olivier Richard, Christian Pérez, Flavien Quesnel, Cyril Rohr, and Luc Sarzyniec.  
526 Adding virtualization capabilities to the Grid’5000 testbed. In *Cloud Computing and Services*  
527 *Science*, volume 367 of *Communications in Computer and Information Science*. Springer, 2013.
- 528 6 Raphaël Barazzutti, Thomas Heinze, André Martin, Emanuel Onica, Pascal Felber, Christof  
529 Fetzer, Zbigniew Jerzak, Marcelo Pasin, and Etienne Rivière. Elastic scaling of a high-  
530 throughput content-based publish/subscribe engine. In *34th International Conference on*  
531 *Distributed Computing Systems*, ICDCS. IEEE, 2014.
- 532 7 David Bernstein. Containers and cloud: From LXC to Docker to Kubernetes. *IEEE Cloud*  
533 *Computing*, 1(3):81–84, 2014.
- 534 8 Dominic Betts, Julian Dominguez, Grigori Melnik, Fernando Simonazzi, and Mani Subrama-  
535 nian. *Exploring CQRS and Event Sourcing: A journey into high scalability, availability, and*  
536 *maintainability with Windows Azure*. Microsoft patterns & practices, 2013.
- 537 9 Fabienne Boyer, Xavier Etchevers, Noël De Palma, and Xinxu Tao. Architecture-based  
538 automated updates of distributed microservices. In *International Conference on Service-*  
539 *Oriented Computing*, ICSOC. Springer, 2018.
- 540 10 Cloud Native Computing Foundation. Kubernetes. <https://kubernetes.io/>.
- 541 11 CoreOS. Etcd reliable key-value store. <https://coreos.com/etcd/>.
- 542 12 Alexandre da Silva Veith, Marcos Dias de Assuncao, and Laurent Lefevre. Latency-aware  
543 placement of data stream analytics on edge computing. In *International Conference on*  
544 *Service-Oriented Computing*, ICSOC. Springer, 2018.
- 545 13 Frank Dabek, Russ Cox, Frans Kaashoek, and Robert Morris. Vivaldi: A decentralized network  
546 coordinate system. In *ACM SIGCOMM Computer Communication Review*, volume 34, 2004.
- 547 14 Ali Fahs and Guillaume Pierre. Proximity-aware traffic routing in distributed fog computing  
548 platforms. In *IEEE/ACM International Symposium in Cluster, Cloud, and Grid Computing*,  
549 CCGrid, 2019.
- 550 15 Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana  
551 Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. An open-source benchmark suite  
552 for microservices and their hardware-software implications for cloud & edge systems. In *24th*



- 553 *International Conference on Architectural Support for Programming Languages and Operating*  
554 *Systems*, ASPLOS. ACM, 2019.
- 555 16 HashiCorp. Consul. <https://www.consul.io/>.
- 556 17 Wilhelm Hasselbring and Guido Steinacker. Microservice architectures for scalability, agility  
557 and reliability in e-commerce. In *Workshops of the Intl. Conf. on Software Architecture*, ICSA  
558 Workshops. IEEE, 2017.
- 559 18 Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: Wait-free  
560 coordination for internet-scale systems. In *USENIX Annual Technical Conference*, ATC, 2010.
- 561 19 Valérie Issarny, Georgios Bouloukakis, Nikolaos Georgantas, and Benjamin Billet. Revisiting  
562 service-oriented architecture for the IoT: a middleware perspective. In *International Conference*  
563 *on Service-Oriented Computing*, ICSOC. Springer, 2016.
- 564 20 Netflix. Eureka 2.0. <https://bit.ly/2Mcexda>.
- 565 21 David A Nichols, Pavel Curtis, Michael Dixon, John Lamping, et al. High-latency, low-  
566 bandwidth windowing in the jupiter collaboration system. In *ACM Symposium on User*  
567 *Interface Software and Technology*, 1995.
- 568 22 Hyun-Gul Roh, Myeongjae Jeon, Jin-Soo Kim, and Joonwon Lee. Replicated abstract data  
569 types: Building blocks for collaborative applications. *Journal of Parallel and Distributed*  
570 *Computing*, 71(3):354–368, 2011.
- 571 23 Sanhaji A. (Orange Labs Networks, France). Private communication, 2019.
- 572 24 Olena Skarlat, Matteo Nardelli, Stefan Schulte, Michael Borkowski, and Philipp Leitner.  
573 Optimized iot service placement in the fog. *Service Oriented Computing and Applications*,  
574 11(4), 2017.
- 575 25 Genc Tato, Marin Bertier, Etienne Rivière, and Cédric Tedeschi. Sharelatex on the edge:  
576 Evaluation of the hybrid core/edge deployment of a microservices-based application. In *3rd*  
577 *Workshop on Middleware for Edge Clouds & Cloudlets*, MECC. ACM, 2018.
- 578 26 Genc Tato, Marin Bertier, and Cédric Tedeschi. Designing overlay networks for decentralized  
579 clouds. In *Int. Conf. on Cloud Computing Technology and Science*, CloudCom. IEEE, 2017.
- 580 27 Genc Tato, Marin Bertier, and Cédric Tedeschi. Koala: Towards lazy and locality-aware  
581 overlays for decentralized clouds. In *2nd IEEE International Conference on Fog and Edge*  
582 *Computing*, ICFEC, 2018.
- 583 28 Giovanni Toffetti, Sandro Brunner, Martin Blöchliger, Florian Dudouet, and Andrew Ed-  
584 monds. An architecture for self-managing microservices. In *AIMC Workshop*. ACM, 2015.
- 585 29 Nicolas Viennot, Mathias Lécuyer, Jonathan Bell, Roxana Geambasu, and Jason Nieh. Synapse:  
586 A microservices architecture for heterogeneous-database web applications. In *10th European*  
587 *Conference on Computer Systems*, ACM EuroSys, 2015.
- 588 30 Shiqiang Wang, Rahul Urgaonkar, Ting He, Kevin Chan, Murtaza Zafer, and Kin K Leung.  
589 Dynamic service placement for mobile micro-clouds with predicted future costs. *IEEE*  
590 *Transactions on Parallel and Distributed Systems*, 28(4), 2016.
- 591 31 Shiqiang Wang, Rahul Urgaonkar, Murtaza Zafer, Ting He, Kevin Chan, and Kin K Leung.  
592 Dynamic service migration in mobile edge-clouds. In *IFIP Networking Conference*, 2015.
- 593 32 Hengfeng Wei, Yu Huang, and Jian Lu. Specification and Implementation of Replicated List:  
594 The Jupiter Protocol Revisited. In *22nd International Conference on Principles of Distributed*  
595 *Systems*, OPODIS, Leibniz International Proceedings in Informatics (LIPIcs), 2018.
- 596 33 Hongyue Wu, Shuiguang Deng, Wei Li, Min Fu, Jianwei Yin, and Albert Y Zomaya. Service  
597 selection for composition in mobile edge computing systems. In *International Conference on*  
598 *Web Services*, ICWS. IEEE, 2018.
- 599 34 Ye Xia, Xavier Etchevers, Loic Letondeur, Adrien Lebre, Thierry Coupaye, and Frédéric  
600 Desprez. Combining heuristics to optimize and scale the placement of iot applications in the  
601 fog. In *IEEE/ACM 11th Int. Conf. on Utility and Cloud Computing*, UCC, 2018.
- 602 35 Uwe Zdun, Elena Navarro, and Frank Leymann. Ensuring and assessing architecture confor-  
603 mance to microservice decomposition patterns. In *International Conference on Service-Oriented*  
604 *Computing*, ICSOC. Springer, 2017.