# Breaking down monoliths with Microservices and DevOps: an industrial experience report

Danilo Pianini
*Alma Mater Studiorum—Università di Bologna*
Cesena, Italy
danilo.pianini@unibo.it – 0000-0002-8392-5409

Alessandro Neri
*Maggioli S.p.A.*
Santarcangelo di Romagna, Italy
alessandro.neri@maggioli.it – 0000-0002-9649-8880

***Abstract*—Recent trends in software production fostered the adoption of microservice architectures, where a product is the result of the coordinated execution of several loosely coupled autonomous services, thus promoting modularity, scalability, and integration of legacy products by wrapping. This architectural style also promotes parallel development, as different teams can be in charge of different services; however, this parallelization is at first sight at odds with the established practice of continuous integration: a change to a single service may cause cascading effects that the local testing cannot capture, and the overall functionality may thus get compromised even though all services apparently work. In this paper, we report an experience of a successful and thorough implementation of DevOps techniques into a large business, carried out by a relatively small team. We discuss the steps taken to build a continuous integration pipeline performing system-wide quality assurance, the development practices that enable such a pipeline to be effective, and the lessons learned by applying these practices in a digital publishing industry setting.**

***Index Terms*—Microservice architectures, DevOps, Continuous Integration, Parallel development, Distributed testing, Continuous delivery, Experience report**

## I. Introduction and background

The evolving world of Internet-based services provided by enterprises is pushing towards architectures that favor composability, scalability, and isolation. Microservice architectures, where applications are decomposed into small-sized, (semi-)autonomous, and isolated (usually containerized) services are a quickly rising trend, being adopted not just by large software enterprises, but also by smaller players [8], [14], [27]. There are several reasons behind the growing success of microservice architectures. First, microservices can "wrap" existing services, and thus provide a way to a gradual modernization of existing services. Second, as their name suggests, they foster the decomposition of monolithic applications into multiple isolated and coordinated services, promoting the architectural application of the Single Responsibility Principle (SRP). Third, they allow for higher scalability and resilience, as microservices can usually be deployed in multiple copies with a relatively small effort, and can be configured to scale horizontally as requests increase, and be restarted in case of failure. Fourth, the maintenance and evolution of many small services are radically different from the one of a large software monolith: components may be evolved separately, adopting agile development techniques.

The trend towards microservice architectures harmonically intertwines with the growing interest in the DevOps philosophy. The DevOps philosophy challenges the traditional way systems are developed, promoting shared responsibilities and communication across teams, usually instead organized vertically (in so-called silos); starting from the separation between development and operations [9]. Several slightly different definitions of DevOps are found in the literature [15]; yet, they agree on the goals and on the means to be used to pursue these goals: DevOps is meant to reduce the *time* between a change in the codebase and its actual reification in the production environment, without any loss (actually, with an increment) in quality [28]; this is achieved through four principles [16]: collaboration, automation [9], measurement, and monitoring. DevOps practices originated in the agile community and are often seen as an evolution of agile practices [11], [17], focusing on automation and on keeping the software always in a working state by continuous integration [25]. There is evidence in the literature that these techniques improve the software quality [20], increase collaboration [7], and raise the perceived job satisfaction [12].

Of course, microservice architectures and DevOps philosophy work rather well together: in principle, an application could be microservice-ified even with a traditional development process, and DevOps techniques can be applied to monolithic software products; however, the peculiar traits of the two shine when combined [4]. In fact, on the one hand, DevOps practices simplify microservice-ification and maintenance of microservice-based systems; on the other hand, the complex orchestration of such services generates an elaborate sequence of building, verification, and deployment tasks, which fully highlights the benefits of the DevOps approach.

In this work, we report an experience of microservice-ification of an existing industrial software product, with a strong focus on the DevOps practices that enabled the transformation, and their effect on the quality of software and the process. The remainder of this paper is structured as follows: Section II introduces the original software system and the development practices that were applied before the documented action; Section III discusses how the initial monolithic software got partitioned into microservices, the new DevOps workflow the related automated procedures, and the project's timeline; Section IV measures the impact of the adoption of

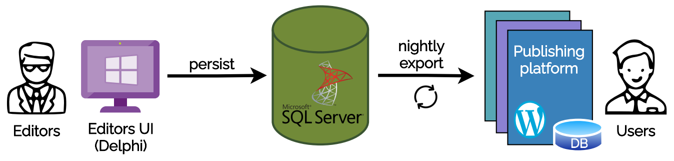

Fig. 1. Original architecture of `sisred`.

microservices and DevOps, drawing lessons for the future; finally, Section V concludes the work.

## II. Legacy system analysis and requirements

This manuscript summarizes the experience gained by evolving an existing system's architecture to be microservice-based and the related adoption of DevOps practices. The target system is an editorial management platform codenamed `sisred` (a shorthand for the Italian "Sistema Redazionale"—roughly equivalent to "editorial system"), developed and maintained at Maggioli S.p.A.[1], a large multinational editorial company based in Italy, counting about 2000 active collaborators at the time of writing[2].

### A. *Original architecture and implementation*

The original software was a classic client-server application geared towards professionals in law. Editors are domain experts paid by Maggioli that act as information sources and are provided a local installation of a graphical client allowing them to add editorial products to a central database. Typical editorial products are national or regional laws, legally binding documents from public institutions (e.g., ministries), and judicial sentences. Changes are then propagated towards end destinations, usually WordPress-based websites or search engines, portals for which Maggioli requires paid access. Propagation can be triggered manually by editors but is usually left to a nightly cronjob. The reference architecture is depicted in Figure 1. Implementation of the graphical client is in Delphi, while the export logic is written in Windows batch files.

### B. *Organization and operations*

In this section, we report the results of an interview performed before the transition to microservices and DevOps, regarding several aspects of the development process: team structure, versioning, parallel development, testing, delivery, and issue tracking.

The operations and maintenance team of `sisred` counted just five members: one project lead, one quality assurance manager, and three developers (one senior backend developer, one full-stack developer, and one frontend developer). The project lead was responsible for organizing all the activities revolving around the project. There was no archive of activities, no shared platform for task listing and assignment, and no formal way for developers to contribute bottom-up to the definition or evolution of development priorities. All activities were organized manually by the team lead, and they came from two main sources: direct email requests and tickets reported via a third-party service[3], which were then transcripted to emails. There was no direct correspondence between issues/emails and changes in the codebase. Despite the process being improvable, none of the team members complained specifically about the team organization practices; team members identified as the reason the good communication, favored by the small team size and the shared physical workplace. A minor complaint was the company-wise mandatory use of Skype as the only authorized remote communication tool, but the directive was removed as part of the COVID-19 pandemic reorganization response.

The application was partially versioned. The Delphi client was versioned inside a Subversion (SVN) repository [22], with no branching policy: all developers worked in parallel on the `trunk` branch. There was no commit message policy, and version numbers were assigned incrementally (a single version number, increasing at each commit). The team clarified that the original versioning system was not used in reality as a versioning system, but rather as a source code backup. Parallel work was a source of issues, as a few days of local development could require several hours of integration work. Changes to the database schema were not included in versioning; once written, they were manually exercised into a testing environment and, if found to be working, reified in production.

Local development happens with two main tools: the Delphi IDE for the development and (manual) testing of the user interface, and Microsoft SQL Server Management Studio for managing a local instance of SQL Server 2000. By interviewing the development team, we discovered that there was a history of compatibility issues between specific version combinations of Microsoft SQL Server and the Windows operating system, that hindered development on several occasions: different operating system versions may show inconsistent behavior, making issues hard to reproduce or, worse, fail to intercept issues that may get into production.

All testing was performed manually, in the aforementioned local instance of SQL Server 2000: changes to the database architecture or stored procedures were implemented locally and, in case of manual testing success, replicated in production. There was no form of test automation, neither for the user interface nor for the operations on the database, use cases and manual testing procedures were not formalized in any way and depended on the developer in charge of the testing procedure; moreover, the team also did not track past issues and did not perform regression testing. Testing perceived as potentially disruptive was performed on a copy of the production environment (staging environment). In this case, the team was required to go through a procedure that required external interaction: a ticket must be opened to the IT/operations team, and the latter usually provided an instance of Microsoft Windows Server and Microsoft SQL Server in a

[1]https://www.maggioli.it/
[2]https://archive.is/bPuaZ
[3]https://osticket.com/

few days. A clear dissatisfaction with the overall procedure emerged from the interview; most complaints were related to communication issues and misunderstandings that resulted in avoidable waste of time, which caused the staging environment creation to be considered a bottleneck in the development process.

The delivery protocol, although formalized, was largely manual: (*i*) changes were first committed on the SVN repository; (*ii*) an executable file was created from the IDE and assigned a version number manually; (*iii*) the database on the staging environment was manually updated (schema and stored procedures); (*iv*) the export logic was manually updated on the testing environment; (*v*) a manual test procedure was performed on the staging environment, including export logic and user interface functionality checks; (*vi*) once everything was deemed working, the external operations team was contacted to move staging into production; (*vii*) once production was ready, updates to the desktop application were delivered to editors via email. The general feeling on the process was that much time was wasted in repetitive operations; in particular, sending updates via email was perceived as cumbersome. The most critical aspect was the interaction with an external operations team, which can be a bottleneck and slow down the delivery process.

## C. *Desiderata and constraints*

Requirements for the system refactoring were gathered by interviewing editors and company managers. The two categories of stakeholders expressed different concerns.

Company managers were mainly interested in lowering the time-to-market of updates and novel features. Another concern that emerged is a reduction in the service downtime: in its original state, the system was not available while performing nightly exports towards target publishing platforms, which translated into a downtime of about 30 minutes every night. The problem was perceived as minor, as the system only targeted the Italian internal market (which does not span multiple time zones), and can thus tolerate service interruptions during non-working hours. However, downtimes critically interfered with the mid-term company goal to evolve `sisred` into a stand-alone product to be sold to other companies. Under this point of view, two further optional requirements emerged: first, the system should be able to scale out and be adaptable to different loads; second, the system or part of it should be deployable on external services (cloud-ified), for better elasticity and cost control. Company managers also imposed some technological constraints; source control must remain entirely internal, and must not be sent to external servers; requests for cloud services must be satisfied through an existing Microsoft Azure subscription; any new technology should preferably be open source and free to use.

Editors had a primary concern that they made very clear: they needed the entire feature set of the original product, with no regressions. Changes that implied a period (even transitory) in which the system was working sub-par were not welcome. From a non-functional point of view, they expressed

| **Service** | **Implementation status** | **Technologies** |
|---|---|---|
| Web Client | to be developed by the team | HTML, Javascript, AngularJS |
| API Gateway | to be developed by the team | Java, Spring Boot, Azure SQL, Elasticsearch, Azure Blob Storage |
| IDP | existing as service, to be integrated | Keycloak Server |
| OCR | existing as service, to be integrated | Apache Tika Server, Tesseract 4 |
| NLP | to be developed by the team | Java, Spring Boot, Apache OpenNlP |
| DC | existing, to be converted to service and integrated | Java, Spring Boot |
| Pdf2Html | existing, to be converted to service and integrated | Python, C, pdf2htmlEX |

TABLE I
SUMMARY OF THE SERVICES COMPOSING THE MICROSERVICE-IFIED VERSION OF `SISRED`. THE FINAL DISTRIBUTED APPLICATION IS LARGELY OBTAINED BY INTEGRATING EXISTING SERVICES.

the desire for the system to be accessible from devices other than personal computers and operating systems other than Microsoft Windows. Most other requirements were related to functional aspects of the system, which can be summarized as follows: immediate publishing of novel contents, without nightly synchronization; automatic classification of documents; automatic retrieval of publicly available contents (e.g., from the Italian “Gazzetta Ufficiale,” where laws are officially published); support for multimedia files; a search function that does not require to pre-filter the content kind; an optical text recognition system allowing for automatic import of printed documents; a referencing system allowing to hyperlink documents; and support for anonymization of documents containing sensitive data.

## III. Microservice-ification and DevOps

The team approached the problem by applying a set of techniques known with the name of Domain-Driven Design (DDD) [10], which has been reported to be particularly successful in building microservice architectures [18], [21], [23]. Very succinctly, DDD mandates software architects to understand the domain at hand, reify such domain into software entities, and design the software around these. Understanding the domain model requires interaction with a domain expert. In the case we are presenting, the team team discussed and collaborated with two editors with over a decade of experience.

## A. *Rethinking the architecture with microservices*

Crucially, following the identification of the core domain and sub-domains, a strategic pattern in DDD suggested identifying logic boundaries among sub-contexts, the so-called bounded contexts. We found bounded contexts to be key in decomposing the old monolithic system into several small-sized and encapsulated services, as they provide a natural segmentation of the original domain, which in turn provide reasonable boundaries for encapsulating services and exposing clean APIs for inter-service coordination, easing the reification of the single responsibility principle. From the analysis, the

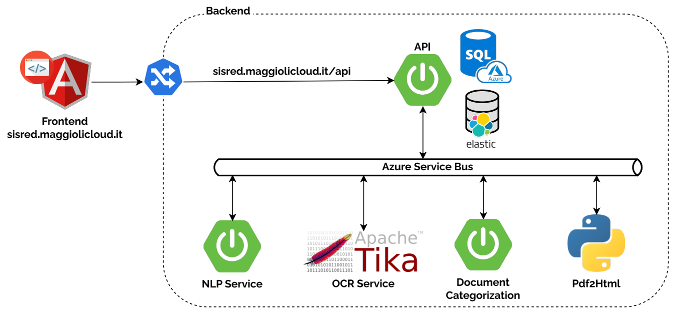

Fig. 2. Microservice-ified architecture of `sisred`.

following services emerged as a reasonable segmentation of the original domain model:

- **Web Client**: replaces the Delphi client as the frontend for editors;
- **API Gateway**: a single access point to the backend, meant to work as a broker, dispatching frontend requests to the appropriate implementing backend service;
- **Identity Provider (IDP)**: a stateful service responsible for authentication and session persistence of users;
- **OCR**: translates images with text into editable documents;
- **NLP**: Natural Language Processing service, scanning text document in search of potential references (e.g., to laws) and sensitive data (e.g., emails, names, social security codes, etc.);
- **Document Categorization (DC)**: classifies text documents based on their content, proposing a categorization;
- **Pdf2Html**: creates an HTML representation of a PDF document, preserving contents and look.

Among these services, Web Client, API Gateway, and NLP needed to be implemented from scratch by the team; IDP must reuse the existing authentication service implemented by Maggioli; while the remainder of services had been developed in the past within the company, and needed to be integrated with the new system. Table I summarizes the services, their initial status (whether they needed to be developed from scratch, existed as a stand-alone application, or existed as a service but needed integration), and the main technologies involved. Figure 2 shows the final architecture of the system: editors access a web frontend, and all backend services communicate through a shared bus.

### B. Agile Workflow

Along with the new shape of the software, the team also reshaped workflow and roles. The main goal of the reorganization was to enable each team member to effectively operate on any of the system components, consistently with the “no silos” philosophy at the base of DevOps [7]. Reorganization followed agile practices, which are considered to be foundational in DevOps approaches [17].

The team worked with two-week sprints, at the end of which a meeting with domain experts was held; internally, the team agreed to have a stand-up meeting every two days. A good agile workflow is usually supported by tools that allow the implementation of progression tracking for tasks, bug fixes, and new features. This tracking should ideally be linked to the actual progression of code, whose development is meant to be carried out in parallel, and whose progresses should be assessed via code review. For these reasons, and following recent trends in modern version control [2], [6], the team ditched Subversion in favor of a distributed version control system, the specific tool of choice was git [26]. By itself, git does not provide any first-level abstraction for tasks or issues, nor does it provide direct support for code reviewing. These operations are usually provided upon git by repository management tools, such as GitHub and Gitlab. Since the company already had internal experience with the latter, it was selected as the repository management tool.

Within a single sprint, developers can self-assign tasks and assume the role of activity manager for a specific task, bug fix, or feature, for which they will be in charge of performing a code review screening contributions made by all team members. Preferential means of intra-team communication are direct oral communication for information that does not require persistence, and Slack for written, persistent, or remote communication. This horizontal organization requires intense communication to be effective, and it thus well fits small-sized and geographically co-located teams, while it is unclear how it would perform on larger or geographically widespread teams.

### C. Branching model

One issue with the previous organization of the system concerns the relationship between the current state of the code on the repository, the environment where the system running at that version was deployed, and the missing correspondence between the two. As the first step towards a DevOps approach to development, the team decided that it was paramount to have a clear binding between the code state and its target environment. The team identified four key environments.

1) **Local**: the configuration of each development machine. This environment has cardinality at least equal to the number of developers. In this environment, the focus is on finding a good trade-off between management complexity (installation, maintenance, performance, etc.) and similarity to the actual production environment.
2) **Feature Review**: a sandboxed environment for integration testing. New features may impact multiple services at once, and thus they require a sandboxed copy of the production environment. There should be one such environment for each feature under active development. This kind of environment is also to be leveraged for producing agile demos of novel features, to be showcased to people other than team members.
3) **Staging**: pre-production environment, that should replicate the production environment as closely as possible. It is the last step before a new version of the system enters production. There should be a single copy of such an environment.

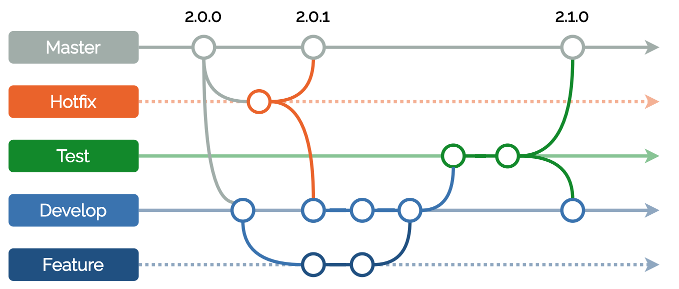

Fig. 3. A pictorial representation of the selected branching model. Solid horizontal lines depict "living" branches; `master`, `test`, and `develop` are always alive, while `hotfix/` and `feature/` branches are ephemeral.

4) **Production**: the working system, where editors can log in and use the provided services. There should be a single instance of this environment.

These environments got mapped to different *branches* in the git repository. In particular, the team adopted a customized version of the well-known "git-flow" branching model[4], with three long-lived branches (`master`, `develop`, and `test`) and two kinds of short-lived branches (`feature/` and `hotfix/`). `master` always hosts the latest stable version, and its code is in execution in the *production* environment. `test` is similar to release branches in classic git-flow, but instead of being a class of short-lived branches is a single long-lived branch, whose current state is reified into the *staging* environment; releases are created by merging the code from `test` into `master`. The `develop` branch represents ongoing development, and is not mapped to any environment; when code on `develop` is in a good state, it gets merged into `test`. The `hotfix/` branches are ephemeral branches meant to fix severe bugs that hit production; they originate from `master` and, once the bug is solved, get merged into both `master` (creating a new release) and `develop`. Finally, the `feature/` branches host most of the actual development activity, they originate from `develop` and get merged into the same branch once development is complete; crucially, for every `feature/` branch, there is a corresponding *feature review* environment that hosts a running instance of the feature code.

Figure 3 depicts the selected branching model. Operationally, every activity tracked by the repository management system is reified as a `feature/` branch. Once an activity is complete, a code review round is required before merging the corresponding `feature/` on `develop`; the operation is performed via a Gitlab merge request (equivalent to a pull request in other repository management systems), where the activity manager's positive review is required for the merge to happen. A similar process would apply to other merges: from `develop` to `test`, and from `test` to `master`. However, the team decided that a single code review phase was sufficient, and the merges mentioned above are thus performed without a merge request; further checks could be introduced in the future.

[4]https://archive.vn/tQ8wp

An important choice the team had to make was whether to have a single repository with the code of all services or split the system into multiple repositories Since every microservice is autonomous and may use different technologies, the most natural solution is to have one repository per microservice. However, this approach comes with some downsides: changes to a service's API may require adaptation of other coordinated services, and in a multi-repository setup this leads to time frames in which the development lines of different products are inconsistent. Also, a single repository makes it easier to organize a single consistent continuous integration pipeline (see the next section), particularly for the implementation of automated end-to-end testing procedures, which require elaborate environment setup operations. There is a hybrid solution that was not explored between the two options: one repository per microservice, and a single main repository importing all the others as submodules. Submodules are a git feature that allows repositories to be imported as part of other repositories, and can express recursive dependency relationships among repositories [1]. Such a solution has great potential, but slightly complicates the repository management; the team thus decided to use a single repository for the whole `sisred`, at least for the moment: the solution is known and might be adopted in the future, should for instance the evolution of some modules outpace the rest of the product, or should some services be required in multiple projects.

## D. Automated pipeline

One pillar of DevOps is automation [9], [16], namely the ability to minimize repetitive operations. Automation in DevOps is usually applied throughout the software lifecycle: dependency resolution and retrieval, compilation, static code analysis, packaging, API documentation generation, testing, containerization, delivery, and deployment are all (to varying degrees) subject to automation. A good DevOps system should feature an automated pipeline that, for each change in the codebase, performs all the automatic checks, ideally up to the deployment of a copy of the whole system. One goal of the development team is to keep the production and development versions of the software always "in the green," namely, passing all quality assurance and deployment requirements. An automated pipeline is a de facto enabler for the practice of continuous integration [25], namely the practice to merge the work of all developers frequently. For this practice to apply successfully, in fact, all integrations need to be verified, and such a verification needs to be performed automatically.

In our reference project, considering the team size, developers cannot be in charge of manually maintaining the correspondence between branches and environments defined in Section III-C: the operation needs to be automatic to be sustainable. The pipeline realized by the team, depicted in Figure 4, is triggered by pushes to the main copy of the repository or by the opening or update of merge requests. It considers the whole repository and for each service, it prepares a job to

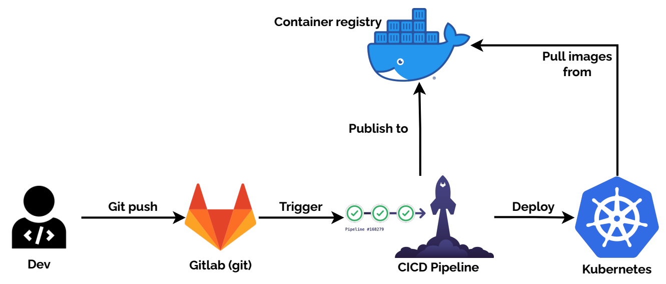

Fig. 4. A pictorial representation of the automated CI/CD pipeline

.

be executed on an isolated worker instance, which builds the service, performs quality assurance (static code analysis and unit testing), and creates a containerized version of the service, which is then delivered to an internal container registry. Once all microservices are ready and delivered, the pipeline relies on Infrastructure as Code (IaC) [13] (specifically, Terraform[5]) to prepare a deployment infrastructure. IaC is the practice of managing infrastructural resource provisioning automatically through declarative specifications, that can thus be versioned and be part of the DevOps automation infrastructure [3]. In our case, the IaC tool is instructed to instance all infrastructural components (databases, Azure Service Bus, Azure Monitor...) and inject credentials into a Kubernetes-managed cluster on which the microservices will run once deployed. Once the infrastruture is set up, the actual deployment is perfomed. Notably, the entire operation requires no human intervention at all. Once a new commit reaches the reference copy of the repository, a running version of the system is deployed in the appropriate environment in minutes—unless the system fails quality control, in which case developers are alerted that the system needs intervention.

## E. *Multiphase Automated Quality Assurance*

In the previous section, we mentioned quality assurance several times, as part of the automated pipeline leading from a development act (push of a commit or change – including creation – of a merge request) to a deployed version of the system. In this section, we summarize the discussion the team had regarding quality assurance, and the consequent decisions, as well as their implementation.

First and foremost, quality assurance in software (intended at large, not just in microservices) embraces multiple phases of the software production lifecycle: static analysis can be performed on source code and (for compiled languages) on binaries; testing happens on runtime at different scales and degrees of integration (unit, architecture, integration, end-to-end); monitoring systems in production allows to assess their health, quality of service, and intercept issues that involve the infrastructure; finally, security is a cross-cutting concern, that should be considered in each of the aforementioned phases. Of course, comprehensive quality assurance requires time, and it is, in general, a repetitive operation; as such, it is a suitable target for DevOps tools and automation [24].

*1) Static Analysis:* is performed first, immediately after a successful compilation phase. Failures in this phase prevent subsequent phases (including runtime testing), such a policy was created to avoid wasting resources for verifying the runtime behavior of software that would not end up in production in any case. Static analysis checks were classified into three domains: maintainability, reliability, and security. Maintainability checks primarily intercept code smells, style issues, and architectural coherence. Code smells may be a source of reliability or performance issues, and they hinder the understandability of code, potentially inducing developers to introduce further code smells. Similarly, a consistent style helps readability, understandability, improves the quality of differential code comparisons, and overall promotes quicker and more frequent merges [29]. Architectural coherence statically verifies that architectural rules are not violated, e.g., that there are no references to namespaces that should be hidden. Reliability checks scan for bugs and suspicious practices on specific languages by detecting well-known technology-specific programming anti-patterns (for instance, performing string comparison using the == operator in Java, which fails when the compared strings are equal but not internalized). Security controls are classified into two categories: vulnerabilities, which are detected by searching for well-known patterns, cause a failure of the check, and mandate a fix; and hotspots, which are security-sensitive pieces of that, although not matching any vulnerability pattern, require human review and explicit approval.

*2) Runtime testing:* is performed if static analysis succeeds. It is organized into three levels: unit testing, integration testing, and end-to-end (E2E) testing. Unit testing is the first to be executed after static analysis, and its scope is a single feature of a single microservice. Integration tests have a larger scope, may involve more than a single service, and focus on verifying that *interaction* features work as expected. E2E tests validate the application in its entirety, by simulating the interaction of a user with the product, implementing the use case directly. To do so, they must execute *after* deployment, as they require the entire system to be up and running. For `sisred`, the team picked Cypress[6] as E2E enabler.

*3) Monitoring:* is the inspection of the system at runtime, through collection and analysis of telemetry data, to intercept and tackle issues affecting the infrastructure before they can propagate to business-critical processes [9]. To this end, the team of `sisred` exploited an existing monitoring system (Azure Monitoring), hooking it with the running instance of `sisred` during the reification of the infrastructure operated in the IaC phase. The monitoring system features two main components: a dashboard, showing real-time updates on the system status; and an automatic alert system, notifying the developer when unexpected and potentially disruptive events occur (e.g., in the case a container terminates unexpectedly

[5]https://www.terraform.io/

[6]https://www.cypress.io/

and gets restarted). The team identified two severity levels for alerts, resulting respectively in a notification on the shared workspace (low severity) or in an email to every team member (high severity).

*4) DevSecOps:* is a recent trend that promotes security in every phase of DevOps [19]. In the context of `sisred`, security is indeed considered in multiple phases of the automated pipeline. As mentioned before, the static analysis includes vulnerability checks and hotspot detection. Another critical source of potential security issues is relative to containerization: since microservices in `sisred` are deployed as containers, security issues with container images may well affect the deployed application. To intercept these issues, the team adopted a tool named Trivy[7], a vulnerability scanner for containers, and integrated it into the continuous integration pipeline.

### *F. Organization and timeline*

The project lead, acting as DevOps engineering head, had a crucial role in the group: their responsibility included researching and evaluating novel tools and practices and training teammates. The project lead thus progressively learned and experimented with new tools and techniques, discussing the novelties with the remainder of the team once they were considered suitable for integration in `sisred`. A crucial part of the discussion included detailed explanations of the *expected benefits* of adopting a novel practice or tool. A plenary training session conducted by the project lead followed the discussion to make every member of the team sufficiently proficient with the novel tool at hand: since the beginning, the team shared the goal to make everyone able to understand and improve the novel process. In case of need (for instance, if a team member was required a more profound knowledge of some technology), further one-to-one sessions were organized involving one team member and the lead. The team operated with considerable autonomy.

Overall, the project took several years, from the initial conception in late 2016 to the first production release in July 2020. The timeline depicted in Figure 5 shows that an extended period (about two years) spaced out the decision to begin the endeavor and the first commit. The first year included the propedeutic organization (e.g., assembling the team) and an activity of explanation of the expected benefits of the transition. Initially, non-technical figures, especially domain experts (editors with years of experience), were reluctant to change as the benefits to the end-user were unclear and the existing workflow was substantially unchanged since fifteen years. Their opinion began to change once they were showed some sketch-ups of the renewed user interface, but what really sparked interest and enthusiasm (resulting in improved feedback and commitment) were initial prototypes of the web-based user interface with AI-assisted tagging. The second year was then primarily spent on understanding the principles of DevOps, DDD, and microservice architectures. No team member was full-time on the project during the initial phases. The evolution of the project picked up rather quickly once the core of the development methodology was ironed out. The 2019 Coronavirus pandemic imposed a further acceleration to the project: editors began working at home (entirely or partially); thus, they asked for a version of `sisred` portable on mobile devices and personal PCs and that did not require a time-consuming procedure for installation or update.

[7]https://github.com/aquasecurity/trivy

| Metric | Unit | Prev. | Cur. | Change |
|---|---|---|---|---|
| Release Frequency | releases/day | 0.071 | 2.7 | +3700% |
| Commit-to-release time | hours | 8/24 | 0.19 | -97.6%/-99.2% |
| Commits per day | commit/day | 2 | 7.1 | +255% |
| Mean Time To Recovery (MTTR) | hours | 36 | 0.5 | -98.6% |
| Support tickets frequency | tickets/months | 40 | 19 | -52.5% |
| Issue resolution time | days | 4 | 3 | -25% |
| Nightly downtime | minutes/night | 30 | 0 | -100% |
| Dev. env. setup | minutes | 120 | 9 | -92.5% |
| Prod. env. setup | working hours | 16 | 0.35 | -97.8% |

TABLE II
SUMMARY OF THE METRICS CONSIDERED FOR EVALUATING THE IMPACT OF ADOPTING A MICROSERVICE ARCHITECTURE AND RELATED DEVOPS PRACTICES, TECHNIQUES, AND TOOLS.

## IV. RESULTS AND DISCUSSION

In this section, we present an evaluation of the process of architectural refactoring and adoption of DevOps for the development of `sisred`.

### *A. Measurable improvements*

Before beginning the refactoring, we collected several metrics that we believed could be useful to assess the efficiency of the software development process. Metrics for the original system and development process were obtained by interviewing the development team. Where applicable, these values were computed during the interview: release frequency (obtained by counting how many update emails were sent out in a time frame); commits per day (obtained by the SVN logs); new support tickets and issue resolution time (obtained from the OSTicket statistics); development and production environment setup time were directly measured by respectively asking a developer to set up a pristine laptop (which took about two hours), and ordering to the operations team a new server VM (delivered in one and a half working days) and letting a developer set it up (which took about four working hours). Other metrics are instead estimates made by developers as many activities were not formally tracked and it was not possible to obtain objective measures: Commit-to-release time, MTTR, and nightly downtime suffer from this bias. Values for the renewed system were all measured, as the required information was provided by the pipeline and repository management analytics.

Table II summarizes the results. The new architecture and practices impacted dramatically on almost all metrics. The impact is particularly prominent for those activities that required interaction with external teams, such as creating a production environment. In contrast, activities whose bottlenecks are

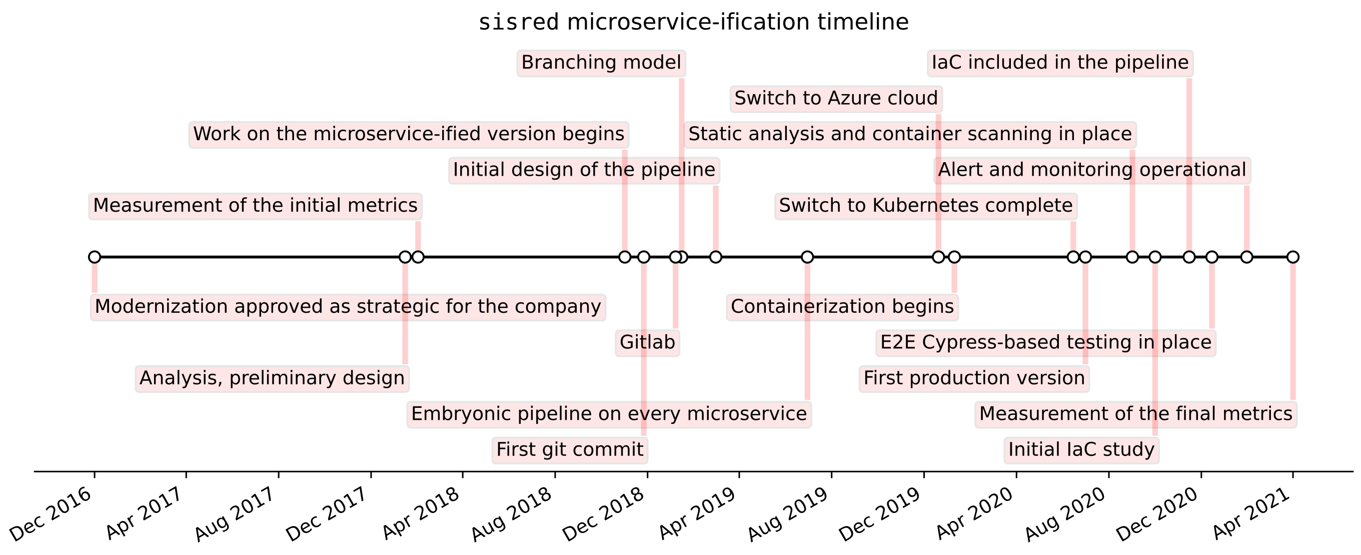

Fig. 5. Original architecture of `sisred`.

usually bound to local development and debug time (such as the mean issue resolution time) were less affected. None of the metrics regressed.

The *release frequency* mostly benefits from the complete automatization of the procedure. Version release was considered an expensive operation in the past, consuming overall one person-day, and thus performed on average only once every two weeks. Automatization was the driving factor behind the measured improvement.

The *commit-to-release time*, improves due to the complete removal of email-mediate interaction between developers: commits pushed on the appropriate branch cause a new release with no further intervention; also, only microservices that actually changed are re-deployed, shortening the time further.

*Commits per day* grow as a consequence of the changed role of the version control system: it was previously used as a backup system, and thus received commits towards the end of the day; currently, any self-contained change gets pushed.

The *MTTR* drops significantly as a combined effect of the cloud-based deployment, which removes most of the issues related to hardware failures and recovery of in-house servers, and introduces tools enabling fast recovery (e.g., almost instant database restoration); recovery time in case of disaster scenarios requiring the whole environment to be re-created benefit from the automated environment setup (see production environment setup time evolution).

The *support ticket frequency* is reduced as a consequence of two factors: first, the quality assurance performed in the pipeline intercepts most of the issues introduced with the usual development; second, the monitoring system promotes proactivity, as developers often intercept anomalies before the end-user is affected.

The *issue resolution time* remains similar: although there is some support from the novel infrastructure, most of the time is spent in the actual development of a resolution, which remains essentially unaffected by the novel practices.

The new system has no scheduled *nightly downtime*, the value drops to zero.

Both the *production* and *development environment setup time* are reduced significantly. The latter is set up by cloning the repository and issuing `docker-compose up`, with no external intervention, with the operation being currently network-limited (we measured the setup time for a new machine with an uninitialized cache, the time is consistently shorter in case the environment is simply restored). The former is more complicated, as it involves the creation of the actual infrastructure in the IaC phase, the release of all microservices, and the E2E testing—yet, since all operations are automated and there is no human intervention, the overall setup is orders of magnitude faster than the original one.

### B. *Opinion of the development team*

After a few months after the new version of `sisred` was operational, we interviewed the team to understand better how the change in the product impacted the workflow, the service maintenance time, and the overall satisfaction with the changes.

The interviewees identified the most notable difference to the previous state in the rarity of intervention requests from operators external to the team. The only condition where intervention from another team is required (that never happened up to the time of writing) is a disaster recovery procedure mandating access to the backups for restoration. Although the development team has more responsibilities than previously, the change is perceived as positive overall, as problem resolution is vastly simplified and streamlined.

When asked about the time spent for service maintenance, the development team appeared to be in difficulty in quantifying the improvement in terms of person-hours saved; they instead insisted that the time is now invested differently, and it is hard (or even meaningless) to compare the two workflows in these terms. They explained that, with the old infrastructure, most of the time spent on maintenance was not caused by software evolution but rather by operational issues: maintenance was intended as "keeping the system in nominal conditions." Most of the logic was concentrated client-side, which was rarely updated, and the dominant source of concern was the server-side: the system was deployed on a non-replicated Windows Server system which was a single point of failure. Interviewees reported that the following issues, which were a primary concern with the previous system, disappeared entirely with the new architecture:

- failures due to operating system updates, including both failures at applying updates and newly introduced incompatibilities;
- sporadic downtimes due to local blackouts or to scheduled maintenance of the local energy infrastructure;
- sporadic downtimes caused by absent Internet connectivity due to internal network configuration errors, scheduled maintenance and update of the local infrastructure, or external network failures;
- restoration from backup due to company-wide ransomware attacks via cryptolockers [5] (reportedly happened more than once);
- critical failures caused by human error, mostly due to stored procedures tested directly on the production environment, leading to data corruption or deletion, and thus requiring restoration from backup.

On the other hand, the new architecture introduced *new* maintenance activities that need to be performed:

- application or verification (when automatically applied) of software updates (mostly dependency updates);
- security audits;
- maintenance and update of the pipeline;
- experimentation with the pipeline (novel tool, novel practices, fine-tuning);
- infrastructure benchmarking and optimization (allocation of CPUs and memory, number of replicas, scaling).

The team quantified the time spent in these new activities in about one person-day per week, but they stressed that these maintenance operations are fundamentally different from the "old" maintenance operations. For example, they mentioned that dependency updates, testing, and bug fixes were considered development activities in the past, but most are now considered part of maintenance. We note that the change in the workflow induced a shift in the *perceived meaning of maintenance* in the team: the focus moved from maintaining a single instance of the product to maintaining (and evolving) the entire process of construction and verification of product instances.

### C. Lessons learned

Given the results summarized in Section IV-A, the first obvious lesson learned is that adopting DevOps and converting legacy applications to microservices can be hugely beneficial. Converting the architecture to microservices requires the domain to be partitionable into bounded contexts, and the benefits of a DevOps approach to development are more pronounced for those activities where communication and interaction are a bottleneck.

One relevant lesson is to make the transition a *coordinated effort* shared with the entirety of the team. The case presented here largely benefited from the small size of the development team and the motivation to renew practices perceived as time-consuming, repetitive, cumbersome, and obsolete. Larger teams would likely include members that would benefit less from the change: in these cases, we believe that it is key to *communicate* and shed awareness of potential benefits on all members. Along the same line, there is a delicate balance between the benefits that applying a state-of-the-art practice introduces and the *cost* of its introduction and application, which includes the learning curve and adaptation to new work modes. Sometimes, picking a simpler solution, even if technically not optimal, may produce similar benefits at a lower cost. In our experience, this was the case for the repository configuration, which was set up as a single repository rather than multiple repositories to be imported using submodules.

"Anticipation of change" is a well-established software engineering principle, and we believe it applies to the development process too. Extensive knowledge of the existing techniques and some experimentation with existing tools allows for building awareness of what is doable today, estimating the costs of implementing a practice, and predict how techniques and tools will evolve. This could be leveraged to achieve two results: first, reach the aforementioned balance between benefits and complexity/cost; second, be prepared to update and upgrade the practices in use to react effectively to changes in the development structure (e.g., the acquisition of new team members).

From the opinions gathered in Section IV-B, we learn that adopting DevOps practices radically changes the perception of the processes related to software production and maintenance. In particular, the latter is no longer intended to simply guarantee service continuity to end-users but evolves into providing a stable and reliable environment for developers: the product's final quality is primarily a consequence of the quality of the process. We also note that, although the time spent on maintenance intended in classic terms (having an instance of the system working as expected) is reduced considerably, new activities that could be classified as "maintenance of the process" get introduced. It is thus pretty difficult to estimate improvements in terms of "person-hours saved," as the time is spent *differently*, with bottlenecks and repetitive operations replaced by novel activities requiring human intervention, resulting in improved overall *quality* of the product and the development process.

## V. Conclusion

In this experience report, we documented the process that led a small team of a large company to refactor a monolithic software product and its development process, by decomposing it into microservices and switching to the DevOps philosophy. In doing so, we analyzed the previous state, performing interviews with the team, understanding the previous architecture and practices, and measured several process performance indices. We then discussed the architectural changes introduced to decompose the system, the DevOps workflow designed to improve the software evolution process, the branching policy and its relationship with runtime environments, the automated pipeline supporting the development process, and finally the automated quality assurance practices applied throughout such pipeline. We finally compared the renewed system and development process metrics with the previous one's, drawing useful lessons. We found that leveraging DevOps practices has enormous potential for improving the software production process; and that decomposition of existing monoliths into small-size, isolated, and coordinated services plays very well with the aforementioned practices.

## References

[1] Abildskov, J.: Additional git features. In: Practical Git, pp. 139–161. Apress (2020). https://doi.org/10.1007/978-1-4842-6270-2_8, https://doi.org/10.1007/978-1-4842-6270-2_8

[2] de Alwis, B., Sillito, J.: Why are software projects moving from centralized to decentralized version control systems? In: 2009 ICSE Workshop on Cooperative and Human Aspects on Software Engineering. IEEE (2009). https://doi.org/10.1109/chase.2009.5071408, https://doi.org/10.1109/chase.2009.5071408

[3] Artac, M., Borovssak, T., Nitto, E.D., Guerriero, M., Tamburri, D.A.: DevOps: Introducing infrastructure-as-code. In: 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C). IEEE (May 2017). https://doi.org/10.1109/icse-c.2017.162, https://doi.org/10.1109/icse-c.2017.162

[4] Balalaie, A., Heydarnoori, A., Jamshidi, P.: Microservices architecture enables devops: Migration to a cloud-native architecture. IEEE Software **33**(3), 42–52 (2016). https://doi.org/10.1109/MS.2016.64

[5] Brewer, R.: Ransomware attacks: detection, prevention and cure. Network Security **2016**(9), 5–9 (Sep 2016). https://doi.org/10.1016/s1353-4858(16)30086-1, https://doi.org/10.1016/s1353-4858(16)30086-1

[6] Clemencic, M., Couturier, B., Closier, J., Cattaneo, M.: LHCb migration from subversion to git. Journal of Physics: Conference Series **898**, 072024 (oct 2017). https://doi.org/10.1088/1742-6596/898/7/072024, https://doi.org/10.1088/1742-6596/898/7/072024

[7] Colavita, F.: DevOps movement of enterprise agile breakdown silos, create collaboration, increase quality, and application speed. In: Proceedings of 4th International Conference in Software Engineering for Defence Applications, pp. 203–213. Springer International Publishing (2016). https://doi.org/10.1007/978-3-319-27896-4_17, https://doi.org/10.1007/978-3-319-27896-4_17

[8] Di Francesco, P., Lago, P., Malavolta, I.: Migrating towards microservice architectures: An industrial survey. In: 2018 IEEE International Conference on Software Architecture (ICSA). pp. 29–2909 (2018). https://doi.org/10.1109/ICSA.2018.00012

[9] Ebert, C., Gallardo, G., Hernantes, J., Serrano, N.: Devops. IEEE Software **33**(3), 94–100 (2016). https://doi.org/10.1109/MS.2016.68

[10] Evans: Domain-Driven Design: Tacking Complexity In the Heart of Software. Addison-Wesley Longman Publishing Co., Inc., USA (2003)

[11] Hemon, A., Lyonnet, B., Rowe, F., Fitzgerald, B.: From agile to DevOps: Smart skills and collaborations. Information Systems Frontiers **22**(4), 927–945 (Mar 2019). https://doi.org/10.1007/s10796-019-09905-1, https://doi.org/10.1007/s10796-019-09905-1

[12] Hemon-Hildgen, A., Rowe, F., Monnier-Senicourt, L.: Orchestrating automation and sharing in DevOps teams: a revelatory case of job satisfaction factors, risk and work conditions. European Journal of Information Systems **29**(5), 474–499 (Jul 2020). https://doi.org/10.1080/0960085x.2020.1782276, https://doi.org/10.1080/0960085x.2020.1782276

[13] Hüttermann, M.: Infrastructure as code. In: DevOps for Developers, pp. 135–156. Apress (2012). https://doi.org/10.1007/978-1-4302-4570-4_9, https://doi.org/10.1007/978-1-4302-4570-4_9

[14] Knoche, H., Hasselbring, W.: Drivers and barriers for microservice adoption – a survey among professionals in germany. Enterprise Modelling and Information Systems Architectures (EMISAJ) p. Vol 14 (2019) (2019). https://doi.org/10.18417/EMISA.14.1, https://emisa-journal.org/emisa/article/view/164

[15] Leite, L., Rocha, C., Kon, F., Milojicic, D., Meirelles, P.: A survey of devops concepts and challenges. ACM Comput. Surv. **52**(6) (Nov 2019). https://doi.org/10.1145/3359981, https://doi.org/10.1145/3359981

[16] Lwakatare, L.E., Kuvaja, P., Oivo, M.: Dimensions of DevOps. In: Lecture Notes in Business Information Processing, pp. 212–217. Springer International Publishing (2015). https://doi.org/10.1007/978-3-319-18612-2_19, https://doi.org/10.1007/978-3-319-18612-2_19

[17] Lwakatare, L.E., Kuvaja, P., Oivo, M.: Relationship of DevOps to agile, lean and continuous deployment. In: Product-Focused Software Process Improvement, pp. 399–415. Springer International Publishing (2016). https://doi.org/10.1007/978-3-319-49094-6_27, https://doi.org/10.1007/978-3-319-49094-6_27

[18] Merson, P., Yoder, J.: Modeling microservices with DDD. In: 2020 IEEE International Conference on Software Architecture Companion (ICSA-C). IEEE (Mar 2020). https://doi.org/10.1109/icsa-c50368.2020.00010, https://doi.org/10.1109/icsa-c50368.2020.00010

[19] Myrbakken, H., Colomo-Palacios, R.: DevSecOps: A multivocal literature review. In: Communications in Computer and Information Science, pp. 17–29. Springer International Publishing (2017). https://doi.org/10.1007/978-3-319-67383-7_2, https://doi.org/10.1007/978-3-319-67383-7_2

[20] Perera, P., Silva, R., Perera, I.: Improve software quality through practicing DevOps. In: 2017 Seventeenth International Conference on Advances in ICT for Emerging Regions (ICTer). IEEE (Sep 2017). https://doi.org/10.1109/icter.2017.8257807, https://doi.org/10.1109/icter.2017.8257807

[21] Petrasch, R.: Model-based engineering for microservice architectures using enterprise integration patterns for inter-service communication. In: 2017 14th International Joint Conference on Computer Science and Software Engineering (JCSSE). IEEE (Jul 2017). https://doi.org/10.1109/jcsse.2017.8025912, https://doi.org/10.1109/jcsse.2017.8025912

[22] Pilato, C., Collins-Sussman, B., Fitzpatrick, B.: Version Control with Subversion. O'Reilly Media, Inc., 2 edn. (2008)

[23] Rademacher, F., Sorgalla, J., Sachweh, S.: Challenges of domain-driven microservice design: A model-driven perspective. IEEE Software **35**(3), 36–43 (May 2018). https://doi.org/10.1109/ms.2018.2141028, https://doi.org/10.1109/ms.2018.2141028

[24] Roche, J.: Adopting DevOps practices in quality assurance. Communications of the ACM **56**(11), 38–43 (Nov 2013). https://doi.org/10.1145/2524713.2524721, https://doi.org/10.1145/2524713.2524721

[25] Schaefer, A., Reichenbach, M., Fey, D.: Continuous integration and automation for devops. In: Lecture Notes in Electrical Engineering, pp. 345–358. Springer Netherlands (Sep 2012). https://doi.org/10.1007/978-94-007-4786-9_28, https://doi.org/10.1007/978-94-007-4786-9_28

[26] Spinellis, D.: Git. IEEE Software **29**(3), 100–101 (May 2012). https://doi.org/10.1109/ms.2012.61, https://doi.org/10.1109/ms.2012.61

[27] Viggiato, M., Terra, R., Rocha, H., Valente, M.T., Figueiredo, E.: Microservices in practice: A survey study. CoRR **abs/1808.04836** (2018), http://arxiv.org/abs/1808.04836

[28] Zhu, L., Bass, L., Champlin-Scharff, G.: Devops and its practices. IEEE Software **33**(3), 32–34 (2016). https://doi.org/10.1109/MS.2016.81

[29] Zou, W., Xuan, J., Xie, X., Chen, Z., Xu, B.: How does code style inconsistency affect pull request integration? an exploratory study on 117 GitHub projects. Empirical Software Engineering **24**(6), 3871–3903 (Jun 2019). https://doi.org/10.1007/s10664-019-09720-x, https://doi.org/10.1007/s10664-019-09720-x