Cromlech: Semi-Automated Monolith Decomposition into Microservices

Davide Cocco, Simone Staffa, Giovanni Quattrocchi, Alessandro Margara, Gianpaolo Cugola

Abstract—Microservices architectures conceive an application as a composition of loosely-coupled sub-systems that are developed, deployed, maintained, updated, and scaled independently. Compared to monoliths, microservices speed up evolution and increase flexibility. For these reasons they are becoming the reference architecture for many practitioners. A key challenge to embrace a microservices architecture is how to decompose an application into microservices: a choice that deeply affects all subsequent development phases in ways that are difficult to foresee and evaluate. Without any tool to support their reasoning, developers may erroneously evaluate the various alternatives, leading to inaccurate decomposition choices that would result in increased development, operations, and maintenance costs. This paper tackles the problem with Cromlech, a semi-automatic tool to decompose a software system into microservices. Cromlech (i) takes in input a high-level model of the system in terms of functionalities and data entities accessed by those functionalities, (ii) formulates decomposition as an optimization problem, and (iii) outputs a proposed placement of functionalities and data onto microservices, using a visual representation that helps reasoning on the resulting architecture. Cromlech evaluates design concerns, communication overheads, data management requirements, opportunities and costs of data replication. Our evaluation on a real-world industrial application shows that Cromlech consistently delivers more efficient solutions than simple heuristics and state-of-the-art approaches, and provides useful insights to developers.

Index Terms—microservice architecture, service decomposition, service modeling, software architectures

1 Introduction

IT practitioners are increasingly migrating from so-called monoliths to microservices architectures, which decompose a software system into independently deployed services to ease evolution and maintenance [1]. Monoliths are developed using a single programming language and packaged as one complex deployment unit. When the application grows in scale, monoliths present significant limitations [2]: the intricate dependencies among components become difficult to maintain and individual functionalities cannot be managed independently from one another. If a single component fails, the whole system may become unavailable, and the only way to scale the application is to replicate the whole deployment unit. Instead, the microservices architecture [3], [4] conceives applications as a composition of loosely-coupled units: each of them is an independent process that communicates with the others through lightweight network protocols such as HTTP or MQTT. Each unit (i.e., a microservice) includes a set of logically-related application components and is developed, operated, and deployed independently from the others. This means that each microservice can be implemented using a different technology stack, may use an independent data store, and can be managed without affecting the other units. Moreover, in case of failures, the system would not be completely unresponsive and only a subset of its functionalities would be unavailable [5].

The main challenge in this migration is how to decompose an existing monolith into microservices [6], [7], [8], since this choice may significantly impact both *organizational* and *operational* concerns, which often pursue conflicting goals [9], [10]. Organizational concerns favor highly decentralized decompositions, where individual services represent a single business aspect and include only functionalities that are strongly related to each other, i.e., they are highly cohesive. Cohesion increases the agility in managing the system, but may introduce an overhead for operating it, typically in terms of an increased communication between services. Conversely, operational concerns push towards more centralized solutions that reduce communication [11] and data management costs. Indeed, communication costs are higher in cross-microservice calls, which are necessary to access remote functionalities or to retrieve remote data [12]. A common approach to mitigate the latency for remote data access in microservices architectures is replication. Microservices can access a local replica of the data they are interested in (improving read access latency), but this comes at the cost of propagating updates to all replicas. This trade-off further complicates the design of a decomposition, making this process impossible to manage without proper support tools.

In summary, when migrating to microservices, software engineers need to carefully evaluate the complex trade-off between organizational and operational aspects, a task that may be extremely difficult and error-prone without the help of any support tool [13]. Some approaches have been presented in the literature to help software engineers in the decomposition process [14], [15]. Some solutions are only theoretical [6], [16] and provide a set of best practices and guidelines, others [17], [18] describe tools that analyse the application source code and automatically decompose it into a set of microservices. While these solutions are an initial step in the right direction, there are still open challenges. Theoretical frameworks leave practitioners in charge of manually decomposing the monolith with some complex application-specific design decisions to be evaluated; automated tools are often tight to a single programming language or technology stack [19] and do not guarantee that the resulting decomposition is aligned with developers and business needs. Most importantly, none of these solutions explicitly

• *Dipartimento di Elettronica, Informazione e Bioingegneria*
Politecnico di Milano
mail: name.surname@polimi.it

addresses the tension between organizational and operational requirements, thus neglecting part of a multi-faceted problem.

Our research proposes Cromlech, a semi-automatic tool to decompose a software system into microservices that carefully considers the subtle trade-off between organizational and operational aspects. Cromlech takes in input (i) a high-level description of a software system in terms of *operations* and *data entities*, (ii) the maximum allowed number of microservices to create, (iii) a parameter that indicates the relative importance of organizational aspects over operational ones, allowing engineers to set their preferences based on the specific environment in which the system is being developed and operates. Cromlech parses the application model and instructs a Mixed Integer Linear Programming (MILP) solver to optimize the placement of operations and data entities according to the users' needs. The computed decomposition is then displayed on a visualizer so that users can investigate the solution, evaluate its structure and costs, and, if needed, reiterate the process with a different input configuration. Cromlech relies on an abstract model of the application and consequently it is technology agnostic. It considers several factors in the decomposition process including cohesion, communication overhead, data management requirements, and the costs and benefits of data replication. Despite the optimization problem may be difficult to solve, Cromlech generates good solutions within some minutes to few hours, depending on the complexity of the system under analysis. This allows software architects to rapidly obtain viable solutions that they can iteratively refine at will.

Cromlech builds on our experience with Pangaea [20]. Like Cromlech, Pangaea considers both organizational and operational concerns and allows developers to set their relative weights. In Pangaea, organizational aspects are centered around data entities: this captures well the semantic relations among data, but does not ensure that operations related to the same business domain are deployed onto the same microservice. Moving from the observations we collected from developers when using Pangaea on a real-world software system, we designed Cromlech to focus primarily on operations, thus targeting decompositions where microservices expose an interface (set of operations) that matches business domains more coherently. The placement of data derives from the location of operations, avoids distributed data management by co-locating data entities that require coordinated access, and exploits replication to improve read access performance when suitable.

Our evaluation suggests that Cromlech produces architectures where microservices clearly reflect individual business functionalities, and selects the granularity of the decomposition and the placement of data to reduce operational costs. Moreover, our experience with Cromlech highlights the complexity of manual optimization and the benefits of a decision support tool. Compared to human-designed architectures, some solutions proposed by Cromlech were difficult to see. At the same time, it was easy to manually adapt architectures produced by Cromlech, to better cover the software engineers' requirements.

In essence, this paper continues our research on designing support tools to decompose a software system into microservices considering both organizational and operational requirements. Compared to our former solution, Pangaea, the paper propose: (i) a new system model focused on operations, (ii) a novel MILP formulation that exploits the new model, and (iii) a comprehensive evaluation that features two real-world applications and an in-depth comparison with simple heuristics,

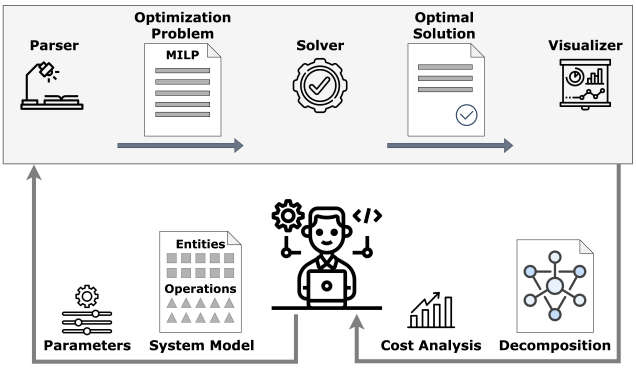

Fig. 1: Cromlech: overview of the workflow.

manual solutions made by practitioners, Pangaea, and another state-of-the-art decomposition approach called Service Cutter (SC) [21]. Results show that Cromlech consistently outperformed other approaches by generating more efficient solutions in-line with user needs. Moreover, Cromlech allowed developers to easily evaluate their manual decompositions: for instance, it highlighted that solutions developed by practitioners were only optimized for organizational aspects, but were not efficient in reducing operational costs.

The rest of the paper is organized as follows. Section 2 and Section 3 present Cromlech and the MILP optimization problem it is based on. Section 4 presents the empirical evaluation of Cromlech and the comparison with other relevant approaches. Section 5 reviews related work. Section 6 concludes the paper.

2 Solution Overview

Fig. 1 presents the Cromlech[1] workflow, organized in three main steps: *system modeling*, *optimization*, and *visualization*.

System modeling. Cromlech allows users to define a model of the system to decompose as a YAML (a data format frequently used for configuration tasks) text file. It describes the data entities and operations that compose the application, along with their mutual relations. This user-written system model is the input of a *parser* that pre-processes data and generates a MILP problem.

System modeling. Cromlech allows users to define a model of the system to decompose, which describes the data entities and operations that compose the application, along with their mutual relations. The model is encoded as a YAML file, a data format frequently used for configuration tasks. For compatibility, the input file may be automatically translated from other formats: for instance, we provide translation tools from the TOSCA standard[2]. This user-written system model is the input of a *parser* that pre-processes data and generates a MILP problem.

Optimization. A *solver* takes this MILP problem and produces a solution according to a set of *parameters* (provided as input along with the system model) that express user preferences.

Visualization. A *visualizer* creates a visual representation of the solver's solution and presents to the user the decomposition along with a detailed analysis of its costs. Developers can then assess the proposed solution and decide whether to accept it or to refine the system model and input parameters.

In the rest of the section, each phase is described in detail.

1. Source code available at http://github.com/deib-polimi/Cromlech
2. https://www.oasis-open.org/standard/tosca/

2.1 System modeling

The design of the Cromlech modeling framework balances two requirements: (i) expressivity, to capture organization and operational concerns; (ii) simplicity, to limit the effort for developers to build the model. Thus, in Cromlech *data entities* and *operations* are characterized by small, yet meaningful, sets of attributes.

Data entities. Data entities are basic elements of data that Cromlech treats as atomic units. The concept of data entity is independent of the specific data model and level of granularity, allowing developers to adapt the modeling framework to their needs. For instance, in a relational data model, a data entity can model a single table: Cromlech will treat the table as an unbreakable unit and map it to microservices accordingly. Alternatively, developers may decide to model multiple related tables as a single data entity or to split a table into multiple data entities. In the first case, Cromlech will not distinguish individual tables and will consider them as a whole. In the second case, Cromlech will have the opportunity to assign the various parts of the table to different microservices. A data entity e provides the following properties.

name: a label that uniquely identifies e in the model.

description: an optional string that developers use to annotate relevant information associated to e (for instance, the database tables e refers to).

Operations. Operations represent units of execution of the application, which are candidates to become logic functionalities assigned to microservices. Each operation accesses (reads and writes) data entities and is associated with a single microservice. An operation o is characterized by the following properties.

name: a label that uniquely identifies o in the model.

data access: the list of data entities that o accesses. For each data entity, developers can specify if the access is *read-only* or *read-write*. Cromlech interprets accesses as dependency relations between operations and data entities, and it attempts to co-locate on the same microservice an operation and the data entities it accesses. Placing a data entity e and an operation o that accesses e on different microservices incurs a cost in terms of communication.

frequency: a number that says how frequently o is invoked. In the decomposition process, Cromlech focuses on reducing the costs associated with operations that are invoked more frequently.

transactional: indicates whether the operation needs to be executed with transactional semantics. As common in microservices architectures, our model assumes that transactional semantics is only possible within individual microservices and not across microservices. Accordingly, a transactional operation will be always located on the same microservice where all the data entities it accesses are deployed.

co-located operations: list of other operations that need to be located on the same microservice as o. Using this attribute, developers may indicate operations that belong to the same business unit and must be deployed together on the same microservice. Accordingly, co-located operations help developers to express organizational constraints that Cromlech cannot break. At the same time, the presence of co-located operations reduces the number of acceptable solutions and may simplify the task for the solver.

The above attributes delineate an informative model of an application while limiting modeling complexity. We note that some attributes could be automatically or semi-automatically derived. For instance, static analysis tools could extract attributes and operations, while monitoring approaches could precisely estimate the frequency of operations in the system to be decomposed. We will explore the integration of these tools to further increase the simplicity of Cromlech in future work.

2.2 Optimization

The system model provided by the user is parsed and pre-processed in order to properly generate the optimization problem. In particular, the pre-processing step: (i) removes the entities that are accessed by a single operation, as they can be easily added to the microservice that hosts the operation at the end, and (ii) eliminates operations that do not access any remaining entity, as they can be added to any microservice without affecting the final decomposition in terms of organizational or operational concerns. After the pre-processing step, Cromlech generates the MILP problem that aims to find an optimal placement of operations and entities onto a set of microservices, balancing organizational concerns and operational costs.

Organizational concerns are measured by the *cohesion* metric (the higher the better), that is the affinity and coherence of operations and data entities within individual microservices and the decoupling among different ones.

Operational concerns are measured by *operational costs* (the lower the better), that sum *communication* and *data management* costs. Communication costs measure the need of inter-service communication. Depending on the specific technology being adopted, microservices communicate using synchronous communication (for instance, HTTP/REST API) or asynchronous propagation of messages (for instance, using a message broker such as RabbitMQ[3] or a queuing system such as Apache Kafka[4]). Cromlech allows data entity to be replicated in different microservices with the goal of improving access speed (fewer remote calls implies lower communication cost). Data management costs reflect the additional effort for keeping each replica up to date. Cromlech assumes that any operation can read from a co-located replica, but all the updates (writing mode) are executed on a so-called *leader-replica* and pushed asynchronously to the others (eventual consistency). Thus, operations that require transactional semantics must be co-located with the leader replica, a condition that is formalized as a constraint in our formulation (see Section 3). This assumption is well-known in the literature as the *single writer principle* [22].

Cromlech solver takes as input a small number of parameters that guide the decomposition process based on the requirements of developers. They include:

number of microservices: indicates the maximum number of microservices that the decomposition can use. The solver may assign entities and operations only to a subset of microservices, resulting in a decomposition of fewer microservices.

organization-operations ratio: a real number α that indicates the importance developers attribute to organizational concerns (cohesion of microservices) over communication concerns (the costs of remote data access and replication), on a scale between 0 and 1. The default value is $\alpha = 0.5$, which suggests to Cromlech that organization and communication concerns are equally important. Increasing the value would favor solutions where microservices are internally highly cohesive (leading to a

3. https://www.rabbitmq.com/
4. https://kafka.apache.org/

potentially higher number of microservices and thus increasing the cost of communication) while decreasing the value would favor solutions that reduce inter-service communication (potentially at the cost of decreased cohesion).

The reduced set of parameters presented above provide developers flexibility when needed without relinquishing simplicity. Moreover, the organization-communication ratio α provides a single parameter to steer the decomposition towards design concerns or runtime concerns.

2.3 Visualization

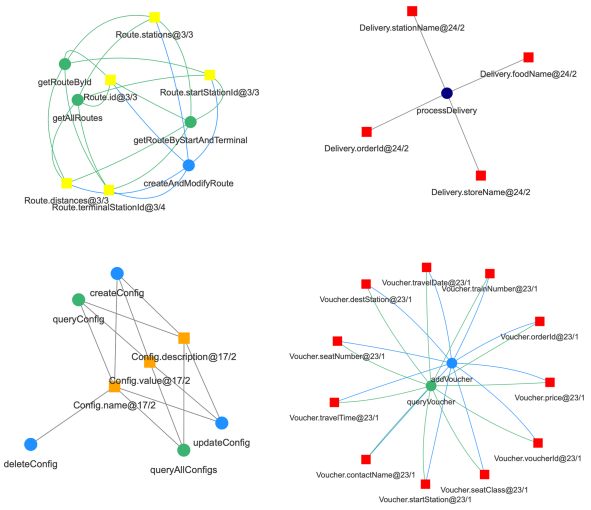

Fig. 2: Visualization of a decomposition into 4 microservices.

We conceive Cromlech as a decision support tool: it may not fully automate the decomposition process but it should help developers reason on the system and evaluate the consequences of a given decomposition choice in terms of design and operational costs. Accordingly, we built a visualizer component that offers a graphical representation of the proposed decomposition as a dynamic Web page. Fig. 2 shows an example of a decomposition where entities (squares) and operations (circles) are mapped to microservices. The depicted decomposition splits the monolith into 4 microservices which clearly define, respectively, 4 different bounded contexts: routes, deliveries, config, and vouchers. Entities colored in red, orange, and yellow are, respectively, leader replicas, non-leader replicas, and non-replicated data. Dark blue, light blue, and green colors indicate, respectively, write operations with transactional semantics, write operations without transactional semantics, and read-only operations. In addition, Cromlech outputs a detailed report with the cohesion, communication, and data management costs of the proposed solution. Developers may use the report to evaluate the trade-offs of the solution and to refine their system model or choice of input parameters.

3 Optimal Monolith Decomposition

We formulate monolith decomposition as a MILP problem by denoting E the set of data entities, O the set of operations defined in the system model, and M the set of microservices where data entities and operations must be placed. Two decision variables x and y encode the placement of operations and data entities onto microservices, respectively:

$$x_{o\in O,m\in M} = 1 \text{ if } o \text{ is placed on } m, 0 \text{ otherwise.}$$
$$y_{e\in E,m\in M} = 1 \text{ if } e \text{ is placed on } m, 0 \text{ otherwise.}$$

Our model allows data entities to be replicated at different microservices. As common in microservices architectures, we assume that a single service is responsible for all updates (write accesses) to a given data entity $e \in E$. This is the leader replica for e and we use a variable l to encode its placement.

$$l_{e\in E,m\in M} = 1 \text{ if the leader replica of } e \text{ is placed on } m, 0 \text{ otherwise.}$$

3.1 Organizational concerns

The cohesion metric represents the benefit of decomposing the software systems into independent and highly cohesive modules. Let us denote O_m the set of operations that are associated with microservice $m \in M$, that is:

$$\forall_{m\in M}\ \forall_{o\in O}\ o \in O_m \leftrightarrow x_{o,m} = 1$$

Then, the cohesion metric for microservice $m \in M$ is defined as:

$$Coh_m = \frac{\sum_{o1\in O_m, o2\in O_m} S(o1,o2)}{|O_m|^2}, \text{ if } |O_m| > 0$$
$$Coh_m = 0, \text{ otherwise}$$

Where $S(o1,o2)$ is the *similarity* between operations $o1$ and $o2$. Ideally, a high similarity should indicate that two operations belong to the same business domain. Accordingly, placing them onto the same microservice increases the cohesion metric. We build our definition of similarity on the assumption that two operations are similar if the set of data entities they access is similar. For each data entity $e \in E$ and for each operation $o \in O$, let us denote $acc_{e,o}$ as a boolean variable that is 1 if and only if o accesses e either in read or write mode. We extract the value of such variables from the *data access* property of operations in the input model. For each operation $o \in O$, let us define E_o as the set of entities that o accesses either in read or in write mode[5].

$$\forall_{o\in O}\ \forall_{e\in E}\ e \in E_o \leftrightarrow acc_{e,o} = 1$$
$$S(o1,o2) = \frac{|E_{o1} \cap E_{o2}|}{min(|E_{o1}|, |E_{o2}|)}$$

The total cohesion is the average of the cohesion metrics for each service $m \in M$, weighted by the number of operations in m. It is a value between 0 and 1, where higher values are better.

$$Coh = \sum_{m\in M} Coh_m \cdot \frac{|O_m|}{|O|}$$

3.2 Operational concerns

Operational costs include communication and data management costs. The former occur when an operation $o \in O$ needs to access a data entity $e \in E$ but the two are placed on different microservices, while the latter measure the costs associated with replication. Indeed, replication allows operations to access locally replicated data without incurring the cost of remote data access, but it requires keeping remote replicas up-to-date. As mentioned above, we assume that, for a given data entity $e \in E$, a single leader replica is responsible for all updates (write accesses) to e.

5. The formulation is generated after the pre-processing step, which already (i) removed all data entities that are accessed by a single operation and (ii) operations that do not access any (remaining) data entity.

Formally, we distinguish two communication costs: $R_{o,e}$ and $W_{o,e}$, one for reading and one for writing entities, while data management costs are associated to each entity as $MngCost_e$. To precisely define these elements of cost we need to introduce two additional boolean variables $accR_{e,o}$ and $accRW_{e,o}$ for each data entity $e \in E$ and for each operation $o \in O$. They hold 1 if and only if o accesses e in read-only or in read-write mode, respectively. We extract these values from the *data access* property of operations in the input model. We define $R_{o,e}$ as the cost that $o \in O$ incurs for reading a data item $e \in E$. The cost is 0 if e is placed in the same microservice $m \in M$ where the operation resides. Otherwise, it is the cost for accessing the leader replica, which is proportional to the frequency f_o of invocation of o.

$$R_{o,e} = \sum_{m \in M} accR_{e,o} \cdot f_o \cdot x_{o,m} \cdot (1 - y_{e,m})$$

$W_{o,e}$ is the cost that $o \in O$ incurs for writing a data item $e \in E$. The cost is 0 only for operations that are located on the leader replica for e, otherwise, it is proportional to the frequency of invocation of o. In other words, operations always access the leader replica when writing e, even if they have a local replica for e on the same microservice on which they are placed.

$$W_{o,e} = \sum_{m \in M} accRW_{e,o} \cdot f_o \cdot x_{o,m} \cdot (1 - l_{e,m})$$

Thus, communication costs are computed as follow:

$$CommCost = \sum_{o \in O, e \in E} (R_{o,e} + W_{o,e})$$

Finally, an entity $e \in E$ pays a data management cost for keeping replicas up to date. It is proportional to the number of (non-leader) replicas of e and to the frequency at which e is updated:

$$MngCost_e = \sum_{o \in O} accRW_{e,o} \cdot f_o \cdot ((\sum_{m \in M} y_{e,m}) - 1)$$

Thus, the total data management cost is computed as

$$MngCost = \sum_{e \in E} MngCost_e$$

Summing up all contributions, the operational costs for a given placement of entities and operations onto microservices are defined as:

$$OpCost = CommCost + MngCost$$

3.3 Objective function and constraints

The goal of the optimization problem is to maximize the cohesion metric Coh while avoiding operational costs $OpCost$. Since Coh is a real number between 0 and 1 we want also to normalize the operational costs $OpCost$ to a scale between 0 and 1. The minimum value of $OpCost$ is already 0 as it happens when a fully centralized architecture is adopted (with no remote data access and no replication). Conversely, the maximum value that $OpCost$ may reach occurs in a fully decentralized architecture, where each operation is placed on a different microservice (unless this is not allowed by some of the constraints in the problem) and data entities are replicated in every service to maximize the cost for propagating updates. Let us denote this maximum operational cost as $OpCost_{max}$. The objective of our optimization problem becomes:

$$Obj = \alpha \cdot Coh - (1 - \alpha) \cdot \frac{OpCost}{OpCost_{max}}$$

Valid solutions to the problem need to satisfy the following constraints.

Each operation is deployed on exactly one microservice.

$$\forall_{o \in O} \sum_{m \in M} x_{o,m} = 1$$

Each entity is deployed on at least one microservice.

$$\forall_{e \in E} \sum_{m \in M} y_{e,m} \geq 1$$

Each entity has exactly one leader replica.

$$\forall_{e \in E} \sum_{m \in M} l_{e,m} = 1$$

The leader replica is a replica.

$$\forall_{e \in E, m \in M} y_{e,m} \geq l_{e,m}$$

Deployment needs also to enforce the constraints expressed by the developers with respect to the co-location of multiple operations. In particular, for any pair of operations $o1 \in O$ and $o2 \in O$, let us define a boolean variable $coloc_{o1,o2}$ that is 1 if and only if developers requested the co-location of $o1$ and $o2$ (*co-located operations* attribute of operations in the input model). Using this variable we may define the co-location constraint as follow:

$$\forall_{o1 \in O, o2 \in O} \sum_{m \in M} x_{o1,m} \cdot x_{o2,m} \geq coloc_{o1,o2}$$

Finally, operations with transactional semantics need to be on the same microservice of the data they access. This enforces a common approach in microservices architectures, where transactional semantics is not enforced across microservices, but only within microservices. In other words, if an operation requires strong guarantees in terms of atomicity, isolation, or integrity when accessing data elements, it needs to be executed in the same microservice that hosts the leader replica for all those elements. For any operation $o \in O$, we define a binary variable tr_o that is equal to 1 if and only if developers requested o to be executed with transactional semantics (*transactional* attribute of operations in the input model). Using this variable we may define the transactional constraint as follow:

$$\forall_{o \in O, e \in E, m \in M} x_{o,m} \geq tr_o \cdot acc_{e,o} \cdot l_{e,m}$$

Notice that the above problem is not linear for two reasons. (i) Some formulas include a multiplication of binary variables. For instance computing co-located operations involve multiplying x by x, and computing the operational costs involve multiplying x, y, and l. (ii) Some formulas include a multiplication of a binary variable and an integer variable. For instance, computing the coherence metric involves multiplying x by the overall number of operations in a given microservice (an integer number). To address these issues, we use well known approaches [23] to linearize any product of two variables.

4 Evaluation

This section presents the empirical evaluation of Cromlech, which aims to answer three research questions:

RQ1 What is the quality of the decompositions generated by Cromlech and how do they compare with decompositions generated by other state-of-the-art approaches?

RQ2 How do the input parameters of Cromlech affect its solutions?

RQ3 How do practitioners benefit from the usage of Cromlech?

The main challenge in evaluating a decomposition approach is the lack of a ground truth. Given a software system there is not a single *optimal* decomposition into microservices, and developers may favor one decomposition over another depending on technological, operational, organizational constraints, or even personal preferences. For the same reasons, comparing two decompositions is not trivial, as small changes in the mapping of data entities and operations onto microservices may lead to significant differences in the quality of a decomposition as perceived by a team of developers.

To address these challenges, we consider as case studies two software systems, Tutored and TrainTicket, for which we have a manual decomposition into microservices that we can use as a reference. Tutored[6] is a real-world monolithic application, while TrainTicket has been proposed as a benchmark for microservices systems by the software engineering research community [24]. Moreover, we compare Cromlech results not only with the decompositions provided by the use-cases, but also with those generated by alternative approaches, and we compare them using multiple quantitative and qualitative metrics, including the objective function of Cromlech, multiple metrics of similarity with the reference architecture, as well as a detailed manual inspection.

We organize the rest of the section as follows: Section 4.1 presents our evaluation methodology. Section 4.2 and Section 4.3 analyze the results we obtained using Tutored and TrainTicket as case studies, respectively. Section 4.4 comments our findings and Section 4.5 presents possible threats to their validity.

4.1 Methodology

Case studies. The first case study we adopt for our evaluation is a real-world software system developed by Tutored, a tech startup that works in the education sector, and consists of a REST API developed with Node.js, Express, and Typescript. The application is currently live in production, and Tutored's Web and mobile applications are using it. The system was designed as a monolith to speed up the initial development. Recently, Tutored has been experiencing a significant traffic growth on its platforms, and the developers are considering to decompose the system into microservices to improve scalability and flexibility. Tutored has been previously used to evaluate the Pangaea decomposition approach [20] and we have a reference manual decomposition provided by the developers.

The second case study, TrainTicket is a Web application consisting of 41 microservices and used as a benchmark for software engineering research studies [24]. It represents a mock application for buying and managing train tickets, including user registration and session management.

Alternative approaches. We compare Cromlech with two alternative state-of-the-art approaches. Like Cromlech, they both take in input a system model and produce a decomposition of the system into microservices. However, they differ both in terms of the input model they require and in terms of the objective function and optimization mechanism they adopt.

ServiceCutter [21] considers both data elements and operations as *entities* to be placed onto microservices. It defines a large catalog of coupling criteria that link entities to each other, such as semantic similarity, and consistency or security constraints. The resulting modeling approach gives much freedom to developers, who can represent the relations between entities in a very precise way. For these reasons, however, ServiceCutter models may be more complex to build and validate than those of Cromlech. ServiceCutter internally represents the structure of the system as a graph, using coupling criteria to determine the weights of the relations between entities. It exploits clustering algorithms to compute the decomposition. As entities are associated to a single microservice, ServiceCutter decompositions do not exploit replication. Developers can choose three different clustering algorithms to decompose the monolith: Girvan-Newman [25], Leung [26] and Chinese Whispers [27].

Pangaea [20] formulates decomposition as a linear programming problem, as Cromlech does. In fact, some of the authors were involved in the development of Pangaea and many design choices in Cromlech derive from the lessons learned while working on Pangaea. Like Cromlech, Pangaea considers both organizational and operational concerns and allows developers to set their relative weights. In Pangaea, organizational aspects are centered around the relations between data elements. Conversely, Cromlech focuses on aggregating related operations, aiming to better identify individual business domains. Pangaea provides a simple model of operational aspects: operations pay a cost to access remote data and replication has an overhead (selected as a parameter by the developer) that increases linearly with the number of replicas. It does not include a concept of leader replica and does not model the propagation of updates to followers, as Cromlech does.

Similarity metrics. As both the case studies we adopt offer a reference decomposition, we are interested in capturing the similarity of the decompositions generated by the tools under analysis with such references. To do so, we defined two quantitative metrics to measure the similarity between two decompositions D_1 and D_2 of the same software systems.

The first one, *operations similarity*, looks at all possible pairs of operations. Given a pair of operations (o_i, o_j), we say that two decompositions D_1 and D_2 are similar with respect to (o_i,o_j) if o_i and o_j are co-located (they are in the same microservice) both in D_1 and D_2 or if they are not co-located (they are not in the same microservice) neither in D_1 nor in D_2. Consequently, we define $Sim^{D_1,D_2}(o_i,o_j)$ as a value that is 1 if D_1 and D_2 are similar with respect to (o_i, o_j) and -1 otherwise. The operations similarity between decompositions D_1 and D_2, Sim^{D_1,D_2} is defined as the normalized sum of the similarities of each pair of operations:

$$Sim^{D_1,D_2} = \frac{\sum_{o_i \in O, o_j \in O, o_i \neq o_j} Sim^{D_1,D_2}_{o_i,o_j}}{|O| \cdot (|O|-1)/2}$$

Second similarity metric, *Data similarity*, is defined likewise, but considers pairs of data entities instead of pairs of operations, and evaluates whether decompositions co-locate the primary replicas of the entities or not.

Experimental environment. We performed all experiments on a machine equipped with a 6-core/12-threads Intel Core i7-8700 CPU and 64 GB of RAM running Linux 5.10 (Debian). Cromlech was executed with the default input parameters presented in Section 2, unless otherwise specified. As a solver, we used Gurobi 9.5.1[7]. The combinatorial nature of the problem could make the search of an optimal solution too long. In practice, we observed that the value of the best solution found by Cromlech tends to

6. https://www.tutored.me

7. https://www.gurobi.com

stabilize after a period of time that depends from the specific case study and configuration parameters. Hence, for each case study, we set a maximum timeout after which we stopped the search in all our experiments. We report the values of the objective functions over time for each case study in the respective section.

4.2 Tutored case study

Ops (before/after pre-proc)	71/69
Data entities (before/after pre-proc)	271/166
Transactional ops (before/after pre-proc)	7/4
Cohesion of the monolith	0.1579

TABLE 1: Main characteristics of the Tutored software architecture

Table 1 presents the most relevant characteristics of the software architecture of Tutored. The system model was provided by Tutored engineers and comprises 71 operations and 271 data entities. The developers identified 7 transactional operations and did not specify any further requirement for co-locating operations. The pre-processing step reduced the number of operations to 69 (4 of which are transactional) and the number of data entities to 166. These represent the input for the decomposition problem that we consider for Cromlech, ServiceCutter, and Pangaea. The cohesion of a monolithic architecture (that is, the worst possible value of cohesion) is 0.1579. Later in this section, we will use this number as the baseline to evaluate the cohesion of the various decompositions.

Manual decomposition. The reference decomposition for the Tutored use case is a manual decomposition defined by Tutored engineers. Designing the decomposition took about one working day (6 to 8 hours). Given the complexity of reasoning on a large number of data elements, the engineers considered data at the granularity of database tables rather then individual columns, as we did in Cromlech. We present the main statistics of the manual decomposition in Table 2 and we present the size and cohesion of individual microservices in Table 3. The engineers focused mostly on organizational concerns, and identified 4 main business domains that they mapped onto 4 microservices: (1) *Content and activities* provides operations to access content, such as video streams, webinars, and posts. It also exposes operations related to activities and events. (2) *Users* provides operations to manage accounts. It includes transactional operations related to social network accounts. (3) *Jobs* provides operations related to job offers and interviews. (4) *Curricula* provides operations to compile a curriculum vitae.

Number of services	4
Cohesion metric [0..1]	0.3715
Operational cost [0..1]	0.4094
Total value (with $\alpha = 0.5$) [-1..1]	-0.0379

TABLE 2: Manual decomposition for Tutored: main characteristics.

Table 2 includes a quantitative analysis of the manual decomposition using the definition of cohesion metric and operational cost of Cromlech. The analysis confirms the focus of the software engineers on organizational aspects (at the price of higher operational cost at run time). The cohesion metric has a value of 0.37. Recall that cohesion is on a scale from 0 to 1, where the worst possible value of the monolithic solution for Tutored is 0.1579. As

Service	Num of ops	Coh metric
Content and activities	19	0.2459
Users	14	0.5643
Jobs	25	0.2429
Curricula	11	0.6354
Total	69	0.3715

TABLE 3: Manual decomposition for Tutored: number of operations and cohesion per service.

we will better see when discussing and comparing other decompositions, a cohesion of 0.37 is relatively high for a decomposition of 69 operations onto only 4 microservices. However, this value can be improved by considering more microservices: for instance, the microservice named *content and activities* clearly includes subsets of operations and data entities that are disconnected from each other. As we will see, Cromlech identifies these subsets and suggests to split them to increase cohesion and reduce operational costs. The operational cost of the manual decomposition is 0.41 (again, on a scale from 0 to 1, lower values are better), which is very high for an architecture with only 4 microservices, as we will see in comparison with alternative solutions. Most data entities are stored in a single service, without replication. A single exception is the *Education* table that the engineers decided to replicate on all services. Being a central component in the data model of the application, this table alone accounts for 63% of the operational costs to access remote data and 100% of the replication costs (that is, the cost for propagating updates to all replicas).

Cromlech. To evaluate Cromlech on the Tutored architecture, we performed the following experiments: (i) We forced Cromlech to use the same number of microservices (only 4) proposed by the software engineers for their manual decomposition. (ii) After observing the decomposition generated by Cromlech and the values of cohesion metric and operational costs it provides, we tried to manually improve it. This experiment aims to see if the feedback Cromlech provides can be useful for developers to gain more insight on their system and refine the decomposition in an iterative process. (iii) We let Cromlech compute the best decomposition by setting a high number of microservices (15). In all experiments, we run Cromlech with different values of α to study the sensitivity of our algorithm to this parameter and its ability to capture and control the preferences of the developers.

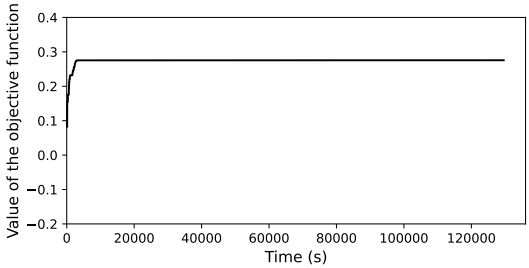

Fig. 3: Value of the best decomposition found over time (15 microservices, α=0.5)

Before analyzing the quality of the decomposition Cromlech produces, let us discuss the time it requires to compute them. Fig. 3 shows how the value of the decomposition computed by Cromlech increases (i.e., improves) over time, considering 15 microservices and $\alpha = 0.5$. Albeit the timeout of 36 hours was not enough to find a global optima for the optimization problem,

the figure shows how the value of the decomposition already becomes stable after 1 hour. With a smaller number of microservices this time further decreases: for instance, with 4 microservices, the value becomes stable after 10 minutes. As we demonstrate in the rest of the section, the decomposition calculated after this time have a similar or better quality than alternative solutions. Overall, this indicates that Cromlech can find good solutions within tens of minutes for a real-world application.

α	Cohesion	Op cost	Total	Op sim	Data sim
0.1	0.3154	0.0000	0.0315	-0.1961	-0.2191
0.3	0.3154	0.0000	0.0946	-0.1986	-0.1279
0.5	0.3340	0.0179	0.1581	-0.1023	-0.2041
0.7	0.3840	0.0720	0.2472	0.1978	0.1956
0.9	0.3935	0.1200	0.3422	0.4552	0.4084

TABLE 4: Cromlech decomposition for Tutored (maximum number of services=4): cohesion, operational cost, total value of the objective function, and similarity while changing α.

Table 4 presents a quantitative assessment of the decompositions we obtained when forcing a maximum number of 4 microservices, for different values of α. For each solution, we present the value of the cohesion metric, operational cost, and the total value of the objective function. We also present the similarity with the reference (manual) decomposition in terms of operations and data. Both the value of cohesion and the operational cost increase with α, indicating that the parameter works as expected: for small values of α, Cromlech sacrifices cohesion to avoid incurring a high operational cost, while for large values of α Cromlech targets higher cohesion values despite the increased operational cost. Overall, the value of the objective function increases with the organization-operations ratio α, which led us to think that increasing the number of microservices (which helps further improving organizational concerns) could yield better results.

Before confirming this hypothesis with more experiments, we compared Cromlech's solutions with the manual one (see again Table 2), observing that Cromlech provides better cohesion for $\alpha > 0.6$, while incurring a lower operational cost for any value of $\alpha < 1$. Cromlech's solutions become more similar to the manual decomposition in terms of operations as α increases (last two column in Table 4): this confirms that the manual decomposition privileges organizational (cohesion) over operational concerns.

Service	Num of ops	Cohesion
Content and activities	12	0.5008
Users	32	0.4056
Jobs	8	0.6839
Others	17	0.1583
Total	69	0.3935

TABLE 5: Cromlech decomposition for Tutored ($\alpha = 0.9$): number of operations and cohesion per service.

Afterwards, we manually analyzed the solutions generated by Cromlech in details. We present our findings and conclusions by referring in particular to the decomposition obtained with $\alpha = 0.9$, which provides a high cohesion while maintaining a low operational cost (about 12% of the maximum). Table 5 shows the number of operations and the cohesion of each service. The total cohesion metric (last line) is the average of the cohesion of each service weighted by the number of operations in that service. (1) *Content and activities* is similar to the corresponding microservice in the manual decomposition, but contains a subset of its operations (12 instead of 19), which results in a higher cohesion (0.5008 instead of 0.2459). (2) *Users* is much larger than the corresponding microservice in the manual decomposition (32 operations instead of 14). It also contains operations related to webinar and curricula, which frequently access the same data entities as the operations on users, in particular data entities belonging to the *Education* table. This choice reduces the operational cost with respect to the manual decomposition, with limited impact on cohesion despite the much larger number of operations (0.4056 instead of 0.5643). (3) *Jobs*, as in the case of *content and activities*, is a subset of the corresponding microservice in the manual decomposition, with 8 instead of 25 operations, and a higher cohesion (0.6839 instead of 0.2429). (4) *Others* contains 17 operations that appear to belong to at least 4 sets that are highly cohesive internally but not strongly coupled with each other (operations related to *skills*, *experience*, *academia*, and *languages*). This service negatively affects cohesion metric (with a value of 0.1583). It seems that Cromlech correctly identified cohesive blocks, but could not split them due to the forced limit in the number of microservices.

Moving from these considerations, we evaluated the value of a solution with 7 microservices, where we manually split *Others* into the 4 sub-components we identified (see Table 6). This solution retains the same operational cost of that with 4 microservices and increases cohesion from 0.3935 to 0.5038. These numbers indicate that the added services are indeed independent in terms of the data elements they consider (which increases the cohesion) and do not communicate with each others (which lowers the operational cost). In this scenario, Cromlech proved to be a valid tool to support decision making: despite the number of services was limited for the problem at hand, the proposed solution let us immediately identify the limitation and find a better decomposition.

Num serv.	Cohes.	Op cost	Total	Op sim	Data sim
7	0.5038	0.3838	0.4150	0.3879	0.4959

TABLE 6: Refinement of Cromlech decomposition with 7 services: cohesion, operational cost, total value of the objective function, and similarity with $\alpha = 0.9$.

α	Num serv.	Cohes.	Op cost	Total	Op sim	Data sim
0.1	15	0.4315	0.0000	0.0432	-0.090	-0.3907
0.3	15	0.5681	0.0300	0.1494	0.1338	-0.1160
0.5	15	0.6137	0.0625	0.2756	0.2251	0.0490
0.7	15	0.7551	0.2385	0.4570	0.4757	0.5977
0.9	15	0.8131	0.3491	0.6969	0.5422	0.7675

TABLE 7: Cromlech decomposition for Tutored (maximum number of services=15): cohesion, operational cost, total value of the objective function, and similarity while changing α.

As a final experiment, we run Cromlech increasing the maximum number of microservices to 15 (see Table 7). The results enable many interesting observations. Regardless of the value of α, Cromlech proposes decompositions that include all 15 microservices. The value of the objective function increases with α, again indicating the relevance of cohesion for the Tutored application. Compared to the solution with 7 microservices and considering the same value of $\alpha = 0.9$, the use of 15 microservices not only increases cohesion from 0.5038 to 0.8131, but also further decreases the operational cost from 0.3838 to 0.3491. Despite a much higher number of microservices, this decomposition has a higher similarity with the manual one in terms of operations for large

values of α (when Cromlech gives a higher weight to cohesion). This indicates that Cromlech correctly identifies the semantic similarities between operations that guided the manual decomposition (closely related operations remain co-located). Yet, it exploits the higher number of microservices to increase the cohesion.

Interestingly, Cromlech also decreases the operational cost: in absolute terms, the manual decomposition (with only 4 microservices) incurred an overall operational cost of over 0.4, and Cromlech manages to always keep it below 0.35 despite considering almost 4 times more services. As operational costs directly derive from dependencies between operations and data entities, they clearly indicate that using only 4 microservices (as suggested by the developers) yields sub-optimal results in terms of decomposition into independent units. Instead, the computational power accessible with an automated tool, better tackles the combinatorial nature of the problem and identifies all independent units without negatively affecting the operational cost.

Pangaea. To evaluate Pangaea, we use the same model of Tutored as in the paper that introduces Pangaea [20]. Like Cromlech, Pangaea offers a parameter to weight organizational and operational concerns: for our comparison we consider the reference scenario proposed in the original paper, giving the same weight to both concerns.

Num of serv.	Cohes.	Op cost	Total	Op sim	Data sim
4	0.306	0.2323	0.037	0.4092	0.1030

TABLE 8: Pangaea decomposition with 4 microservices and $\alpha = 0.5$: cohesion, operational cost, total value of the objective function, and similarity with the manual solution.

Table 8 shows the main metrics of Pangaea's solution. The solution presents a lower cohesion metric than the solution of Cromlech with 4 microservices for any value of $\alpha > 0$, and a higher operational cost for any value of $\alpha < 1$ (cfr Table 4). The total value of the objective function is higher in Cromlech for any value of $\alpha > 0.1$. Recall that these numbers are computed based on the data provided by the developers when compiling the model of the Tutored application, so they indicate that Cromlech better captures both organizational aspects (cohesion) and operational costs. In fact, the developers considered Pangaea as helpful to reason about operational costs, but they were not willing to use the decomposition proposed by the tool [20].

As in the case of Cromlech, the solution proposed by Pangaea presents some differences with respect to the manual decomposition, both in terms of operations and in terms of data. This seems to confirm how automated tools that aim to optimize multiple concerns find solutions that the developers did not consider. The low value of data similarity (0.1030, see Table 8) is particularly interesting. Indeed, Pangaea models applications based mainly on data and relations between data: despite the developers who provided the model were the same that proposed the manual solution, their modeling approach guided Pangaea to a different decomposition.

Beside considering numerical values, we also manually analyzed the quality of Pangaea's solution. The decomposition consists of the following microservices (also summarized in Table 9): (1) *Users* includes 30 operations: as in the case of Cromlech, it includes many operations that are associated to the central *education* data entities, such as operations related to *skills* and *languages*. (2) *Content and job offers* includes 27 operations, from two main subdomains (content and job offers, as the name implies). (3) *Experience* includes only 8 operations, but still includes operations and entities that are not always strictly related. (4) *Others* includes only 4 operations that cannot be easily ascribed to one business domain.

Service	Num of ops	Cohesion
Users	30	0.3370
Content and job offers	27	0.2351
Experience	8	0.3545
Others	4	0.4554
Total	69	0.306

TABLE 9: Pangaea decomposition for Tutored ($\alpha = 0.5$): number of operations and cohesion per service.

Despite some similarities with the solutions proposed by Cromlech, the microservices in the final decomposition are more difficult to describe in terms of business domains. The first two microservices are very large and include heterogeneous aspects. The last microservice does not represent a clear domain. Numerically, this reflects to relatively low values of cohesion for all microservices. The total value of the cohesion metric is increased by cohesion metric of *Others*, which benefits from containing only few operations. In comparison, Cromlech (see Table 5) proposed microservices with much higher cohesion, except one microservice (*Others*) that clearly includes more business domains. We conclude that Cromlech better captures the semantic relations between operations and clearly suggests to developers how to improve: further splitting some operations.

ServiceCutter. In our evaluation, we consider for ServiceCutter the same model adopted in the evaluation of Pangaea [20], which is the most detailed representation of the use case that ServiceCutter could successfully handle: models encoding entities at a finer granularity were too complex for ServiceCutter, which could not generate any valid solution due to runtime errors. This model primarily captures data entities and their mutual relations. When needed, we compute the optimal placement of operations based on the placement of the data entities they access.

Algorithm	Num serv.	Cohes.	Op cost
Girvan-Newman	2	0.1636	0
Leung	10	0.3701	0.3241
Chinese Whispers	5	0.3505	0.1714

TABLE 10: ServiceCutter decompositions for Tutored: cohesion and operational cost with different algorithms.

ServiceCutter may compute decompositions using three different clustering algorithms. Table 10 presents the characteristics of the decompositions that ServiceCutter generates for each clustering algorithm. The Girvan-Newman algorithm produces a solution that is very similar to the monolith, and only manages to identify one independent cluster of data entities and operations (related to the *Experience* database table), which slightly improves cohesion. The Leung algorithm improves cohesion by distributing data and operations across 10 microservices. However, the cohesion metrics remains lower than in any solution of Cromlech with 15 microservices (see Table 7), and comparable or lower than any solution of Cromlech with 4 microservices and $\alpha > 0.1$ (see Table 4). At the same time, the communication cost is higher than any solution of Cromlech with 4 microservices and $\alpha < 1$ and any solution of Cromlech with 15 microservices and $\alpha < 0.8$.

Service	Num of ops	Cohesion
Users	33	0.3239
Content	29	0.2275
Experience	4	1
Qualification	2	0.95
CareerDay	1	1
Total	69	0.306

TABLE 11: ServiceCutter decomposition for Tutored (Chinese Whispers algorithm): number of operations and cohesion per service.

We obtained the best solution using the Chinese Whispers algorithm. The solution consists of the following 5 microservices, also summarized in Table 11. (1) *User* is very similar to the one identified by Pangaea and includes 33 operations. (2) *Content* aggregates many small business domains (job offers, Webinars, streams, posts, events), with 29 operations. The heterogeneity of these domains reflects in a low cohesion score of 0.2275. (3) *Experience* includes 4 operations related to the *Experience* database table. (4) *Qualification* includes 2 operations related to the *Qualification* database table. (5) *CareerDay* only includes a single operation. The unbalanced distribution of entities reflects in a relatively high operational cost, higher than any decomposition of Cromlech with 4 microservices and $\alpha < 1$. Most importantly, the proposed decomposition appears to be not very useful for the developers, as many business domains are aggregated in 2 very large services, while the remaining services are very small and do not justify an independent unit of deployment.

In summary, for the Tutored application, the use of clustering algorithms as proposed by ServiceCutter produces decompositions that are either very similar to the monolith, or incur a high communication cost for the cohesion metric they provide, or are unbalanced and hence not very useful to guide the decision of the developers.

4.3 TrainTicket case study

Ops (before/after pre-proc)	125/124
Data entities (before/after pre-proc)	137/137
Transactional ops	16
Cohesion of the monolith	0.0961

TABLE 12: Main characteristics of the TrainTicket software architecture.

We selected TrainTicket as a second case study because it was conceived as a reference example of microservices architecture. It consists of 41 microservices implemented with four different programming languages (Java, JavaScript, Python, Go). We analyzed the codebase and performed a cleaning step to remove elements that could bias our study. In particular: (i) we removed services that only perform stateless computations without accessing or mutating any data entity (e.g., services only exposing verification operations); (ii) we removed duplicate services, containing two variants of the same operations, to be used as possible alternatives; (iii) we removed services used for administrative operations, as they are used only occasionally and should not be considered in the evaluation of the operational cost. In absence of other documentation, we considered all operations to have the same frequency. After this cleaning step, the architecture contains 27 services, 137 data entities, and 125 operations. The preprocessing step of Cromlech removed one additional operation, as it only accesses a single data entity. From the analysis of the project, we also identified 16 operations that in our opinion are only correct if executed with transactional semantics. As this is an assumption that we derived from the codebase and could not be fully verified, we opted for considering *both* a scenario in which these operations are not forced to be transactional *and* a scenario in which they are forced to be transactional. Table 12 summarizes the main characteristics of the TrainTicket software architecture.

To evaluate the decomposition tools, we give them in input the full set of data entities and operations and we observe how they split them into microservices.

Reference decomposition. The reference decomposition presents a very high cohesion (0.88 on a scale from 0 to 1), much higher than the reference decomposition in the Tutored scenario. This is probably due to the fact that TrainTicket has been designed as a microservice architecture. The operational cost is 0.53, which means 53% of the cost that a fully distributed solution (that does not consider any operation as being transactional) would incur. While this number may appear quite high, it is not surprising for a solution that splits 137 data entities across as much as 27 microservices (about only 5 data entities per microservice, on average), also considering that operations and data entities are densely connected in this use case. As a comparison, the manual decomposition in the Tutored use case had 0.41 operational cost despite consisting of only 4 services. As in Tutored, few data entities account for a large fraction of the operational cost. For instance, the *Order* data entity accounts for more than one third of the overall cost. However, the distribution is less skewed than in Tutored.

Number of services	27
Cohesion metric [0..1]	0.8822
Operational cost [0..1]	0.5328
Total value (with $\alpha = 0.5$) [-1..1]	0.3494

TABLE 13: Manual decomposition for TrainTicket: main characteristics.

In general, the analysis of this use case reveals a software architecture that is clearly conceived to distinguish business domains and associate them with microservices that are highly cohesive (as captured by the high value of our cohesion metric) despite requiring invocations to each other's operations (as captured by the relatively high value of the operational cost). This differentiates the TrainTicket use case from the previous Tutored use case, where the reference decomposition consisted of a limited number of services incorporating multiple business domains, obtaining a much lower value of cohesion (0.37).

Cromlech. Fig. 4 shows the value of the best decomposition found by Cromlech when we do not consider transactional operations and when we include them. In both cases, we consider a maximum number of 27 microservices and we set $\alpha = 0.9$. In absence of transactional operations, Cromlech has a large degree of freedom in associating operations to microservices, which results in a very high execution time: the value of the objective function periodically improves even after days of execution, and the values we report in the remainder of the paper were obtained with 10 days of processing. However, this is a very extreme and unrealistic scenario, as in practice a basic knowledge of the application let developers set some constraints in the form of transactional operations. Including transactional operations reduces the complexity of the problem by constraining the co-location of transactional operations and the data entities they access. Indeed, by

only considering 16 operations (out of 124) as transactional, the value of the objective function becomes stable after only 10 hours.

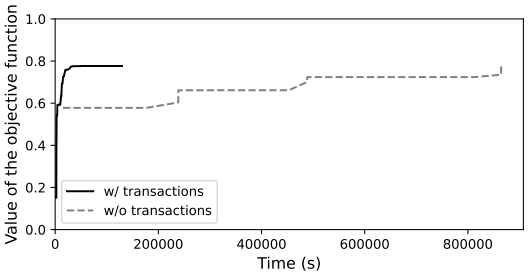

Fig. 4: Value of the best decomposition found over time (27 microservices, α=0.9)

α	Num serv.	Cohesion	Op cost	Total	Op sim	Data sim
0.1	11	0.5470	0.0149	0.0413	0.5581	0.5453
0.3	27	0.8089	0.0768	0.1889	0.8135	0.7050
0.5	27	0.8969	0.1252	0.3859	0.8707	0.7426
0.7	27	0.8407	0.2144	0.5242	0.8429	0.7681
0.9	27	0.8965	0.3247	0.7744	0.8922	0.8100

TABLE 14: Cromlech decomposition for TrainTicket without any transactional operations: cohesion, operational cost, and similarity.

Let us start our analysis from the case in which we do not consider transactional operations. Table 14 shows the main metrics of the decomposition computed by Cromlech. With a value of α greater then 0.1, Cromlech uses 27 services as the reference architecture, and the decomposition presents a very high similarity with the reference one both in terms of operations and data. This indicates that Cromlech could indeed replicate many of the choices of the reference architecture. Cromlech manages to significantly reduce operational costs for any value of α. For α greater than 0.1, the cohesion is also comparable to that of the reference decomposition (and sometimes even better). As a result, the objective function is higher than in the reference decomposition for α equal or greater than 0.5.

We manually analyzed the decomposition of Cromlech for α=0.9. We observed that the reduction in operational cost is mainly achieved by moving the data entities related to *Order*, which produce the highest operational cost in the reference architecture. These data entities are co-located with other entities that many operations access together. As a result, the total cost of the *Order* data entities is reduced by nearly a factor of 4, with no impact on cohesion.

α	Num serv.	Coh	Op cost	Total	Op sim	Data sim
0.1	8	0.3890	0.0040	0.0353	0.2279	0.4094
0.3	22	0.5690	0.0400	0.1427	0.4996	0.4429
0.5	27	0.8216	0.2000	0.3108	0.8148	0.4429
0.7	27	0.8640	0.2579	0.5274	0.8479	0.4513
0.9	27	0.9136	0.4494	0.7773	0.8867	0.4513

TABLE 15: Cromlech decomposition for TrainTicket considering 16 transactional operations: cohesion, operational cost, and similarity while changing α.

We repeated the experiment in a scenario in which we consider 16 transactional operations, as previously mentioned. As in the case of Tutored, we observe that parameter α works as expected: increasing α we obtain a better cohesion at the price of higher operational cost. For small values of α, Cromlech avoids operational costs by limiting the number of microservices, which reduces their internal cohesion. Only with $\alpha > 0.3$ Cromlech exploits all 27 microservices. For $\alpha > 0.7$, the decompositions have a higher cohesion than the reference one, and the operational cost is lower for any value of $\alpha < 1$. As in the previous use case, Cromlech manages to reduce operational cost with respect to the reference decomposition, indicating that developers tend to prioritize cohesion and may overlook operational costs due to the difficulty of computing them. At the same time, Cromlech still produces solutions that retain a high cohesion, comparable or higher than in the reference decomposition for $\alpha \geq 0.5$.

α	Num serv.	Cohesion	Op cost	Total	Op sim	Data sim
0.1	12	0.5478	0.2230	-0.1459	0.6913	0.6870
0.3	18	0.7411	0.1314	0.1305	0.8505	0.7967
0.5	21	0.8273	0.1983	0.3145	0.8733	0.8392
0.7	23	0.9004	0.347	0.5262	0.9329	0.8924
0.9	24	0.9056	0.4275	0.7723	0.9436	0.9027

TABLE 16: Pangaea decomposition for TrainTicket: cohesion, operational cost, and similarity while changing α.

Pangaea. We evaluated Pangaea using an input model with the same operations, entities, and relationships defined for Cromlech. Notice that in Pangaea entities where aggregated onto tables, whereas Cromlech works at the granularity of individual columns. As a result, the problem was much simpler for Pangaea and it always converged to a solution within tens of minutes. In all cases, we set a timeout of 2 hours. We retrieved five decompositions for different values of α, which were evaluated using the same metrics employed for Cromlech. The results, reported in Table 16, show that Pangaea obtained similar results compared to Cromlech. In particular, if we consider the results of Cromlech reported in Table 15, for low values of α, Pangaea performed better in terms of cohesion but with higher operational cost. On the contrary, for high values of α, Pangaea obtained lower operational costs but also a lower cohesion.

The decompositions produced by Pangaea appear to be similar to the reference solution. As for Cromlech, the operational similarity is quite high (e.g., 0.9436 for $\alpha = 0.9$). This demonstrated the goodness of the reference solution, being the system originally conceived with a microservice architecture. Pangaea solutions obtained also a high data similarity (e.g., 0.9027 for $\alpha = 0.9$), while Cromlech obtained lower values (e.g., 0.4513 for $\alpha = 0.9$). This seems to be due to the fact that, unlike Pangaea, Cromlech exploits data replication more frequently, resulting in decompositions that are less similar to the reference solution but incur lower operational cost.

ServiceCutter. To evaluate ServiceCutter, we defined an input model that includes data entities, operations, and their mutual relations at the same granularity as in the input model used for Cromlech, without defining any transactional operation. ServiceCutter was tested using the Girvan-Newman, Leung, and Chinese Whispers algorithms, all in their default configurations with all the criteria set to have equal weights (corresponding to $\alpha = 0.5$ in our approach). For Girvan-Newman, we also limited the number of services to 27, in alignment with our earlier observations.

Table 17 presents the results we obtained. Girvan-Newman's decomposition yielded 26 services, closely aligning with the maximum value set. It obtained a cohesion score of 0.8436, which is lower than both the reference decomposition (0.8822) and Cromlech's score of 0.8969 (with α=0.5 and no transactional

Algorithm	Num serv.	Coh	Op cost	Total	Op sim	Data sim
Girvan-Newman	26	0.8436	0.4993	0.1721	1.000	0.589
Leung	9	0.5037	0.0570	0.2233	0.654	0.589
Chinese Whispers	15	0.6557	0.0855	0.2850	0.751	0.589

TABLE 17: ServiceCutter decompositions for TrainTicket without any transactional operations: cohesion, operational cost, and similarity.

operations). In terms of operational costs, this solution outperformed the reference decomposition (achieving 0.3135 compared to 0.5328) but not Cromlech that produced lower values for any values of $\alpha < 0.9$. Leung's decomposition generated 9 services, prioritizing operational costs (as low as 0.0580) over cohesion (0.5037). The Chinese Whispers algorithm, on the other hand, slightly improves cohesion to 0.6555, yet it remains below the reference and Cromlech scores, while maintaining low operational costs (0.0855). In terms of data similarity, all algorithms achieved identical scores, indicating a consistent approach across different methods. Thus, the variations in the scores lie in the different ways operations are grouped. In this respect, Girvan-Newman's decomposition perfectly matches the reference implementation, leading to an operations similarity of 1. If we consider the total score, no matter the algorithm, Cromlech (with $\alpha = 0.5$ and without any transactional operations) outperformed all the algorithms, demonstrating its ability in optimizing both cohesion and operational costs in a balanced way.

In our pursuit to refine the assessment of ServiceCutter and its comparison with Cromlech, we carried out an additional analysis leveraging 4 different configurations for each algorithm. In particular, we identified two tuning parameters that mirror our conceptual model: "Identity and Lifecycle Similarity" (abbreviated as *ide*) as a proxy for measuring cohesion and "Latency" (abbreviated as *lat*) to represent operational costs. While the previous experiments were run with default settings, that is, all the parameters set to "M" (in a range from "XS" to "XXL"), for this analysis we run the experiments with extreme values ("XS" and "XXL") to capture the full spectrum of outcomes and understand the trade-offs at play. We also evaluated the total score with three values of α (0.1, 0.5, 0.9) to understand the effects of varying emphases on cohesion versus operational costs within our evaluation framework.

The results reported in Table 18 indicate a significant difference in the sensitivity of the algorithms to the tuning parameters *ide* and *lat*. For the Girvan-Newman algorithm, the outcomes in terms of the number of services, cohesion, and operational cost remained invariant despite changing the configurations for *ide* and *lat*. In contrast, the Chinese Whispers and Leung algorithms exhibited variability with different configurations, but the effects were not uniform. For Chinese Whispers, increasing the *ide* parameter from XS to XXL resulted in an increase in the number of services, which aligns with the expectation that multiple and smaller services would yield higher cohesion. However, changes in the *lat* parameter did not alter the number of services, which is counterintuitive since one would expect latency optimization to impact service granularity. In contrast, the Leung algorithm showcased an opposite behaviour since altering the *ide* parameter did not affect the number of services, whereas adjustments to *lat* did. This unpredictability underscores the algorithm-dependent nature

Algorithm	Conf	Num serv.	Coh	Op cost	Total		
					$\alpha=0.1$	$\alpha=0.5$	$\alpha=0.9$
Girvan-Newman	*ide*=XS	26	0.8436	0.4993	-0.3650	0.1721	0.7093
Girvan-Newman	*ide*=XXL	26	0.8436	0.4993	-0.3650	0.1721	0.7093
Girvan-Newman	*lat*=XS	26	0.8436	0.4993	-0.3650	0.1721	0.7093
Girvan-Newman	*lat*=XXL	26	0.8436	0.4993	-0.3650	0.1721	0.7093
Chinese Whispers	*ide*=XS	12	0.5453	0.0297	0.0277	0.2577	0.4877
Chinese Whispers	*ide*=XXL	22	0.7203	0.3853	-0.2748	0.1674	0.6097
Chinese Whispers	*lat*=XS	16	0.7655	0.2503	-0.1487	0.2575	0.6639
Chinese Whispers	*lat*=XXL	16	0.7297	0.2416	-0.1445	0.2440	0.6325
Leung	*ide*=XS	8	0.3980	0.0123	0.0286	0.1928	0.3569
Leung	*ide*=XXL	8	0.4306	0.0793	-0.0283	0.1756	0.3796
Leung	*lat*=XS	5	0.2948	0.0966	-0.0575	0.0990	0.2556
Leung	*lat*=XXL	9	0.4730	0.0297	0.02053	0.2216	0.4227

TABLE 18: ServiceCutter decompositions under different configurations.

of the configuration effects and suggests that understanding the internal mechanics of each algorithm is crucial for proper tuning.

Furthermore, it is important to note that none of the configurations explored for these algorithms could exceed the total score (0.7744) achieved by the best configuration of the Cromlech ($\alpha = 0.9$, without any transactional operations). This suggests that, despite the complex and fine-grained control offered by ServiceCutter, Cromlech achieves superior performance with a more straightforward and intuitive parameter (the α factor) which provides practitioners with a more direct and deterministic approach to balance cohesion against operational costs.

4.4 Discussion

We learned several lessons from the analysis above. First of all that the two case studies present significant differences. In Tutored, developers provide a reference decomposition with a small number of services, clearly insufficient to isolate independent business domains. Based on the input information about the system, the tools we analyzed provide very diverse answers. Pangaea generates services with heterogeneous sizes, where it is difficult to identify business domains. ServiceCutter is heavily influenced by the algorithm adopted: some algorithms provide trivial solutions (close to a monolith) or solutions that incur high operational costs. Cromlech identifies the business domains and clearly suggests the use of a higher number of services by aggregating multiple business domains into one service.

Instead, TrainTicket is explicitly conceived to represent a reference microservices architecture. Indeed, all the approaches we considered (Cromlech, Pangaea, and ServiceCutter) produce decompositions that are similar to the reference. However, Cromlech manages to further reduce the operational cost — also exploiting data replication — while retaining a separation of business domains that is close to the reference. Adjusting the configuration parameters of ServiceCutter has the potential to optimize outcomes for either increased cohesion or reduced operational costs. However, our findings demonstrate that this tuning process is highly dependent on the specific algorithm in use and is far from being straightforward. In contrast, Cromlech offers an easily adjustable parameter that simplifies the balancing between cohesion and operational cost, providing a user-friendly interface for practitioners seeking effective optimization.

In summary, in both cases Cromlech guides towards solutions that preserve the semantic relations between operations (organizational concerns) while balancing them with operational concerns. The latter are overlooked in previous literature, as the results of ServiceCutter indicate. Pangaea introduces an explicit trade-off between organizational and operational concerns, but its simple modeling approach, based on the relations between data entities, does not yield results that are comparable to Cromlech (as we observe in the case of Tutored). We think that the main advantages of Cromlech with respect to Pangaea in terms of modeling are: the focus on operations, which better reflect business domains, the definition of constraints for transactional semantics and replication that more closely reflect their concrete implementation in microservices architectures.

4.5 Threats to validity

These are the threats that may hurdle the validity of our results [28].

Internal threats. When comparing Cromlech with other approaches, we used the same definitions of cohesion and communication cost we use to compute Cromlech's objective function. This can create a bias in the assessment, since Cromlech is built with the goal of optimizing these metrics while other tools are not. However, these metrics are defined using well established criteria for evaluating microservices architectures such as cohesion, inter-service communication, and data replication, broadly discussed in the literature [9], [10], [11]. Also, their values directly derives from a user-defined system model that indicates the relations between data entities and operations in an unambiguous way. Moreover, by changing the organization-communication ratio α, we explored a broad range of cases, from the ones that favor highly decentralized solutions to those that favor larger microservices, and Cromlech consistently outperformed other solutions in almost all scenarios. Finally, our qualitative analysis confirms the quantitative one, showing that the solutions provided by Cromlech are well balanced and able to capture the semantic of business domains.

External threats. Cromlech was tested with two use cases, threatening the generalization of results. While we plan to extend our evaluation to more scenarios in the future, the use cases are representative of a real-world system and a reference architecture for microservices. To the best of our knowledge, they do not have any peculiar aspect that could invalidate the claims in the paper. They also show complementary characteristics: Tutored was originally implemented as a monolithic application, whereas TrainTicket was designed as a microservices architecture.

Construct and conclusion threats. The experiments presented in this section show that Cromlech is able to outperform alternative approaches in the tested use cases and that its behavior is sensible, but yet predictable, to the input parameters. Cromlech helps in evaluating the trade-offs that arise in the monolith decomposition process and users can iteratively exploit its insights by changing the inputs to test different scenarios.

5 Related Work

As many developers choose microservices as their reference architecture, the decomposition problem becomes increasingly relevant. While decomposition remains a tedious manual task performed by experts with a deep knowledge of the system, over the last few years developers and practitioners have proposed decomposition criteria, methodologies, and tools to simplify the process. This section presents these proposals and compares them with Cromlech.

As mentioned in Section 1, our work extends Pangaea [20] and shifts the focus from data entities to operations to better capture organizational concerns both in its modeling framework and in the MILP problem formulation. Our evaluation shows the benefits of the new approach, providing decompositions that are easier to understand by practitioners and more efficient from an operational point of view.

ServiceCutter [21] is probably the work that most closely resembles our proposal. Like Cromlech, ServiceCutter relies on a model of the software system that considers data elements and operations, together with a wide range of relations between them. Although many of these criteria are optional, ServiceCutter still requires more input from users compared to Cromlech. Decomposition choices are based on clustering algorithms, and developers can select various algorithms. We used ServiceCutter as a reference for our evaluation in Section 4: our results indicate that Cromlech generates decompositions that are less expensive in terms of operations cost and more insightful for developers.

Levcovitz et al. [16] propose a technique to identify microservices starting from a monolithic enterprise system. They describe the system in terms of three types of entities: (i) facades (entry points of the system), (ii) business functions, (iii) database tables. They define a rigorous six-steps process to define microservices, which considers the dependencies between the entities above, as well as their relations to business areas and processes. As Cromlech, the approach dictates a structured and precise decomposition methodology, but it is manual and does not consider operational concerns, which are key to create an efficient decomposition.

Laigner et al. [15] extended the above methodology in two main ways: (i) they take in input a repository of functions and output concrete code for microservices in the form of relational actors; (ii) they automate allocation of functions to services, using a MILP approach. With respect to Cromlech, the proposed methodology is limited to three-tiered REST-based architectures and retains a long preparatory manual procedure to define the dependencies among entities.

Chen et al. [29] propose a semi-automatic, dataflow-driven approach: developers need to manually construct a simplified dataflow diagram for business logic functionalities and apply some preparatory steps to make the diagram amenable to automatic decomposition. later, an automated algorithm identifies microservices candidates from the diagram. The approach outputs very fine-grained microservices, and further manual intervention may be necessary to combine some of them into larger units.

Two approaches use static and dynamic analysis to infer a model of the software system. Mazlami et al. [17] rely on static analysis of a software repository, which considers not only the structure of the code, but also how the different developers contributed to the project. This enables inferring coupling relations between components that are both technical and managerial. Taibi and Sista [13] introduce a data-driven approach based on process mining. The decomposition starts from log files collected from a monolith at runtime and infers functionalities and data entities from frequent execution paths. It outputs a visual representation of the main building blocks of the system together with some candidate proposals for

decomposition, which still require manual validation. We plan to explore static and dynamic analysis as future work, to (partially) infer the system model and further simplify its definition.

The technique by Baresi et al. [18] automatically infers a monolith decomposition starting from an OpenAPI specification. It matches terms in the specification against a reference vocabulary to determine their semantic similarity. In this way, it identifies and groups together operations that share the same reference concepts, with the goal of maximizing cohesion. We see this technique as orthogonal to Cromlech: semantic similarity could help to infer coupling between system components, which is an input parameter for Cromlech.

The work presented by Ntentos et al. [30] assumes an existing decomposition that is iteratively improved using three different metrics: i) persistent data storage of services, that is how coupled are microservices in terms of data entities, ii) *service interconnections*, which measures the effect of intermediary components and protocols that allow inter-service communications, and iii) *dependencies through shared services*, that is the direct or transitive dependencies among microservices. Their approach measures these metrics at every code base update (e.g., at each commit onto Continuous Integration pipeline) and automatically detects violations from set thresholds. When a violation occurs, the approach automatically computes a set of architectural modifications based on the aforementioned three metrics that are proposed to the software architect. This work is also complementary with Cromlech and can be use to refine an initial decomposition generated with our approach. However, compared to Cromlech, it does not consider organization concerns and the cost and benefits of data replication.

Sellami et al. [31] present MSExtractor, an approach to decompose monoliths written in an object-oriented (OO) programming language in microservices. The application is conceived as a set of classes that are automatically classified as either *inner*, if they are only used as internal components, or *interfaces*, if they expose public endpoints to the users. Through this classification and a semantic analysis of the code, they formulate an optimization problem that aims to i) maximize the cohesion within each microservice, and minimize the coupling among different microservices. To find a close to optimal solution, they employ a search-based algorithm, namely IBEA (Indicator-Based Evolutionary Algorithm). Compared to this work, Cromlech can be used with any existing monolith and not only OO ones. Moreover, MSExtractor does not explicitly consider communication and replication costs.

The approach presented by Selmadji et al. [14] automatically analyzes the source code of a monolith and, along with optional expert recommendations, it generates a decomposition into microservices. They use a heuristic and a clustering technique that consider both organizational and data-related aspects of the monolith with the aim of maximizing the quality of each generated microservice. Compared to Cromlech, they do not consider operational costs and neither they allow data entities to be replicated in different microservices. Moreover, their technique relies on a static analysis that assumes an application to be written in object-oriented style, while Cromlech supports any kind of application.

6 Conclusions

Devising a suitable decomposition of an application into microservices is key to exploit the potentials of microservices architectures. Unfortunately, decomposition is a complex task that encompasses both technical and managerial concerns, and it is hard for developers to weight the benefits and shortcomings of a given decomposition choice.

This paper introduced Cromlech, a semi-automated tool that helps developers decomposing an application into microservices. Cromlech builds on a simple but informative model of the application and formulates an optimization problem that balances organization, communication, and data management requirements. It outputs a graphical representation of the proposed decomposition together with detailed data on the costs it incurs.

Our evaluation on a real-world application shows the validity of Cromlech, which offers useful insights to developers. We plan to further develop the research along several directions: (i) refine the modeling approach to enable a more fine-grained modeling when suitable; (ii) integrate static analysis and monitoring tools to automatically generate the system model; (iii) evaluate alternative solving strategies (for instance, heuristics) to improve performance and scalability of the tool to more complex models; (iv) extend the visualization tool, offering editing capabilities for interactive adjustments of the solution; (v) investigate the use of Cromlech as a decision-support tool for the evolution and refactoring of an existing microservice-based software system.

References

[1] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, *Microservices: Yesterday, Today, and Tomorrow*. Springer, 2017, pp. 195–216.

[2] S. Newman, *Monolith to Microservices*. O'Reilly, 2020.

[3] J. Thönes, "Microservices," *IEEE software*, vol. 32, no. 1, pp. 116–116, 2015.

[4] J. Lewis and M. Fowler, "Microservices, a definition of this new architectural term," 2016. [Online]. Available: https://martinfowler.com/articles/microservices.html

[5] O. Zimmermann, "Microservices tenets," *Computer Science-Research and Development*, vol. 32, no. 3, pp. 301–310, 2017.

[6] A. Balalaie, A. Heydarnoori, P. Jamshidi, D. A. Tamburri, and T. Lynn, "Microservices migration patterns," *Software: Practice and Experience*, vol. 48, no. 11, pp. 2019–2042, 2018.

[7] S. Newman, *Monolith to microservices: evolutionary patterns to transform your monolith*. O'Reilly Media, 2019.

[8] J. Fritzsch, J. Bogner, S. Wagner, and A. Zimmermann, "Microservices migration in industry: intentions, strategies, and challenges," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2019, pp. 481–490.

[9] A. Singleton, "The economics of microservices," *IEEE Cloud Computing*, vol. 3, no. 5, pp. 16–20, 2016.

[10] P. Di Francesco, P. Lago, and I. Malavolta, "Architecting with microservices: A systematic mapping study," *Journal of Systems and Software*, vol. 150, pp. 77–97, 2019.

[11] M.-D. Cojocaru, A. Uta, and A.-M. Oprescu, "Attributes assessing the quality of microservices automatically decomposed from monolithic applications," in *2019 18th International Symposium on Parallel and Distributed Computing (ISPDC)*. IEEE, 2019, pp. 84–93.

[12] R. Laigner, Y. Zhou, M. A. V. Salles, Y. Liu, and M. Kalinowski, "Data management in microservices: State of the practice, challenges, and research directions," *Proc. VLDB Endow.*, vol. 14, no. 13, p. 3348–3361, 2021.

[13] D. Taibi and K. Systä, "From monolithic systems to microservices: A decomposition framework based on process mining," in *Proceedings of the International Conference on Cloud Computing and Services Science*, ser. CLOSER '19, V. M. Muñoz, D. Ferguson, M. Helfert, and C. Pahl, Eds. SciTePress, 2019, pp. 153–164.

[14] A. Selmadji, A.-D. Seriai, H. L. Bouziane, R. Oumarou Mahamane, P. Zaragoza, and C. Dony, "From monolithic architecture style to microservice one based on a semi-automatic approach," in *2020 IEEE International Conference on Software Architecture (ICSA)*, 2020, pp. 157–168.

[15] R. Laigner, S. Lifschitz, M. Kalinowski, M. Poggi, and M. A. V. Salles, "Towards a technique for extracting relational actors from monolithic applications," in *Simpósio Brasileiro de Banco de Dados*, ser. SBBD '19. SBC, 2019, pp. 133–144.

[16] A. Levcovitz, R. Terra, and M. T. Valente, "Towards a technique for extracting microservices from monolithic enterprise systems," 2016.

[17] G. Mazlami, J. Cito, and P. Leitner, "Extraction of microservices from monolithic software architectures," in *Proceedings of the International Conference on Web Services*, I. Altintas and S. Chen, Eds. IEEE, 2017, pp. 524–531.

[18] L. Baresi, M. Garriga, and A. D. Renzis, "Microservices identification through interface analysis," in *Proceedings of the European Conference on Service-Oriented and Cloud Computing*, ser. ESOCC '17, F. D. Paoli, S. Schulte, and E. B. Johnsen, Eds. Springer, 2017, pp. 19–33.

[19] A. K. Kalia, J. Xiao, R. Krishna, S. Sinha, M. Vukovic, and D. Banerjee, *Mono2Micro: A Practical and Effective Tool for Decomposing Monolithic Java Applications to Microservices*. ACM, 2021, p. 1214–1224.

[20] S. Staffa, G. Quattrocchi, A. Margara, and G. Cugola, "Pangaea: Semi-automated monolith decomposition into microservices," in *Proceedings of the International Conference on Service-Oriented Computing*, ser. ICSOC. Cham: Springer International Publishing, 2021, pp. 830–838.

[21] M. Gysel, L. Kölbener, W. Giersche, and O. Zimmermann, "Service cutter: A systematic approach to service decomposition," in *Proceedings of the European Conference on Service-Oriented and Cloud Computing*, ser. ESOCC '16. Springer, 2016, pp. 185–200.

[22] A. Bellemare, *Building Event-Driven Microservices*. O'Reilly, 2020.

[23] F. Glover and E. Woolsey, "Converting the 0-1 polynomial programming problem to a 0-1 linear program," *Operations Research*, vol. 22, no. 1, pp. 180–182, 1974.

[24] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, W. Li, and D. Ding, "Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study," *IEEE Transactions on Software Engineering*, vol. 47, no. 2, pp. 243–260, 2021.

[25] M. E. Newman and M. Girvan, "Finding and evaluating community structure in networks," *Physical Review E*, vol. 69, no. 2, 2004.

[26] I. X. Leung, P. Hui, P. Lio, and J. Crowcroft, "Towards real-time community detection in large networks," *Physical Review E*, vol. 79, no. 6, 2009.

[27] C. Biemann, "Chinese whispers: An efficient graph clustering algorithm and its application to natural language processing problems," in *Proceedings of the Workshop on Graph Based Methods for Natural Language Processing*, ser. TextGraphs '06. USA: Association for Computational Linguistics, 2006, pp. 73–80.

[28] C. Wohlin *et al.*, "Empirical research methods in web and software engineering," *Web Engineering*, 2006.

[29] R. Chen, S. Li, and Z. Li, "From monolith to microservices: A dataflow-driven approach," in *Proceedings of the Asia-Pacific Software Engineering Conference*, ser. APSEC 2017. IEEE, 2017, pp. 466–475.

[30] E. Ntentos, U. Zdun, K. Plakidas, and S. Geiger, "Semi-automatic feedback for improving architecture conformance to microservice patterns and practices," in *2021 IEEE 18th International Conference on Software Architecture (ICSA)*, 2021, pp. 36–46.

[31] K. Sellami, A. Ouni, M. A. Saied, S. Bouktif, and M. W. Mkaouer, "Improving microservices extraction using evolutionary search," *Information and Software Technology*, p. 106996, 2022.

Simone Staffa received his Master's Degree in Computer Engineering at Politecnico di Milano. He is currently a Backend Engineer at BackdropLabs, a Web3 startup focused on Blockchain-based systems. His research interests include distributed systems, software architectures, and decentralized applications.

Giovanni Quattrocchi received his Ph.D. in Computer Engineering in 2018 from Politecnico di Milano, where he is currently a post-doc researcher. He was a visiting researcher at University of California San Diego and Imperial College London. His research interests include self-adaptive systems, software architectures, edge computing, and blockchain-based systems.

Alessandro Margara is an Associate Professor at Politecnico di Milano. He received his PhD in Information Technology from Politecnico di Milano and was a postdoctoral researcher at the Vrije Universiteit (VU) Amsterdam and at the Università della Svizzera italiana (USI). His research interests include event stream processing and software engineering for distributed and data-intensive systems.

Davide Cocco received his Master's Degree in Computer Engineering at Politecnico di Milano in 2022. He is currently a Software Engineer at Oracle Labs, Zurich. His research interests include software architectures, automated Machine Learning, and highly distributed computations.

Gianpaolo Cugola is a Full Professor at Politecnico di Milano. He received his Dr.Eng. degree in Electronic Engineering from Politecnico di Milano in 1994 and his Ph.D. in Computer Science in 1998. His research interests are in the area of software engineering and distributed systems. In particular, his current research focuses on middleware technology for largely distributed and highly reconfigurable distributed applications.