Is it Worth Migrating a Monolith to Microservices? An Experience Report on Performance, Availability and Energy Usage

Vincent Berry
Polytech Engineering School, LIRMM
Université Montpellier, CNRS
Montpellier, France
0000-0001-7271-4027

Arnaud Castelltort
Polytech Engineering School, LIRMM
Université Montpellier, CNRS
Montpellier, France
arnaud.castelltort@umontpellier.fr

Benoit Lange
ZENITH Team
INRIA & LIRMM
Montpellier, France
0009-0007-7820-8249

Joan Teriihoania
Polytech Engineering School
Université Montpellier
Montpellier, France
Lizard Global, Rotterdam, The Netherlands
joan@lizard.global

Chouki Tibermacine
Polytech Engineering School, LIRMM
Université Montpellier, CNRS
Montpellier, France
0000-0002-2063-0291

Catia Trubiani
Computer Science Department
Gran Sasso Science Institute
L'Aquila, Italy
catia.trubiani@gssi.it

***Abstract*—The microservice architecture (MSA) emerged as an evolution of existing architectural styles with the promise of improving software quality by decomposing an app into modules that can be maintained, deployed, and scaled independently. However, the transition from a monolithic to a microservice architecture is fraught with difficulties, especially when it comes to assessing qualitative aspects, as controversial results can arise. In this paper, we present an experience report on the migration of a monolithic web application and use performance, availability and energy efficiency as quality attributes to shed light on such an architectural transition. Horizontal scaling, i.e., distributing the workload across several service instances, is applied and we study its impact.**

Our main findings are: i) when no app component is replicated, MSA outperforms the monolithic architecture; ii) the monolithic architecture shows performance and availability improvement when replicating the entire app; iii) the replicated MSA version reaches a ceiling when not replicating its routing part (i.e., the API gateway), showing worse response times compared to the replicated monolith; iv) when replicating the API gateway, the MSA version reaches optimal performance with fewer replicates than the monolith; v) when not replicating services, MSA consumes more CPU resources than the monolithic architecture; vi) when scaling up, the MSA version is more efficient than the replicated monolith in terms of memory usage, and it can better exploit CPU resources; vii) when not replicating services, MSA consumes more energy than the monolithic architecture, whereas when scaling up, the MSA version is more efficient than the replicated monolith; MSA version reaches a good balance between CPU and memory usage.

***Index Terms*—Microservices, architecture migration, horizontal scaling, load tests, performance, availability, energy usage.**

I. Introduction

Module Decoupling is one of the oldest principles in software engineering practices [27], [31]. Pushing decoupling at the highest abstraction levels of software systems shifted the focus from programming paradigms, like the structured, including procedural, or object-oriented ones, to architecture styles that promote the design of systems in terms of components, services or microservices (MS). These henceforth first-class entities enable building software systems with a large flexibility in their deployment, delivery, scalability, and maintenance [7]. Component-based architectures promote a modular design of software systems by decoupling their modules through required and provided interfaces, and delaying their instantiation and interconnection by leveraging connectors.

Service-oriented architectures pushed decoupling further by enabling (provider) components to publish services with technology-agnostic interfaces, and (client) components to lookup for connectors by using service registries. In microservice architectures (MSA), services become domain-oriented and cover the whole design of an application. In addition, decoupling is pushed the farthest, by enabling inter-process communication, to make possible independent module deployment, release, and scalability. However, both the decoupling and the introduction of intermediate abstraction layers do not come without cost [3], [23]. A computational overhead arises in running an application as distinct processes and having to "pay" inter-process and network communication costs rather than simply making function calls within a process.

For several years now, we have been developing an auto-grading web application to train students in the Unix command-line interface [8]. The first version of this application has a monolithic architecture implemented using Node and Python for server-side scripts and a templating language (EJS) for the web interface, while data persistence is handled by a Postgres DB server. One of the main features of the

application is the evaluation of archive files submitted by students as answers to exercises. This feature is delegated by the Web (Node) app to Python scripts, each dedicated to an exercise. These scripts are simple in essence, performing unit tests on the content of submitted archives, but can be relatively slow as some tests require traversing files weighting several hundred kilobytes. Moreover, they are launched by making a blocking system call from the backend to a Python interpreter. Overall, when the application is used by several dozens to hundreds of students, this feature seems to constitute a bottleneck in this original monolithic version of the app. To improve the application's behavior and enable it to handle more simultaneous users, we then developed an MSA version.

Our concern was then to measure if this change of architecture was beneficial for the application. To compare its different versions, we mainly focused on the two most frequently addressed attributes [24], namely *performance* and *availability*. To that aim, we performed *load* tests, varying the number of running instances (scaling out) of the entire application, or of just some microservices. There are different interpretations of the load test concept [15], and numerous quality attributes are proposed to measure the success of an MSA implementation or migration [15], [24]. In our experiments, performance and availability were mainly measured by observing request response times and failure rates, respectively, together with measures we proposed. Overall, we considered the following research questions under different load conditions:

- RQ1: Does the extraction of the most time-consuming tasks in a worker microservice improve the performance and availability of the application compared to the monolithic one and its replicated versions?
- RQ2: How does the number of replicas in the MSA version of the application impact its performance and availability?
- RQ3: How resource (system and energy) usage evolves throughout application configurations?

The rest of the paper is structured as follows. Section II introduces the monolithic architecture of the application. Section III details the migration to MSA. Section IV explains the protocol we followed to evaluate the runtime quality of the application's versions. Section V exposes the results of the experiments to answer the research questions. Section VI discusses the related work. Section VII concludes with a discussion on future research directions.

II. The Legacy Architecture of the Studied App

The application under study aims at providing exercises to learn the Unix Command line interface. Instructors can create exercises or pick existing ones to compose sessions with starting and ending dates. During these sessions, students solve exercises and hopefully acquire skills targeted by the selected exercises. The initial architecture of the application is depicted in Figure 1. It is composed of three main components, each deployed as a Docker container:

- The Front-End serves a *Single Page Application* (SPA) based on the React library.

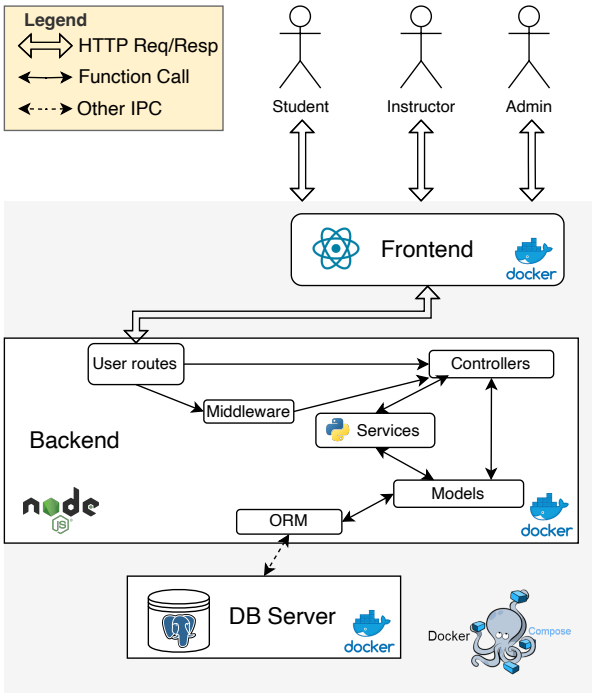


Fig. 1: The monolithic architecture of the studied app.

- The backend of the app is a monolith written with Node-Express and delivering a REST API to the SPA.
- The persistence is a Postgres server managing a database connected only to the backend.

This version of the app under study was designed according to the layered architecture style [34]. Once having loaded the frontend of the app, client browsers interact with the *routes* layer exposing the application entry points. The HTTP requests are first checked by the *middleware* layer (*e.g.*,for authentication and authorization). Then, these requests are handled by the *controllers* layer which identify destination *services* or *models*. The services interoperate with models, which in turn access the database layer through an ad-hoc ORM (Object-Relational Mapping). For historical reasons, controllers sometimes directly access models without going through services. This is settled in the recent versions of the app (following those used in this work). When a client HTTP request asks for data, upon processing completion, the backend sends JSON to the client' browser running the SPA.

The app provides a set of business services, like listing, creating, updating, and deleting `exercises`. The same kind of services (CRUD operations) are provided for managing `users`, exercise `sequences`, exercise `sessions`, etc.

The application has been used for several years. In this initial version, it supports the *simultaneous* connection of

up to roughly 50 students. However, we observed that its performance starts to degrade when we reach around 30 simultaneous users. The bottleneck seems to be the service evaluating students' productions, grading these contributions. This service performs several heavy I/O operations and blocking system calls.

III. The MSA version of the studied app

To improve the performance of the application under study, we apply the hybrid pattern [16] to turn our monolith application (hereafter denoted MON) into an MSA application. We choose functional decomposition [30] to reduce the application's bottleneck. Thus, we isolate and decouple the student production grading part from the rest of the application. In this service, the execution of Python scripts is performed using a blocking system call. This time-consuming part at runtime is not a fixed list of independent steps but rather a set of tests highly depending on the particular exercise to which a student answers. Hence, we cannot break the grading task into a fixed series of predefined steps and apply the saga pattern [18]. We rather consider each grading task as a long-running job that we entrust to a worker, applying the master/worker pattern [17]. We thus create a microservice (denoted `Exercise MS`) dedicated to grading student productions. Decoupling this service from the rest of the application allows both fine-tuned scalability [10], [5], through instantiation of parallel replicas of the service, and resilience, as a failing replica does not compromise the work of other service instances or other application components.

An API gateway service [28] is also implemented by extracting the routing part of the monolith.

The rest of the functionalities from the original monolith stayed as a single *Leftover* module (microservice) accessible from the API gateway. As a whole, this first monolith transformation step, according to the strangler pattern [29] results in a short list of microservices, as depicted in Fig. 2.

At this early stage, the MSA version of our app has a single database shared by the few MSs of our app. This contrasts with the usual MSA style that advocates resorting to the *database-per-microservice* pattern [32], [30], leading to more service decoupling, hence improving fault tolerance and maintainability. However, our work focuses on performance and more precisely on identifying whether some gain can be obtained by isolating and scaling the most time-consuming feature (RQ1)[1]. Moreover, splitting the database from the start between the different services would have implied more inter-service communication to enforce data consistency, likely masking part of the gain that could be obtained by isolating the functionality under study. We first sought an answer to RQ1 before investigating data consistency, domain boundaries, and service autonomy issues. Likewise, we are first interested in checking whether the extraction of the time-consuming part provided some benefit before further decomposing the app into more microservices.

[1]We did not find the database accesses to be as time-consuming as evaluations of the students' productions.

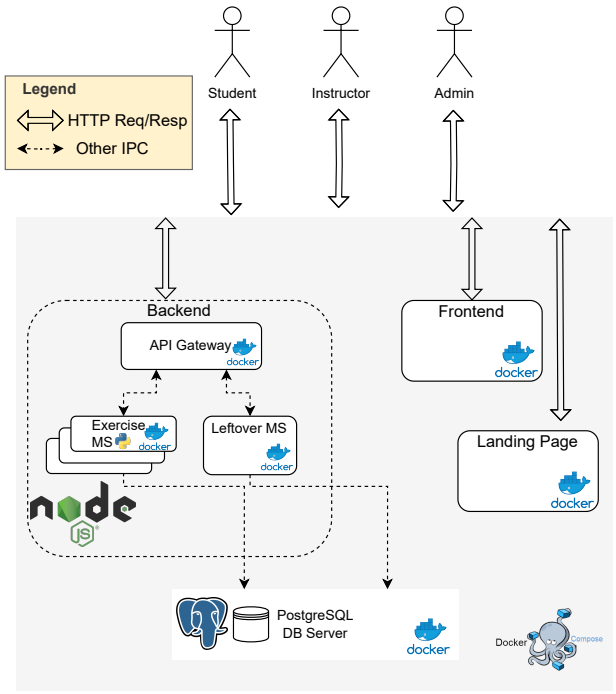


Fig. 2: The microservice architecture of the studied app.

For implementing the communication between the microservices, we have several choices depending on the nature of the communication, i.e., synchronous or asynchronous, and also to some extent depending on the format of the exchanged information, human or machine-readable. In this initial migration step, we implement a version where services communicate through RESTful HTTP calls. Moreover, there is no direct communication between microservices, all requests are routed through the API gateway.

The code of the MSA and monolith versions of the app under study is available at: https://github.com/orgs/ShellOnYou/repositories. This resource also contains links to container images for easier reproduction.

IV. Load Testing Protocol

We elaborate a protocol for load testing the backend of the application and comparing the results of the application's versions before and after the architectural migration. This protocol aims to answer the research questions (RQ1, RQ2 and RQ3) previously stated. Note that the tests go beyond the application's expected load and thus reach high failure rates in some cases, in contrast to other works that focus on pushing the application up to a *Service Level Objective* (SLO) [22], [9]. For our application, we run tests to compare the quality of two architectures (availability and performance [7]) rather than to uncover non-functional quality-related requirements under a certain load.

A. Testing environment and scenario

The testing environment is composed of the following elements: i) a dedicated server, which is used for testing. It hosts the different variants of the application each of which is deployed as a separate Docker-Compose service orchestration. Only one configuration is active at a given time, the others being shut down. To emphasize the constraints of the application during load testing, we eliminate the reverse proxy and directly subject the monolith or API gateway to the load testing. The server shows these settings: Intel(R) Core(TM) i9-10920X CPU @ 3.50GHz processor, with 12 physical CPU cores, appearing as 24 logical cores (threads) thanks to hyper-threading. The server has a Cache of 12.25MB and 128 GB RAM, it runs a Debian 12 OS with Docker 20.10.24; ii) a dedicated machine, which is used for load testing. It is equipped with the same hardware and OS configuration as the server, and it runs the Gatling load testing open-source toolset (https://gatling.io/). Both the client and app server live on the same wired network. The choice of Gatling is motivated by its rich and well-documented API. In addition, the toolset includes a recorder which enables to save usage scenarios (series of HTTP requests) by interacting with the application using a simple Internet Browser (Mozilla Firefox, in our case). This recorder generates the necessary Java/Scala code to replay scenarios. The generated code can be toggled to create a chosen number of (virtual) users during a certain time; each of these users replays the recorded scenario.

In our tests, we run a *scenario* composed of a sequence of HTTP requests that are sent to the application, and which enable to execute successively the following features:

1) *Authentication*: a POST request to `/api/user/login` the API endpoint for getting an access token and a `user_id`. If the app does not answer with an access token, the remaining requests are ignored;
2) *Checking if the user has been successfully logged-in*: a GET request to the `/api/auth/verify` endpoint;
3) *Student production submission (SPS)*: a POST request to the `/api/exercise-production` endpoint with student's data, a tarball containing the answers to a particular exercise and an archived file system that can weight up to 15KB. This submission is supposed to be saved in the database, then analyzed by a python script. A positive HTTP (201) response is expected, whose content is the feedback to the student. This request is labeled `Req-SPS`;
4) *Logout*: a DELETE HTTP request to the `/api/user/logout` endpoint;
5) *Checking if the user has been successfully logged-out*: a GET request to the following API endpoint `/api/auth/verify`.

This sequence of HTTP requests corresponds to a recurrent interaction of the users with the application, soliciting what has been noticed as a bottleneck (`Req-SPS`). We used a Gatling recorder to save this scenario by interacting with the web interface of the application, and in the generated code, we noticed that concretely there are more than 50 HTTP requests sent to the application's front/back-ends to retrieve (usually small) view elements and data. We discard most of these requests and keep only the five described above, as being the most representative. Note that each of these five requests is preceded by an HTTP OPTIONS preflight request to check if the following request's method is supported or not. This is how concrete cross-domain requests are operated. Moreover in our scenario, if the authentication request fails, all the remaining requests (which would fail) are discarded. This way, we avoid an increase in the number of failed requests and hence in the failure rate that would be artificial (*i.e.*, users that cannot log-in do not have access to internal pages and services of the app). Each pair of HTTP (OPTIONS followed by GET/POST/DELETE) requests is followed by 5 seconds pause to emulate pauses arising in real user interactions with the app. We denote by $\mathcal{R}$ this scenario of 10 requests. Note that $\mathcal{R}$ has been prepared so that no functional failure is expected. We do not consider business logic failures in our load tests.

Besides, we note here that the application's containers have been configured to be always automatically restarted by the docker engine after a failure. This has some consequences in the interpretation of the data recorded during the tests. However, since this setup is adopted for all architectural configurations, it does not introduce a bias.

B. Load testing protocol

The load tests create *virtual users* (VUs), each following the above-described $\mathcal{R}$ scenario of 10 requests. The protocol consists of running a series of identical *tests*. To obtain stable and reliable results, we repeat ten times every load test, as done in existing works [13], and we call a *run* every execution. A test is thereby a sequence of 10 runs.

We enforce pauses of 30 seconds between each run of the same test to give the application the possibility to end up ongoing requests and to return to a steady state. To obtain stable and comparable results, each run is only launched when the app is in a steady state. In addition, we remove all generated student productions in the database after each run. Following [21], to simulate different load conditions for our application, each *test* executes two successive *stages*:

1) **Ramp-up stage**: it starts by creating one virtual user every second and then linearly increases the number of users to reach 200 per second during 60 seconds.
2) **Sustained load stage**: it creates 200 virtual users per second during 60 seconds.

We set up our experiments to simulate 200 VUs per second since below that threshold, only a non-significant number of requests fail in some of the simplest configurations of the app under test. As each user issues 10 requests, these are potentially 2k requests that flood the application version under test every second, at some moments of the test. If this is still considered to be a rather low solicitation rate, it is worth reminding that the test scenario includes a request (`Req-SPS`)

that is time consuming for the server while requiring a non-negligible amount of time on the user side (thinking and practicing time to answer questions in the exercise statement). As a reference, when analyzing requests submitted to the application for the two usage weeks for which we have statistics over the last years, a `Req-SPS` was submitted every 411 seconds on average for the 37 persons who used the app during these weeks. According to these stats, 200 VUs per second in our test scenario is a highly stressful situation.

C. Selected Metrics

We evaluated some quality metrics while progressively increasing load conditions, but without reaching the breaking point where the application crashes entirely. When running load tests, we consider each request to be either:

- a **successful request**: one for which we receive an HTTP response with the expected code which can be 200, 201 or 304 for regular requests, or 401 for the last request, because after logging out, we should receive an `Unauthorized` (401 HTTP code) response;
- a **failing request**: one for which we receive a response with a 502, 503, or 504 HTTP code, which is not expected, indicating a variety of errors ranging from gateway timeouts to errors related to some services being unavailable. In addition, we consider low-level (non-HTTP) errors notified by Gatling and related to connections closed prematurely, connection establishment failures, or request/connection timeouts.

For each test, we collect the following measures, most of which are provided by Gatling:

- number of VUs – each sending 10 requests during each test theoretically (*i.e.*, if the log-in request succeeds);
- theoretical total number of HTTP requests sent to the app (denoted by N, where N= $10 \times$ VUs).
- total number of successful requests (denoted by OK);
- total number of failing requests (denoted by KO) accounted by Gatling. From this measure we define the *Failure Rate*, denoted by FR, as KO/(OK+KO);
- we also define the *request acceptance capability* (RAC) as (OK+KO)/N, allowing us to measure the availability of the app, that is its capability to accept new requests;
- effective failure rate (EFR), which corresponds to KO/N;
- ratio of successful requests with mean response time less than 800ms;
- ratio of successful requests with a mean response time between 800 and 1200ms;
- ratio of successful requests with a mean response time greater than or equal to 1200ms;
- application performance index (Apdex [33]), which is an aggregation of the previous three ratios. This index is in the range $[0, 1]$, with 0 meaning an unacceptable response time from the application and 1 meaning an excellent response time;
- throughput, which is a measure of the average number of successful requests by second.

For each run, we additionally compute and collect the following metrics:

- CPU and memory usages. These metrics have been collected using Telegraf[2] and analysed using Grafana[3], those services run on a separate server in our infrastructure, which is dedicated to monitoring the server that runs the application;
- Energy consumption is obtained using a physical power meter, which instruments the server that runs the application. This measurement is more reliable than the measure given by the CPU (using RAPL). The CPU measurement uses a specific instruction set to simulate the CPU's consumption, while a physical device uses an electronic component to monitor power.

From the description above, the scenarios we implement correspond both to *load*, but also to *performance* and *stress* tests in the sense of [21]. Some of the metrics above like RAC, FR and EFR, enable to evaluate availability quality attribute, whereas others like the different ratios of requests executed within time intervals, Apdex, CPU, memory and power usage enable to evaluate performance quality attribute.

D. Different configurations of the Application under test

We launched the load test protocol on the monolithic and microservice architectures in different configurations, considering one or several replicas of each version. To replicate the monolith, we introduce an upstream load-balancer (namely HAProxy), but keeping the database centralized to avoid redundancy and integrity problems. We denote by MON-1 to MON-12 this version of the monolith that is replicated one to twelve times. We also consider the configuration of the monolith where it has no upstream load-balancer, which we denote as *Naked* MON. When considering the MSA version of the application, we can separately replicate each of its components, namely the API Gateway, the `Exercise MS` and the `Leftover MS`. We denoted by MSA-X-Y a configuration with X, resp. Y replicas of the `Leftover MS`, resp. `Exercise MS`. We tested only one configuration where the gateway is replicated twice, namely MSA-2-4-4, having 4 replicas of both `Leftover MS`, and `Exercise MS`. Note that we consider up to 12 replicas since the load test server has 12 physical CPU cores.

V. LOAD TESTING RESULTS

The results and instructions to reproduce them are available at https://github.com/ShellOnYou/SOY_loadTests.

A. Results for RQ1

Experiments to answer $RQ1$ are designed as follows. During each run based on the $\mathcal{R}$ scenario, the total number of requests sent by Gatling to the app varied between 990K and 1.7M (roughly 10 times the number of sent Req-SPS). This variance partly originates from the *ramp-up* stage where Gatling builds a linear progression from 1 to 200 VUs per

[2]https://www.influxdata.com/time-series-platform/telegraf/
[3]https://grafana.com/

Metrics \Arch.	Naked MON	MON × 1	MON × 4	MON × 8	MON × 12	MSA-1-1
Request Acceptance Capability (RAC)	52.93%	70.22%	76.36%	**100%**	**100%**	94.08%
Failure Rate (FR)	27.26%	18.16%	10.60%	**0%**	**0%**	12.18%
Effective Failure Rate (EFR)	61.5%	42.53%	31.73%	**0%**	**0%**	17.38%
FR (Req-SPS)	95.54%	98.28%	46.79%	**0%**	**0%**	96.22%
EFR (Req-SPS)	97.76%	98.92%	62.52%	**0%**	**0%**	96.62%
Mean Response Time (MRT) $<$ 800ms	43.66%	36.16%	78.36%	96.68%	**98.39%**	85.47%
800ms $\leq$ MRT $<$ 1200ms	11.92%	11.29%	6.67%	0.66%	0.35%	7.55%
1200ms $\leq$ MRT	44.42%	52.55%	14.97%	2.66%	1.26%	6.98%
Application Performance Index (Apdex)	0.56	0.55	0.79	0.98	0.99	0.89
System Throughput (ST)	459.74	636.05	764.59	1198.91	**1202.05**	958.18

TABLE I: RQ1 results – Availability and Performance while considering Naked MON (monolith with no upstream load-balancer), MON $\times$ X (monolith with X replicas), and MSA-1-1 (1 instance of each microservice)

second. This is also due to Gatling's inability to establish all the necessary connections when the tests overload the application's versions having the smallest replica number.

Table I summarizes the results observed while evaluating the availability and the performance measures of interest.

The first observation that can be drawn from Table I is that the analyzed configurations do not have the same availability. The naked monolithic architecture can accept only half of the requests (RAC $\approx 53\%$). This is explained by the heavy load put on the app by the 200 VUs per second. At such a rate, the app is so busy that it has to discard new connections. In the configuration with the same architecture, but with an upstream load balancer (MON $\times$ 1), the availability increases (RAC $\approx 70\%$). Indeed, the load balancer, configured with a permissive timeout, acts as a buffer in front of the app. It can wait for the app to be less busy to accept requests. As fewer requests are rejected, the overall number of processed requests is increased, leading to an increase in the system throughput (by $\approx$ 38%). With such different RAC values for Naked MON and MON $\times$1, their FR values cannot be compared. But we can rely on the EFR measure introduced in the previous section. This measure accounts for the number of requests that should have been sent, compared to the score given by Gatling computed from the effective number of requests. Results show an important decrease in the global EFR. Yet, no real difference concerning the time-consuming `Req-SPS`. The number of replicas is still insufficient to handle the load. With four replicas (MON-4), the availability and performance of the monolith version continue to improve (*e.g.*, RAC, FR, ST). Interestingly, many more `Req-SPS` are now correctly processed (as seen by the significant decrease in EFR and FR). Moreover, requests are now processed more quickly (see the increased Apdex). Going up to 8 replicas allows the monolith version to accept all incoming requests (RAC$= 100\%$) and to reach a nearly optimal behaviour. By *optimal*, we mean that the configuration displays a RAC of 100% and answers nearly all requests in less than 800ms. With 200 VUs, this leads to a throughput of ≈ 1200 requests per second. Additional tests (not reported in the tables) show that a naked Node server with the same payloads exchanged with the clients but where all the applicative part has been removed, also displays a throughput of 1200. Thus for 200 VUs, at this level of replication of the Monolith, the applicative part, including the long running tasks, does not weight at all on performance.

The microservice version of the application with just one replica (MSA-1-1) reaches better results than its monolith counterparts (Naked MON and MON $\times$ 1), accepting more than 94% of the incoming requests. Here the introduction of the API gateway has the same beneficial effect as the load-balancer for the monolithic version. The gateway can absorb a large number of requests, passed to other app components after very little processing. Yet, accepting more incoming requests does not guarantee that all requests will be processed before reaching a timeout, this depends on the other services' availability. However, we observe that MSA-1-1 has lower failure rates than Naked MON and MON $\times$1 (EFR$\approx$ 17% for MSA-1-1 vs $\approx$ 62% and $\approx$ 43% for non-replicated configurations of the monolith). Thus, the architectural shift is beneficial even without considering horizontal scaling.

The failure rates (FR and EFR) for `Req-SPS` are extremely high for the monolith and only slightly better for the microservice version. This is somewhat expected as this request is very time consuming, and isolating it in a worker microservice without increasing the resources devoted to it (*i.e.*, with no horizontal scaling) is not sufficient to impede the numerous timeouts that occur for it.

Concerning response times of the non replicated configurations of the app, we can observe that the aggregated score (Apdex) is much better for the microservice version of the application (0.89 for MSA-1-1) than for the monolith version (0.56, resp. 0.55, for Naked MON, resp. MON $\times$ 1). More specifically, $\approx 85\%$ of the requests are processed in less than 800ms by MSA-1-1, while the monolith requires more than 1200 ms to process roughly half of its requests. This faster processing time explains the reason why MSA-1-1 achieves a higher RAC and lower failure rates.

RQ1. The experiments show that the introduction of the MSA is indeed beneficial for the availability and performance of the app under test even without scaling it horizontally. The benefit mostly arises from adding an intermediate routing layer (the API gateway). The same benefit can be obtained for the monolithic version by

putting it behind a load balancer that buffers incoming requests. When scaling the application horizontally, as expected, both architectural versions show improved performance and availability. However, it is worth noting that fewer replicas (hence resources) are needed for MSA to achieve an optimal behaviour, compared to the monolithic architecture.

B. Results for RQ2

Table II shows the results to answer $RQ2$. A first observation is that, when varying the number of MSA replicas from one to four (for the most time-consuming MS, i.e., *Exercise* MS), all incoming requests are accepted ($RAC = 100\%$) and the same configuration achieves 0% of failure rate.

Regarding performance, we do observe different variations. When moving from 1-1 to 1-2, response time and throughput improve. This is not surprising because the most-time consuming requests that hit one instance of the `Exercise MS` in MS-1-1 are distributed across two instances in the MS-1-2 version. When further increasing the number of replicas only for `Exercise MS` (see columns 1-X with X ranging from 4 to 12), the throughput still increases reaching near-optimal values as roughly 1210 requests are received each second by the app. However, we can see that the ratio of quickly answered requests ($<$ 800ms) is reaching a ceiling of approx. 75%. This is the expression of Amdhal's law [4]. In the current case, this is explained by the fact that the bottleneck now moves to the `Leftover MS`, preventing the response time from being improved through horizontal scaling of the most time consuming service. By increasing the number of replicas for the `Leftover MS` (see columns X-X with X ranging from 2 to 12) we go beyond this ceiling and reach 89,76% with MS-4-4 configuration. No other configuration in which we replicate the microservices only is able to give us better results. We explain this by the fact that the bottleneck has shifted in these configurations to the API gateway, which runs as a single process. This is why we ran an additional load test of the 4-4 MSA configuration with 2 replicas of the API Gateway, by adding a load balancer (namely, HAProxy) upstream. With this new configuration, that is called 2-4-4 in Table II (2 replicas of the API Gateway and 4 replicas of each MS), we reach the best results as 100% of the requests are now processed in less than 800ms. We draw the reader's attention to the scores obtained for response time $<$ 800ms, with configurations with a maximal number of replicas (1-12 or 12-12), which are lower than the scores obtained with simpler configurations (like the 95% obtained with MS-1-2, compared to 74.55% and 76.21% with MS-1-12 and MS-12-12 respectively). This is explained by the fact that MS-1-2 has a non-null failure rate; approx. 90% of Req-SPS fail. These requests take longer to execute. This is why configurations with a null failure rate show longer response times.

RQ2. Our experimental results confirm that increasing the number of replicas of microservices improves both availability and performance. Yet, only replicating the most time-consuming service improves the app behaviour only up to some point. To reach an optimal system performance, the other microservices and the API gateway must also be replicated a minimal number of times.

C. Results for RQ3

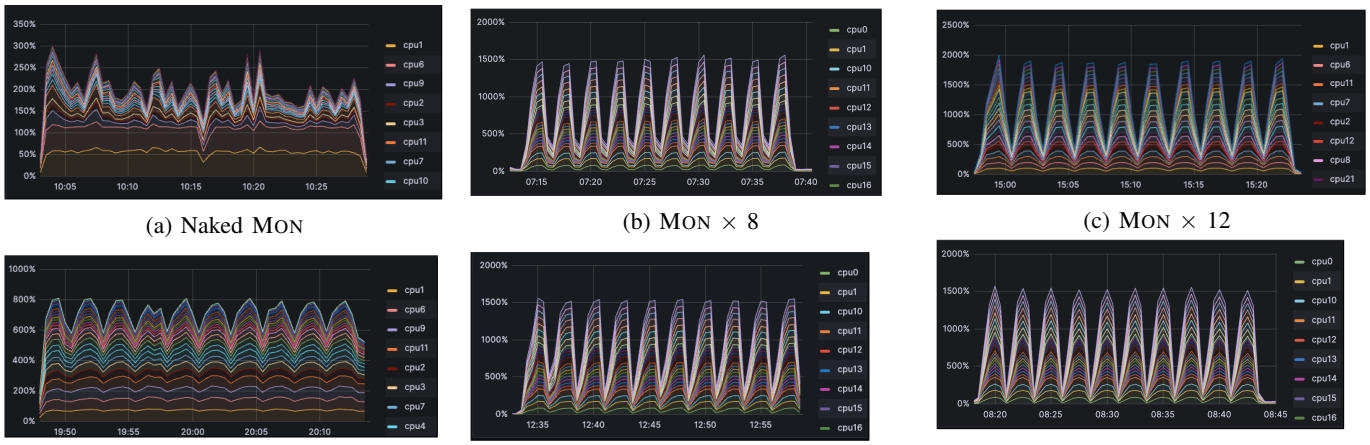
Fig. 3 contributes to answer $RQ3$. It shows the *cumulative* CPU usage due to the app (and the load-balancer when the latter is present). The server was idle and safe for the version of the app under test. The Y-axis scale adapts to the load of the CPU during the tests, some configurations (MON $\times$12 and MSA 2-4-4) mobilizing almost all available threads (24 at a maximum that is 2400% on the scale – as mentioned before, with hyper-threading the 12 cores of our CPU exist at the operating system level as 24 logical cores or threads). The 10 runs of a test appear quite clearly as peaks in the curves (more notably for replicated versions). Note that in the case of the Naked MON, the CPU usage corresponds mainly to two CPU cores, interpreted as the main app hit by the numerous requests to handle, the other being the database server solicited by payloads of around 15 KB for each `Req-SPS`. When replicating either the monolith or services of the MSA version, the total CPU usage largely increases, as more CPU cores are put to work by the load-balancers (HAProxy for the monolith and Docker-compose for the MSA version)[4]. This can be seen by the increased number of CPU cores having a non-negligible part in the cumulative curve, and by the scale of the Y axis (100% representing one CPU core at full use).

In Fig. 3, we put on each column the app configurations that are approximately equivalent in terms of availability and performance (response time). In the first column (Fig. 3-(a) and Fig. 3-(d)), we can see the most basic configurations, Naked-MON and MSA-1-1, on which we can observe different CPU usage profiles. While, as explained above, the naked-MON uses approximately 2 CPU cores, the MSA version mobilizes more CPU cores. The curves reach 800% on the Y axis, but if we remove the noisy measures related to system and Docker processes, as for Naked-MON before, we can estimate to four the number of CPU cores that are mobilized (this can be observed from Fig. 3-(d) where we have four curves distant from each other at the bottom). This corresponds to the four processes run for this configuration: API Gateway, MS-Exercise, MS-Leftover, and Postgres. In the two cases, the CPU cores do not go back to an idle state before the processes they run re-handle a new sustained load of requests. In the second column (Fig. 3-(b) and Fig. 3-(e)), we present CPU usage for configurations that reach 100% of request acceptance capability (RAC) and 0% of failure rate (EFR).

[4]As there is only one API URI for the backend, we need to add an extra layer of load-balancing (HAProxy) to be able to replicate the API. This concerns the Monolith version as well as X-Y-Z MSA when X$>$ 1.

Metrics	MSA configurations									
	1-1	**1-2**	**2-2**	**1-4**	**4-4**	**1-8**	**8-8**	**1-12**	**12-12**	**2-4-4**
Request Acceptance Capability (RAC)	94.08%	98.67%	98.77%	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**
Effective Failure Rate (EFR)	17.38%	10.72%	6.48%	**0%**	**0%**	**0%**	**0%**	**0%**	**0%**	**0%**
EFR (Req-SPS)	96.62%	89.28%	84.57%	**0%**	**0%**	**0%**	**0%**	**0%**	**0%**	**0%**
Mean Response Time (MRT) $<$ 800ms	85.47%	95.82%	93.62%	75.30%	89.76%	75.54%	83.12%	74.55%	76.21%	100%
800ms $\leq$ MRT $<$ 1200ms	7.55%	2.06%	0.79%	2.62%	3.13%	1.83%	10.52%	2.46%	13.87%	**0%**
1200ms $\leq$ MRT	6.98%	2.12%	5.59%	22.08%	7.11%	22.63%	6.36%	22.99%	9.92%	**0%**
System Throughput (ST)	958	1039	1087	**1205**	**1207**	1195	**1205**	1198	1195	**1210**

TABLE II: RQ2 results – Availability and Performance while considering the MSA configurations, columns $X - Y$ mean variant with X, resp. Y, instances of the `Leftover Ms`, resp.`Exercise Ms`; $2-4-4$ means the MSA configuration $4-4$ where additionally the gateway was doubled.



(a) Naked MON (b) MON × 8 (c) MON × 12

(d) MSA-1-1 (e) MSA-1-4 (f) MSA-2-4-4

Fig. 3: CPU usage observed for different configurations of the app under test. The Y axis scale adapts to the CPU load, 100% representing one logical CPU (thread) fully mobilized. It can be observed that the CPU load increases with the number of services / replicas deployed on the system. The Naked MON configuration is the least CPU intensive configuration, mainly mobilizing two CPU threads. The most CPU intensive configuration is MON × 12.

In the two cases, we can see cleaner curves, with evident picks and a return back to an idle state for all CPU cores. The CPU usage is approximately the same (1500% as a maximum cumulative measurement), while in one case we have 8 replicas of the monolith, and in the second we have only 4 replicas of MS exercise. The last column (Fig. 3-(c) and Fig. 3-(f)) depicts CPU usage for the best configurations identified in our load tests: MON × 12, which reaches the best average response times, and MSA-2-4-4 which makes 100% of requests be processed in less than 800ms. Here again, we have the same CPU usage profiles. Yet, the difference is in the maximum cumulative measurement, which reaches 2000% for the replicated monolith, while it is equal to 1500% for the MS version of the app. This shows that with less CPU usage, the MS version can reach optimal, and even better, results for availability and performance compared to the monolith.

Table III shows the evolution of energy (power) usage while increasing the number of replicas for both versions of the app. Memory values for a configuration are the difference between the amount of memory used by the configuration on average and the minimum memory it consumes. These are presented as ratios (percentages) to the total memory of the server (128 GB). Power consumption is measured by a physical power-meter. Results in Table III indicate that, at the beginning, the monolith outperforms the MS version. This may be explained by the monolith using less CPU on average and its memory footprint is less important[5]. What we observed is that at some level (between 4 and 8 replicas), the MS version of the app outperforms the monolith, because it reaches a good balance between CPU and memory usage.

Metrics	Architecture Variants		
	Monolith Configurations		
	Naked MON	**MON × 8**	**MON × 12**
Energy usage (w/h)	60	82	86
Memory usage (%)	0.99	1.34	1.76
	Microservice Configurations		
	1-1	**1-4**	**2-4-4**
Energy usage (w/h)	70	77	80
Memory usage (%)	1.87	1.38	1.07

TABLE III: Memory and energy usage when replicating the monolithic and microservice versions.

[5]In our experiment data, we did not notice significant differences in IO disk activity between the two versions of the application, probably because of the presence of a centralized database.

When comparing configurations having equivalent measures for the availability quality attribute, for example configurations having 100% of request acceptance capability and 0% of failure rate, which are MON × 8 and MSA-1.4 (see the middle column of Table III), the results for resource usage are in favor of the MS version: 82 vs 77 w/h in terms of energy usage. Memory usage is almost the same, 1.34% vs 1.38%. When comparing the best obtained configurations in our load tests (MON × 12 and MSA-2-4-4 – last column of the table) we can observe a net improvement: less energy and memory usage for the MS version (7% less energy usage, from 86 to 80 w/h, but a 40% decrease in memory usage, from 1.76% to 1.07%).

RQ3: The monolith outperforms MS version in energy usage when using a few replicas. The situation is reversed when reaching optimal configurations regarding availability and performance (response time). In terms of memory usage, the MS version outperforms the monolith. They require close memory quantities for a few replicas, but when increasing the replicas, the monolithic version requires more and more memory compared to MSA. This stems from MSA version running lighter processes in terms of memory footprint. Indeed, each service corresponds to only part of the monolith, and configurations we consider do not replicate all services (e.g., only one or two instances of the gateway are considered).

D. Threats to Validity

The validity of the load tests presented in this experience report may be threatened by some biases [41].

Firstly, concerning construct validity, our metrics may not cover all aspects of availability and performance quality attributes. Definitions of these quality attributes in existing quality models are quite large [7]. To mitigate this threat, we have examined all the metrics that were reported by professional tools, like the load testing tool, Gatling, and the observability platform, Prometheus/Grafana. Besides, we also report the physical measures of energy consumption (as estimations of this measure given by CPUs are controversial). Secondly, the app is centralized, being deployed on a single server. If the app was deployed in a distributed infrastructure, which is sometimes the case for large applications with an MSA, part of the results would be different, and probably additional conclusions might be derived, e.g., when analyzing network activity for instance. Since we wanted to test a specific app on a server on which we invested for an on-premise deployment, our study exclusively considers such a kind of setting with a centralized deployment.

About the internal validity, we are aware that the way we implemented our microservice version may be subject to discussion. Indeed, we did not instantiate the database-per-microservice pattern; we may thereby think that the database server may be a bottleneck in our load tests. In the different tests we run we analyzed the logs of the database container and we did not notice any connection refusal or server crash. Besides this, we see that at some level of replication, we reach a null failure rate. The database server is thus able to support the heavy workload induced by our tests.

Regarding the external validity, our experimental results consider one application, we do not have clues on the generalization of the findings. To smooth this threat, we tested different variants of our application that represent typical configurations of enterprise applications, with a monolithic architecture or microservice one, including an API gateway, a multi-container orchestration with a traditional network, and volume configurations. In many other projects that we supervise, we can observe equivalent configurations.

Finally, concerning conclusion validity, though the servers are dedicated to this app and its load test, the app is dockerized and is managed by a combination of Docker and OS processes. This introduces a slight instability in testing the app, as explained in [13]. We tried to smooth this factor by launching 10 runs of each test for all the considered configurations, and we reported the obtained average values. In our case, the hardest was to collect measures on OS resource usage, and particularly delimiting the monitoring time periods. We iterated several times on measure extraction from Prometheus/Grafana to carefully get the most precise measures. Also, we made sure that during all tests no extra process runs on the server and no SSH (or any kind of) connection was established, except those made by Grafana to monitor the app.

VI. Related Work

In the literature, the shift from monolithic to microservice-based architectures is raising attention from the research community, e.g., [14], [30], [19], [38]. However, most of the existing studies do not focus on real software, they investigate the migration targeting ad-hoc applications. This lack of real-world use cases offers intriguing research solutions but they might not address the actual needs of existing applications. Our research effort fosters the migration of an application in production from a monolithic to a microservice architecture, constrained by real-world usage (e.g., the number of users can change during execution). This evaluation of a real use case is partially covered in the literature, for instance, in [11] the banking sector is investigated though neglecting any performance evaluation, and in [36] an experience on the upkeep of a scientific application is presented. Both these papers remark that migrating to a microservices architecture is indeed not a simple task. It requires an evaluation of how communication is performed, how the application can be partitioned, and how data workflow is managed. The migration can lead to potential gains in the application, although sometimes the overhead can be quite significant. Agarwal et al. [1] exploit the implementation structure of a monolith architecture by identifying business functionalities that contribute to the recommendation of microservice architecture candidates. A foundational study on characterizing the workload in microservices is proposed in [37], and significant overhead is found due to the architectural migration. A similar investigation is conducted in [2] with a load testing scenario; the results show a distinct difference in

efficiency between monolithic and microservices architectures as the number of requests increases.

The software deployment (e.g., cloud vs on-premise) can also impact the quality characteristics, since software components can be spread across different data centers or subsections of the same data center, unlike monolithic applications that are deployed on the same node. To tackle this aspect, Jatkiewicz et al. [20] pursue a comparison of monolithic vs microservice architectures (deployed on a single node) with a focus on how vertical/horizontal scaling techniques impact the system quality, as we do in our work. Differently from [20], our experimentation runs much more extensive load tests (reaching more than 1 million requests for each run, while they run a limited number of 22k requests). Besides this, they evaluate performance only and not availability.

In the literature, there are a few works that combine tests and Microservice Architecture (MSA), as outlined in [40]. Jindal et al [22] propose a method for identifying "microservice capacities", which are maximal rates of requests that can be served by a microservice without violating service-level objectives. Camilli et al. [12] present a method for learning performance models by load testing applied to MSAs. The objective is to automate the verification of such systems against performance requirements, but the analysis does not compare different system configurations. In this paper, however, we execute load tests with varying system configurations, thus validating the tested app in terms of load capacity on a target node.

To validate a migration, we need to assess its impact, *e.g.* as investigated by Avritzer et al. [6]. These authors present a method for assessing and comparing the scalability of MSA deployment alternatives. The authors perform experiments in both a bare-metal host environment and a virtualized one. They compare results across these two environments, adjusting the allocated resources and the number of replicas. Some of their observations mirror ours in this paper, for example, "*having more Docker containers improves the performance of the services*" [6] holds for MSA in our case. We employ this kind of study to conduct our load tests and validate the behavior of our migrated application. Similar tests can be found in [25] where Lourenço et al. present a comparative analysis on the migration from a monolithic architecture to microservices that resulted in better scalability. However, this paper highlights the advantages of migration for the introduction of new features and for fixing bugs compared to monolithic architectures.

This paper pursues a detailed comparison between monolithic and microservice architectures. Normalization tests are necessary to evaluate and generalize results. This approach is presented in [39], and the authors provide a simulation toolkit to evaluate the Quality-of-Service of microservices when exposed to evolution scenarios such as auto-scaling, load balancing, and overload control. However, these tests are based on simulated workloads, and not compared to a real use case. Here, all simulations are performed using real data to mimic the day-to-day life of an application. Adhering to this pattern is crucial as real users can create unexpected events which are hard to replicate in a simulated environment. We present an in-depth experiment designed to measure and compare the performance of each architecture. Our focus specifically lies on an application that exposes a time-consuming service. Despite this specificity, we argue on the complexity when comparing different versions of applications based on load tests.

Several works have been conducted on web services. Mazlami et al. [26] present a strategy, based on a formal graph-based model, that allows the extraction of microservices from a monolithic architecture. Zhu et al. [42] formalize load balancing as an emergent property of the microservices ecosystem, round-robin scheduling policy showing better scalability than the workload aware policy (i.e., the shortest waiting queue). Song et al. [35] introduce a dataflow-based domain-specific language that includes the implicit declarations of equivalent microservices and their execution patterns.

To summarize, to the best of our knowledge, although there are efforts to compare monolithic and microservice architectures, our work advances the state-of-the-art by highlighting the impact on both availability, performance and resource usage, of some design choices, in the particular case of a real-world application exposing a time-consuming service.

VII. Conclusion

This paper presents an experience report on the migration of a monolithic web app that includes a feature generating long running tasks. We migrated this monolith to a microservice version and conducted load tests. We considered several configurations for the monolith and its microservice version, with up to 200 virtual users per second, each performing 10 requests (e.g., high loads considering the time consuming tasks).

Results provide the following findings: (i) scaling up the app horizontally is beneficial for both the monolith and the MSA, even if the monolith requires a larger number of replicas to reach a given level of performance and availability; (ii) the migration is worthwhile only when the user requests saturate the monolith that instead enjoys a lower energy consumption, besides sparing the migration's effort; (iii) under heavy load, the replication of the most time consuming parts of the MSA might be not sufficient to reach a given level of performance and availability, (iv) the replication of the API gateway benefits the MSA by buffering incoming traffic; a similar effect is achieved for the monolith by placing it after a load balancer.

Our load testing results make evident the complexity behind the migration process. Thus, in terms of lessons learned, we can conclude that the migration is subject to the applications' peculiarities (e.g., long-running jobs, routing infrastructure, load and communication between services), but the service replication strategy has a significant impact on performance, availability, and energy usage.

Future work includes comparing the MSA version described in this work with two further versions of the app: (i) replacing the REST/HTTP communication between microservices with a message broker; (ii) replacing the file storage (currently in the database) with a dedicated high-performance file storage service. The goal is to evaluate the impact of these different design choices on the quality of the application under analysis.

VIII. Acknowledgment

We thank Lysa Matmar for her help in making available the code, the applications' containers, and the load tests. This work has been partially funded by the MUR-PRIN project DREAM ($20228FT78M$), the MUR-PRO3 project on Software Quality, and the MUR-PNRR project VITALITY ($ECS00000041$).

References

[1] Shivali Agarwal, Raunak Sinha, Giriprasad Sridhara, Pratap Das, Utkarsh Desai, Srikanth Tamilselvam, Amith Singhee, and Hiroaki Nakamuro. Monolith to microservice candidates using business functionality inference. In *International Conference on Web Services (ICWS)*, pages 758–763, 2021.

[2] Omar Al-Debagy and Peter Martinek. A comparative review of microservices and monolithic architectures. In *IEEE International Symposium on Computational Intelligence and Informatics (CINTI)*, pages 000149–000154, 2018.

[3] Marcelo Amaral, Jordà Polo, David Carrera, Iqbal Mohomed, Merve Unuvar, and Malgorzata Steinder. Performance evaluation of microservices architectures using containers. In *IEEE International Symposium on Network Computing and Application (NCA)*, pages 27–34, 2015.

[4] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. AFIPS '67 (Spring), New York, NY, USA, 1967. Association for Computing Machinery.

[5] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[6] Alberto Avritzer, Vincenzo Ferme, Andrea Janes, Barbara Russo, André van Hoorn, Henning Schulz, Daniel Menasché, and Vilc Rufino. Scalability assessment of microservice architecture deployment configurations: A domain-based approach leveraging operational profiles and load tests. *Journal of Systems and Software (JSS)*, 165:110564, 2020.

[7] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice, 4th Edition (SEI Series in Software Engineering)*. Addison-Wesley Pro., 2021.

[8] Vincent Berry, Arnaud Castelltort, Chrysta Pelissier, Marion Rousseau, and Chouki Tibermacine. Shellonyou: Learning by doing unix command line. In *Proceedings of the 27th ACM Conference on on Innovation and Technology in Computer Science Education Vol. 1*, ITiCSE '22, page 379–385, New York, NY, USA, 2022. Association for Computing Machinery.

[9] Betsy Beyer, Niall Richard Murphy, David K Rensin, Kent Kawahara, and Stephen Thorne. *The site reliability workbook: practical ways to implement SRE.* " O'Reilly Media, Inc.", 2018.

[10] André B Bondi. Characteristics of scalability and their impact on performance. In *International Workshop on Software and Performance (WOSP)*, pages 195–203, 2000.

[11] Antonio Bucchiarone, Nicola Dragoni, Schahram Dustdar, Stephan T Larsen, and Manuel Mazzara. From monolithic to microservices: An experience report from the banking domain. *IEEE Software*, 35(3):50–55, 2018.

[12] Matteo Camilli, Andrea Janes, and Barbara Russo. Automated test-based learning and verification of performance models for microservices systems. *Journal of Systems and Software (JSS)*, 187:111225, 2022.

[13] Simon Eismann, Cor-Paul Bezemer, Weiyi Shang, Dušan Okanović, and André van Hoorn. Microservices: A performance tester's dream or nightmare? In *ACM/SPEC International Conference on Performance Engineering (ICPE)*, page 138–149, 2020.

[14] Chen-Yuan Fan and Shang-Pin Ma. Migrating monolithic mobile application to microservice architecture: An experiment report. In *International Conference on AI & Mobile Services (AIMS)*, pages 109–112, 2017.

[15] Felipe Febrero, Coral Calero, and M Ángeles Moraga. Software reliability modeling based on iso/iec square. *Information and Software Technology*, 70:18–29, 2016.

[16] Ken Finnigan. *Enterprise Java Microservices*. Manning Publication, 2018.

[17] Eric Freeman, Susanne Hupfer, and Ken Arnold. *JavaSpaces Principles, Patterns, and Practice*. Addison-Wesley Professional, 1999.

[18] Hector Garcia-Molina and Kenneth Salem. Sagas. In *ACM SIGMOD International Conference on Management of Data*, page 249–259, New York, NY, USA, 1987. Association for Computing Machinery.

[19] Jean-Philippe Gouigoux and Dalila Tamzalit. From monolith to microservices: Lessons learned on an industrial migration to a web oriented architecture. In *International Conference on Software Architecture Workshops (ICSAW)*, pages 62–65, 2017.

[20] Przemysław Jatkiewicz and Szymon Okrój. Differences in performance, scalability, and cost of using microservice and monolithic architecture. In *ACM/SIGAPP Symposium on Applied Computing (SAC)*, pages 1038–1041, 2023.

[21] Zhen Ming Jiang and Ahmed E. Hassan. A survey on load testing of large-scale software systems. *IEEE Trans. on Software Engineering*, 41(11):1091–1118, 2015.

[22] Anshul Jindal, Vladimir Podolskiy, and Michael Gerndt. Performance modeling for cloud microservice applications. In *ACM/SPEC International Conference on Performance Engineering (ICPE)*, page 25–32, 2019.

[23] J. Lewis and M. Fowler. Richardson Maturity Model. http://martinfowler.com/articles/microservices.html, 2014.

[24] Shanshan Li, He Zhang, Zijia Jia, Chenxing Zhong, Cheng Zhang, Zhihao Shan, Jinfeng Shen, and Muhammad Ali Babar. Understanding and addressing quality attributes of microservices architecture: a systematic literature review. *Information and Software Technology*, 131:106449, 2021.

[25] João Lourenço and António Rito Silva. Monolith development history for microservices identification: a comparative analysis. In *International Conference on Web Services (ICWS)*, pages 50–56. IEEE, 2023.

[26] Genc Mazlami, Jürgen Cito, and Philipp Leitner. Extraction of microservices from monolithic software architectures. In *IEEE International Conference on Web Services (ICWS)*, pages 524–531, 2017.

[27] M. D. McIlroy. Mass-produced software components, software engineering concepts and techniques. In *NATO Conference on Software Engineering*, 1968.

[28] Fabrizio Montesi and Janine Weber. Circuit Breakers, Discovery, and API Gateways in Microservices, 2016. ArXiv preprint available here: https://arxiv.org/pdf/1609.05830.pdf.

[29] Sam Newman. *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith*. O'Reilly Media, Inc, USA, 2019.

[30] Sam Newman. *Building Microservices: Designing Fine-Grained Systems, 2nd Edition*. O'Reilly Media, Inc, USA, 2021.

[31] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, dec 1972.

[32] Chris Richardson. *Microservices patterns: with examples in Java*. Simon and Schuster (S&S), 2018.

[33] Peter Sevcik. Defining the application performance index. *Business Communications Review*, 20:8–10, 2005.

[34] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Pearson, 1996.

[35] Zheng Song and Eli Tilevich. Equivalence-enhanced microservice workflow orchestration to efficiently increase reliability. In *International Conference on Web Services (ICWS)*, pages 426–433, 2019.

[36] Leonardo P. Tizzei, Leonardo Azevedo, Elton Figueiredo de S. Soares, Raphael Thiago, and Rodrigo Costa. On the maintenance of a scientific application based on microservices: an experience report. In *International Conference on Web Services (ICWS)*, pages 102–109, 2020.

[37] Takanori Ueda, Takuya Nakaike, and Moriyoshi Ohara. Workload characterization for microservices. In *International Symposium on Workload Characterization (IISWC)*, pages 1–10, 2016.

[38] Victor Velepucha and Pamela Flores. A survey on microservices architecture: Principles, patterns and migration challenges. *IEEE Access*, 2023.

[39] Teng Wang, Xiang He, Haomai Shi, and Zhongjie Wang. Evolutionsim: An extensible simulation toolkit for microservice system evolution. In *International Conference on Web Services (ICWS)*, pages 43–49, 2023.

[40] Muhammad Waseem, Peng Liang, Gastón Márquez, and Amleto Di Salle. Testing microservices architecture-based applications: A systematic mapping study. In *Asia-Pacific Software Engineering Conference (APSEC)*, pages 119–128, 2020.

[41] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, and Björn Regnell. *Experimentation in Software Engineering*. Springer, 2012.

[42] Hong Zhu, Hongbo Wang, and Ian Bayley. Formal analysis of load balancing in microservices with scenario calculus. In *International Conference on Cloud Computing (CLOUD)*, pages 908–911, 2018.