1. Introduction

The need to evolve software systems made many IT practitioners migrate from monolithic architectures to microservices [2]. A monolith is a software system built as a single unit of deployment, with all functionalities offered by a single executable, developed with a single programming language. They present severe limitations when applications grow in scale [7]: the dependencies among components damage maintainability; the deployment of a single complex executable is slow and the only way to scale the system is through vertical scaling or by replicating the entire system; it also presents a single point of failure. Instead, microservices architectures [5, 8] define an application as a composition of independent units denoted as microservices, containing a subset of logically-related functionalities, and are developed, deployed, and maintained independently from each other [6]. Providing a suitable decomposition of a monolithic architecture into microservices is difficult as it requires balancing a suitable level of decoupling between services and internal cohesion with the need of keeping a consistent state for critical data and coherently functioning distributed transactions, which requires communication between the services that increases the more distributed the architecture becomes. A good decomposition helps in reducing the latency associated to cross-service communication.

The decomposition of a monolith into microservices encompasses both technical and managerial concerns. Since microservices are organized around business capabilities: the decomposition needs to produce microservices that are highly cohesive and include all the data and processing components that enable implementing a given capability. Moreover, microservices are developed as independent executables that do not share memory. Depending on the specific adopted technology, they communicate using synchronous remote procedure calls or asynchronous propagation of messages. Communication across microservices may increase the overall response time: thus microservices should be loosely-coupled. Microservice architectures also decentralize data management by design. Each microservice has its partial view of the application domain, and data integrity and consistency in the presence of concurrent operations and data replication are enforced at the application level, and this may require coordination pro-
Executive summary

protocols across microservices that introduce coupling, and may degrade performance at runtime. An effective decomposition should be aware of data integrity constraints and avoid costly coordination by co-locating related data elements within the same microservice. In summary, a suitable decomposition process should capture and balance organization, communication, and data management requirements. Moreover, it should permit developers to customize the weights of such requirements based on their priorities. Most SOTA solutions lack metrics to evaluate the solution, require long manual work by the user, or do not model inter-service communication. Focusing on cohesion and decoupling rather than reducing latency in communications between services. Distributed transactions are never taken into account when performing the decomposition, and assume a solution with unregulated replication of data entities. Consequently, we designed a tool called **Cromlech** which starting from an high level description of the endpoints of the architecture and the data they access, uses a Linear Programming algorithm to produce a decomposition that balances decoupling between the services (and high internal cohesion) and low cross-service communication. The description of the system is provided through a YAML file, and the user can choose a maximum number of services and a weighted objective function for the LP model to modulate the tradeoff between decoupling/cohesion and cross-service communication. It finally offers a visual representation of the proposed decomposition together with a detailed analysis of its costs.

2. **Cromlech**

Developers need to provide a system model defining data entities and operations that build the application, with their characteristics and mutual relations and a set of input parameters that steer the decomposition process based on user preferences. **Cromlech** works in three steps:

1. a parser translates the system model into an optimization problem;
2. a mixed integer linear programming solver outputs a possible allocation of data entities and operations onto microservices;
3. a visualizer produces a visual representation of the proposed decomposition together with a detailed analysis of its costs.

2.1. System model

**Cromlech** enables developers to model an application in terms of data entities and operations. The system model is provided in YAML\(^1\), a data specification language frequently used for configuration files.

**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 be used to model a single table: **Cromlech** will treat the table as an unbreakable unit and map it to microservices accordingly.

A data entity \(e\) is characterized simply by its name as the information relevant to the optimization problem is contained in the operation section.

**Operations** represent units of execution of the application, which are candidate to become logic functionalities of microservices. Each accesses (reads and writes) data entities and is associated to a single microservice. An operation \(o\) is characterized by its name, the list of entities it accesses and in which modality (read or read-write), its frequency (an integer representing how often it is invoked), a boolean called transactional (which indicates if the operation has transactionality requirements), and a list of other operations, called colocated operations which the user can utilize to specify which operations must be in the same service with the considered one.

2.2. Optimization problem

**Cromlech** combines the system model and input parameters provided by developers to formulate an optimization problem which balances two opposing forces:

(i) The cohesion metric of the architecture, representing the cohesion within individual services and the decoupling between different services. It favours decentralized solutions, where each ser-

\(^1\)https://yaml.org
service is only responsible for a limited and highly-cohesive set of data entities and operations.

(ii) The communication costs that derive from cross-service communication and favour solutions that centralize data entities and operations.

The problem defines an objective function where the two components are weighted by a parameter set by developer to define the relative importance of each component. The goal of the solver is to find an optimal allocation of operations and entities onto a set of microservices that maximizes cohesion without incurring in excessive communication costs.

This problem can be formalized 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 architecture, we assume a single writer principle, meaning that a single service is responsible for all updates (write accesses) to a given data entity $e \in E$. Let us call such service as the leader replica for $e$. Accordingly, we use an additional variable $l$ to encode the placement of the leader replica.

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

**Input parameters** Cromlech takes in input a small number of parameters that guide the decomposition process based on the requirements of developers.

- number of microservices as the maximum number of microservices that the decomposition can use.
- cohesion-communication ratio a real number $\alpha$ indicating the importance developers attribute to the cohesion metrics over the reduction of cross-service communication, on a scale between 0 and 1. Increasing the value would favor solutions where microservices are highly cohesive (potentially at the cost of increased communication).

**Cohesion metric** 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:

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

Where $S(o1, o2)$ is the similarity between operations $o1$ and $o2$. A high similarity indicates that two operations belong to the same business domain, thus placing them onto the same microservice increases the cohesion metric. Our assumption is 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 in write mode. 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.

$$\forall o \in O \ \forall e \in E \ e \in E_o \leftrightarrow acc_{e,o} = 1$$

$$S(o1, o2) = \frac{|E_{o1} \cap E_{o2}|}{\min(|E_{o1}|, |E_{o2}|)}$$

The cohesion metric for the entire architecture is the sum of the cohesion metrics for each service $m \in M$, weighted by the number of operations in $m$ with respect to all operations.

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

**Communication costs** The communication costs measure the overhead of cross-service communication.

Our model enables data entities to be replicated at different microservices. Replication is widely used in microservice architectures to improve availability and reduce latency, as operations may access locally replicated data without incurring the cost of remote data access. Hence
communication costs include two aspects: the costs of remote data access and the cost of replication. Remote data access occurs when an operation \( o \in O \) needs to access a data entity \( e \in E \) but the two are placed on different microservices. Replication costs involve keeping remote replicas up-to-date.

Since we assume a single writer principle, for a given data entity \( e \in E \), a single service (the leader replica) is responsible for all updates (write accesses) to \( e \). Otherwise, it is the cost of invocation of \( e \).

Formally, operations incur the following communication costs. \( R_{o,e} \) is 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 \( o \).

For each data entity \( e \in E \) and for each operation \( o \in O \), let us denote \( accR_{e,o} \) and \( accRW_{e,o} \) boolean variables that hold 1 if and only if \( o \) accesses \( e \) in read-only or in read-write mode, respectively. We extract their values from the data access property of operations in the input model.

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

\( W_{o,e} \) is the cost that \( o \in O \) incurs for writing a data item \( e \in E \). Due to the single writer principle, 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.

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

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

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

The total communication costs for a given placement of entities and operations onto microservices are defined as:

\[
Com = \sum_{o \in O, e \in E} (R_{o,e} + W_{o,e}) + \sum_{e \in E} Repl_e
\]

**Objective function and constraints**

The goal of the optimization problem is to maximize cohesion (\( Coh \)) and minimize communication cost (\( Com \)). The cohesion metric is a real number between 0 and 1: for ease of use, we also normalize the communication costs to a scale between 0 and 1. To do so we need the maximum and minimum possible communication costs: the minimum is obtained from the monolith and is always null (no cross-service communication), while the maximum is obtained by the most distributed version possible. Let us define the maximum communication costs as \( Com_{\text{max}} \).

The objective becomes:

\[
Obj = \alpha \cdot Coh - (1 - \alpha) \cdot \frac{Com}{Com_{\text{max}}}
\]

The following constraints must be satisfied:

**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.**

\[
\forall e \in E, m \in M y_{e,m} \geq l_{e,m}
\]

**Operations which the user specified to be co-located must be in the same microservice.**

\[
\forall o_1 \in O, o_2 \in O \sum_{m \in M} x_{o1,m} \cdot x_{o2,m} \geq \text{coloc}_{o1,o2}
\]

Where \( \text{coloc}_{o1,o2} \) is true iff \( o1 \) and \( o2 \) are colocated. **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. If an operation requires 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.

\[ \forall o \in O, e \in E, m \in M \times_{o,m} \geq tr_o \cdot acc_{e,o} \cdot l_{e,m} \]

Where \( tr_o \) is true iff \( o \) has transactionality requirements.

3. Results

We tested Cromlech against two architectures:
- a 70 operation monolith belonging to Tutored
- a 124 operation benchmark, already in microservice format, called Trainticket[4]

To measure the similarity between two decompositions we developed an algorithm that, for each operation pair, counts the times where the pair is in the same service in both decompositions (both_present), the times where the pair is not in the same service in both decompositions (both_absent) and the times where the pair is in the same service only in the first decomposition (only_first) and in the second (only_second).

The similarity between decompositions \( D_1 \) and \( D_2 \) is then expressed as:

\[ S(D_1, D_2) = \frac{(\text{both}_\text{present} + \text{both}_\text{absent} - \text{only}_\text{first} - \text{only}_\text{second})}{\text{total}_\text{pairs}} \]

3.1. Tutored

For this case we aimed to:
- provide decompositions at varying levels of alpha to test its effectiveness
- compare Cromlech’s decompositions against two tools: ServiceCutter[3] and Pangaea, with our metrics and with Pangaea’s metrics
- measure the similarity of Cromlech’s decomposition with the manual decomposition made by Tutored’s developers

The architecture has 4 transactional operations.

<table>
<thead>
<tr>
<th>ops</th>
<th>entities</th>
<th>monolith coh.</th>
<th>worst comm. cost</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>69</td>
<td>166</td>
<td>0.1579</td>
<td>1568</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Metrics of Tutored’s architecture.

The manual solution has 4 services and scores well in cohesion but poorly in communication costs, separating many operations accessing columns of the Education entity, which composes 71.6% of communication costs.

<table>
<thead>
<tr>
<th>comm. cost</th>
<th>coh.</th>
<th>abs. score</th>
<th>services</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.4094</td>
<td>0.3715</td>
<td>-0.0379</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 2: Tutored’s manual solution metrics.

We measured multiple decompositions by Cromlech capped at a maximum of 4 services with levels of alpha increasing by 0.1 in its full range. The execution time was capped at 5 hours.

<table>
<thead>
<tr>
<th>alpha</th>
<th>coh.</th>
<th>comm. cost</th>
<th>abs. score</th>
<th>simil.</th>
</tr>
</thead>
<tbody>
<tr>
<td>.0</td>
<td>.219</td>
<td>0.0</td>
<td>0.219</td>
<td>0.632</td>
</tr>
<tr>
<td>.1</td>
<td>.3154</td>
<td>0.0</td>
<td>0.3154</td>
<td>0.632</td>
</tr>
<tr>
<td>.2</td>
<td>.3154</td>
<td>0.0</td>
<td>0.3154</td>
<td>0.631</td>
</tr>
<tr>
<td>.3</td>
<td>.3154</td>
<td>0.0</td>
<td>0.3154</td>
<td>0.631</td>
</tr>
<tr>
<td>.4</td>
<td>.3154</td>
<td>0.0</td>
<td>0.3154</td>
<td>0.631</td>
</tr>
<tr>
<td>.5</td>
<td>.334</td>
<td>0.0179</td>
<td>0.3164</td>
<td>0.66</td>
</tr>
<tr>
<td>.6</td>
<td>.334</td>
<td>0.0198</td>
<td>0.3142</td>
<td>0.673</td>
</tr>
<tr>
<td>.7</td>
<td>.384</td>
<td>0.072</td>
<td>0.312</td>
<td>0.753</td>
</tr>
<tr>
<td>.8</td>
<td>.3907</td>
<td>0.118</td>
<td>0.279</td>
<td>0.824</td>
</tr>
<tr>
<td>.9</td>
<td>.3935</td>
<td>0.12</td>
<td>0.2823</td>
<td>0.832</td>
</tr>
<tr>
<td>1</td>
<td>.4047</td>
<td>0.5803</td>
<td>-0.1756</td>
<td>0.845</td>
</tr>
</tbody>
</table>

Table 3: Cromlech’s solutions on Tutored.

Cohesion and communication costs effectively grow according to alpha. The range of better solutions is in the middle, meaning that Cromlech penalizes extreme solutions. For low alpha, Cromlech favored the same low communication cost solution. All solutions except alpha=1 outperform the manual decomposition in terms of communication costs, and solutions with alpha>=0.7 outperform it in cohesion. The most similar solutions to the manual decomposition are the most distributed ones, confirming that the manual decomposition prioritized cohesion over communication cost reduction. If we consider the alpha=0.9 solution, we notice 3 distinct business domains: "Content and activities", "Job" and "User". A smaller fourth service containing 4 subgraphs can have those components put in separate services, obtaining a new 7 service solution:

<table>
<thead>
<tr>
<th>comm. cost</th>
<th>coh.</th>
<th>abs. score</th>
<th>similarity</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.12</td>
<td>0.5038</td>
<td>0.3838</td>
<td>0.81</td>
</tr>
</tbody>
</table>

Table 4: Metrics of the manually improved Cromlech decomposition.
Executive summary

This slight modification does not affect communication costs but improves cohesion of almost the 30%. The last measurement we took on this architecture with Cromlech has a cap of 15 services, alpha=0.9, and 20 hours of execution time:

<table>
<thead>
<tr>
<th>comm. cost</th>
<th>coh.</th>
<th>abs. score</th>
<th>similarity</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.464</td>
<td>0.8131</td>
<td>0.3491</td>
<td>0.858</td>
</tr>
</tbody>
</table>

Table 5: Metrics of Cromlech’s decomposition (15 services).

With more services, cohesion improves a lot but communication costs increase: 15% more costly in communication wrt the manual decomposition (which is 4 services) but has more than double the cohesion. We tested ServiceCutter’s three main clustering algorithms (ChineseWhispers, Leung, Girvan-Newman) and obtained these results:

<table>
<thead>
<tr>
<th>algorithm</th>
<th>comm.</th>
<th>coh.</th>
<th>abs. score</th>
<th>simil.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Leung</td>
<td>0.3241</td>
<td>0.3701</td>
<td>0.0560</td>
<td>0.761</td>
</tr>
<tr>
<td>Girvan-Newman</td>
<td>0</td>
<td>0.1636</td>
<td>0.1636</td>
<td>0.485</td>
</tr>
<tr>
<td>Chinese Whispers</td>
<td>0.1714</td>
<td>0.3505</td>
<td>0.1791</td>
<td>0.828</td>
</tr>
</tbody>
</table>

Table 6: ServiceCutter’s metrics.

Girvan-Newman provided a quasi-monolithic solution, while Leung a more distributed solution than Chinese Whispers but much more costly overall. Chinese Whispers is the best solution but is essentially a two-service split which still does not manage to adequately place the operations involving the Education entity. We proceed to also compute the metrics of the 4 service solution by Pangaea:

<table>
<thead>
<tr>
<th>comm. cost</th>
<th>coh.</th>
<th>abs. score</th>
<th>simil.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.2323</td>
<td>0.306</td>
<td>0.0737</td>
<td>0.8183</td>
</tr>
</tbody>
</table>

Table 7: Metrics of Pangaea’s decomposition.

The decomposition is weak in both components. The workers at Tutored also pointed out that it is difficult to categorize the 4 services because the decomposition does not identify distinct business domain. We finally computed the quality of Cromlech’s 4 service alpha=0.9 solution with Pangaea’s metrics:

<table>
<thead>
<tr>
<th>algorithm</th>
<th>comm.</th>
<th>coupl.</th>
<th>repl.</th>
<th>score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pangaea(5)</td>
<td>140</td>
<td>731</td>
<td>47</td>
<td>918</td>
</tr>
<tr>
<td>Pangaea(4)</td>
<td>172</td>
<td>890</td>
<td>46</td>
<td>1048</td>
</tr>
<tr>
<td>Distributed</td>
<td>1430</td>
<td>0</td>
<td>45</td>
<td>1475</td>
</tr>
<tr>
<td>Cromlech(4)</td>
<td>12</td>
<td>1646</td>
<td>54</td>
<td>1712</td>
</tr>
<tr>
<td>SC Chinese Whispers</td>
<td>541</td>
<td>1210</td>
<td>45</td>
<td>1796</td>
</tr>
<tr>
<td>Manual</td>
<td>1553</td>
<td>1208</td>
<td>51</td>
<td>2812</td>
</tr>
<tr>
<td>SC Girvan-Newman(5)</td>
<td>0</td>
<td>2997</td>
<td>45</td>
<td>3042</td>
</tr>
<tr>
<td>SC Girvan-Newman(4)</td>
<td>0</td>
<td>3155</td>
<td>45</td>
<td>3200</td>
</tr>
<tr>
<td>Monolith</td>
<td>0</td>
<td>3818</td>
<td>45</td>
<td>3863</td>
</tr>
<tr>
<td>SC Leung</td>
<td>3627</td>
<td>672</td>
<td>45</td>
<td>4344</td>
</tr>
</tbody>
</table>

Table 8: Pangaea’s metrics with various algorithms (default parameters).

Cromlech scores well, behind Pangaea and the distributed solution. The fact that these metrics favour very distributed solutions is an inherent weakness. Cromlech is efficient at reducing cross-service communication even with these metrics. Here is a comparison of all the approaches tested:

Overall, the key takeaways for these study are:
- the 7 and 15 services solutions greatly outperform the others in cohesion with minimal impact on communication costs
- the manual solution is outperformed in communication costs by almost every other decomposition
- Cromlech is efficient at reaching low communication costs while keeping reasonable cohesion values

3.2. Trainticket

For this architecture we wanted to:
- compare Cromlech’s decomposition with the benchmark and ServiceCutter, with Cromlech’s metrics, both without and with some transactional operations
- measure how similar is Cromlech’s decomposition is to the benchmark
measure the effectiveness of the alpha parameter in modulating the cohesion/communication costs tradeoff.

Trainticket is a large architecture of 124 operations and 137 data entities:

<table>
<thead>
<tr>
<th>ops</th>
<th>entities</th>
<th>monolith coh.</th>
<th>worst comm. cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>124</td>
<td>137</td>
<td>0.0961</td>
<td>807</td>
</tr>
</tbody>
</table>

Table 9: Metrics of Trainticket’s architecture.

We used two versions of the architecture, one without transactional operations and one with 16 transactional operations. Since the architecture was conceived in a microservice format, it shows distinct business domains and 27 services, and the benchmark shows extremely high cohesion:

<table>
<thead>
<tr>
<th>comm. coh.</th>
<th>services</th>
<th>abs. score</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.5328</td>
<td>0.8822</td>
<td>0.3494</td>
</tr>
</tbody>
</table>

Table 10: Trainticket’s benchmark metrics.

Communication costs are also high due to the separation of many operations accessing entities Route and Trip. We measured how closely we could emulate this benchmark by testing the version without transactional operations for a week, with a 27 service cap. The results are:

<table>
<thead>
<tr>
<th>comm. coh.</th>
<th>abs. score</th>
<th>simil.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.20695</td>
<td>0.8923</td>
<td>0.68535</td>
</tr>
</tbody>
</table>

Table 11: Metrics of Cromlech’s decomposition of the Trainticket architecture

We obtain a solution which is visually similar, but reduces communication costs by a large margin by creating a service with relevant routing and order operations, and exceeds the cohesion by a small quantity. Our solution presents 9 identical services and 5 almost-identical services with the benchmark. In comparison, ServiceCutter was only able to reach reasonable solutions with the Chinese Whispers algorithm and could not exceed 21 services. Their metrics are:

<table>
<thead>
<tr>
<th>comm. coh.</th>
<th>abs. score</th>
<th>simil.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.3135</td>
<td>0.8436</td>
<td>0.5304</td>
</tr>
</tbody>
</table>

Table 12: Metrics of ServiceCutter’s decomposition of the Trainticket architecture.

Considering their solution has 6 less services, their cohesion is good but communication costs are lackluster. We tested the version with 16 transactional operations, which took 20 hours to produce a quality solution compared to the version without transactional operations due to the constraint of placing many operations together by default.

<table>
<thead>
<tr>
<th>alpha</th>
<th>coh.</th>
<th>comm.</th>
<th>services</th>
<th>abs. score</th>
</tr>
</thead>
<tbody>
<tr>
<td>.0</td>
<td>0.1367</td>
<td>0.0</td>
<td>2</td>
<td>0.1367</td>
</tr>
<tr>
<td>.1</td>
<td>0.389</td>
<td>0.004</td>
<td>8</td>
<td>0.3886</td>
</tr>
<tr>
<td>.2</td>
<td>0.483</td>
<td>0.021</td>
<td>14</td>
<td>0.462</td>
</tr>
<tr>
<td>.3</td>
<td>0.569</td>
<td>0.04</td>
<td>22</td>
<td>0.565</td>
</tr>
<tr>
<td>.4</td>
<td>0.7054</td>
<td>0.1066</td>
<td>27</td>
<td>0.599</td>
</tr>
<tr>
<td>.5</td>
<td>0.8216</td>
<td>0.2</td>
<td>27</td>
<td>0.8216</td>
</tr>
<tr>
<td>.6</td>
<td>0.8575</td>
<td>0.2473</td>
<td>27</td>
<td>0.6192</td>
</tr>
<tr>
<td>.7</td>
<td>0.864</td>
<td>0.2579</td>
<td>27</td>
<td>0.6061</td>
</tr>
<tr>
<td>.8</td>
<td>0.9041</td>
<td>0.3906</td>
<td>27</td>
<td>0.5135</td>
</tr>
<tr>
<td>.9</td>
<td>0.9136</td>
<td>0.4494</td>
<td>27</td>
<td>0.4642</td>
</tr>
<tr>
<td>1</td>
<td>0.9305</td>
<td>0.728</td>
<td>27</td>
<td>0.2025</td>
</tr>
</tbody>
</table>

Table 13: Metrics of solutions on Tutored’s architecture. (with 16 transactional ops)

Again we can notice how alpha parameter scales with cohesion and communication costs, and how the best range for solutions is in the middle, from around alpha=.5 to about alpha=.9. Also, Cromlech is able to use a varying number of services based on alpha.

Cromlech outperform the benchmark in all alpha points except the extremes and ServiceCutter in the range between alpha 0.3 and 0.7 (included). Overall, this study corroborated the results found in Tutored’s analysis:

- Cromlech outperforms both benchmark and ServiceCutter for its ability to reduce communication costs
- it is also able to effectively modulate the cohesion/communication tradeoff with alpha
4. Conclusions

We feel confident in Cromlech’s ability to produce valuable results in architectures of varying sizes and domain number. Tutored and Trainticket are different in size and "context interconnectivity": while Tutored presents a more homogeneous architecture where contexts are not as easy to delineate, Trainticket presents a native microservice architecture with much clearer contexts. In both scenarios Cromlech performed reasonably well.

The main motivation for this work is helping human workers in finding a decomposition without low cross-service communication, while retaining some of the human-like skill to easily identify business domains. The algorithm succeeded in both tasks in all the solutions presented.

Cromlech is best used to produce an initial solution to be perfected by human experts. For example, the manually improved solution of the 4 service decomposition of Tutored is a slight modification of the algorithmic solution which greatly outperforms it.

Nonetheless, there are some limitations in our approach:

- it is time consuming: the time taken by the solver increases depending on the service cap;
- there is a manual process which still relies on the users’ accuracy in determining operation frequency and transactionality;
- the solver which we chose, Gurobi, is well performing and considered to be state of the art, but is a proprietary software requiring license.

For future works, hybrid approaches combining aspects of both static and dynamic decomposition could be useful: Cromlech could be expanded to derive operation frequency from execution traces and also use them to capture the time dependencies between operations.

The required information could be extracted from widespread specification formats like OpenAPI (like Baresi et al. [1]) which are available for some REST architectures, to avoid long manual processes.

References


Cromlech: a tool for semi-automatic monolith decomposition into microservices

Tesi di Laurea Magistrale in Ingegneria Informatica

Author: Davide Cocco

Student ID: 10540587
Advisor: Prof. Alessandro Margara
Co-advisors: Prof. Gianpaolo Cugola
Academic Year: 2020-2021
Abstract

Microservices are quickly becoming the industry standard as the paradigm chosen for the development and deployment of vast service-oriented web applications. Providing a suitable decomposition of a monolithic architecture to obtain microservices can prove to be a particularly hard practice, even for experts of the system, as it requires balancing a suitable level of decoupling with a satisfactory number of services, with the need of keeping a consistent state for critical data and coherently functioning distributed transactions, which requires communication between the services that naturally increases the more distributed the architecture becomes. Several patterns and best practices have been developed to mitigate the inherent weaknesses associated to distributed systems, but having a high quality decomposition immediately helps in reducing the latency associated to cross-service communication.

For this purpose we designed a tool called *Cromlech* which starting from an high level description of the endpoints of the service oriented architecture and the data they access, can feed such data to a Linear Programming algorithm to produce a decomposition that balances decoupling between the services (and high internal cohesion) and low cross-service communication. The description of the system is provided through a YAML file, and our algorithm allows to choose a maximum number of services for the final decomposition and a weighted objective function for the LP model allowing the user to modulate the tradeoff between internal cohesion/decoupling and cross-service communication.

The testing was performed on two architectures: a commercial one of medium size and a theoretical benchmarking one of large size. We compared our decomposition with the ones obtained through two other decomposition tools, and the results show that *Cromlech* is a competitive system that could help experts of the system in providing a valuable and functional decomposition of their architectures.

**Keywords:** microservices, distributed systems, linear programming
Abstract in lingua italiana

Le architetture a microservizi si stanno velocemente affermando come lo standard per la realizzazione di ampie architetture orientate ai servizi. Ottenere una decomposizione valida di un’architettura monolitica può rivelarsi una pratica particolarmente ardua, persino per esperti dell’architettura, poiché richiede di bilanciare il disaccoppiamento fra un numero ragionevole di servizi con la necessità di mantenere uno stato consistente per i dati, che ha a sua volta bisogno di comunicazione tra servizi la quale aumenta naturalmente a seconda di quanto l’architettura è distribuita. Diversi pattern sono stati sviluppati per mitigare le debolezze associate ai sistemi distribuiti, ma avere una decomposizione di buona qualità aiuta da subito nella riduzione della latenza associata alla comunicazione tra servizi.

Per questo scopo abbiamo realizzato uno strumento chiamato Cromlech, il quale partendo da una descrizione ad alto livello delle operazioni dell’architettura orientata a servizi, fornisce questi dati ad un algoritmo di Programmazione Lineare, che restituisce una decomposizione che bilancia disaccoppiamento tra i servizi (e un’alta coesione al loro interno) e ridotta comunicazione tra i servizi. La descrizione del sistema è fornita con un file YAML, e il nostro algoritmo permette di scegliere un numero massimo di servizi, e una funzione obiettivo pesata permette di modular il tradeoff tra disaccoppiamento e comunicazione tra servizi.

Abbiamo testato l’algoritmo su due architetture: una commerciale di dimensione media e un benchmark teorico di dimensione ampia. Le decomposizioni ottenute sono state comparative con quelle fornite da altri due strumenti, e i risultati mostrano che Cromlech è un algoritmo competitivo che può aiutare gli esperti del sistema nella realizzazione di una decomposizione funzionale e utile delle loro architetture.

Parole chiave: microservizi, programmazione lineare, sistemi distribuiti
3 Cromlech

3.1 Cromlech’s workflow ................................................. 31
  3.1.1 Design of the input file .................................... 32
3.2 Architecture extraction ........................................... 33
  3.2.1 Processing of the input file ................................ 33
  3.2.2 Creation of the solver file ................................. 34
3.3 Optimization problem solver ..................................... 34
3.4 Output files ......................................................... 34

4 Experiments and analysis ........................................... 37

  4.1 Decomposition similarity algorithm .......................... 37
    4.1.1 Decomposition similarity algorithm example .......... 38
    4.1.2 Similarity in terms of data ............................ 39
  4.2 Tutored .......................................................... 40
    4.2.1 Architecture characteristics ........................... 40
    4.2.2 Manual decomposition ................................... 41
    4.2.3 Cromlech’s solution ...................................... 45
    4.2.4 Cromlech’s 15 service solution ........................ 52
    4.2.5 Pangaea decomposition ................................. 55
    4.2.6 Evaluation of Cromlech with Pangaea’s metrics .... 59
    4.2.7 ServiceCutter decomposition ............................ 61
    4.2.8 Recap of Tutored’s decompositions ................... 65
  4.3 Trainticket ....................................................... 68
    4.3.1 Obtaining the YAML input file ......................... 68
    4.3.2 Metrics of Trainticket’s architecture, no transactional operations .. 69
    4.3.3 Metrics of Trainticket’s architecture, with transactional operations . 70
    4.3.4 Trainticket’s benchmark decomposition ............... 71
    4.3.5 Cromlech’s solution, without transactional operations ... 74
    4.3.6 ServiceCutter solution .................................. 78
    4.3.7 Trainticket’s solution, with transactional operations .... 82

5 Conclusion and future works ...................................... 87

Bibliography .......................................................... 89
Introduction

Since the birth of the term around the second half of the first decade of the 21st century, microservice architectures have become increasingly relevant and widespread to account for the growing size of both service oriented architectures and the needs of their users. Many important companies with a sizeable Web presence, such as Amazon, Netflix and eBay [1], have already started implementing microservices to solve the problems posed by their dated and encumbering monolithic architectures, incentivizing other tech companies to evolve in the same direction to avoid losing the competitive edge.

0.1. Monolithic and microservice architectures

A monolithic architecture is traditionally the simplest way to deploy an application, which works by deploying all the functionalities together, as a single package. It is the starting point of almost any service-oriented architecture, and has its major strength in its simplicity: it is easy to understand, deploy, and eventually scale by simply deploying more copies. When the application starts growing, however, this paradigm shows various defects which make it more and more problematic as the size of the monolith increases: it becomes more expensive to deploy and scale, and might even be intimidating to new developers introduced to it due to its size, with the result of less productivity. In the earlier times of the Web, this problem could maybe be ignored or underestimated, but nowadays, with exponentially growing connection bandwidth and web users (and thus customers for the applications) it became necessary to adopt a paradigm shift, at least for the biggest applications.

The microservice paradigm for service-oriented architectures promptly became the solution to the problem. A definition of Martin Fowler [2], defines this architectural style as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies.
Microservices make continuous deployment easier and decompose the monolith in such a way that different teams can work at different services with easy and without loss of productivity, also allowing to obtain greater scalability and possibilities for load balancing.

0.2. Benefits and drawbacks of microservices

The most straightforward advantage of a microservice architecture is its deployment speed: the services can be deployed at different times and their size is just a fraction of the complete codebase, allowing for a much shortened build and deployment time. Although this benefit may seem insignificant for smaller architectures, it becomes absolutely relevant with larger architectures.

Ease of deployment is a similar advantage, as the discrepancy between development environment and production environment represented by dependency on OS versions, library versions, machine constraints etc. is difficult to pinpoint in a larger monolithic architecture, but easy to keep track of in a microservice architecture which is built incrementally. This is also due to the fact that microservices can be easily containerized, virtualized, and orchestrated with technologies such as Kubernetes, positively impacting dependency conflicts.

A much better maintainability and fault tolerance is another benefit, as troubleshooting is easier when considering loosely coupled and smaller services rather than the whole codebase, due to issues being isolated and investigated more promptly. Fault tolerance is better due to the increased ease and speed of deployment of redundant services. Microservice also enjoy better testability than their monolithic counterpart, and they are easier to enhance and modify since working on a single service does not influence the availability of others.

Scalability is another great benefit, as we may need to scale particular functionalities and thus only scale up those particular services, which is something that cannot be done on a monolithic architecture as it needs to be scaled as a whole.

Microservices allow for more technological heterogeneity as each service can be developed with a specific technology stack without worrying about dependencies with other services, and due to the fact that different teams can work on different services as small specialized units, increasing productivity.

On the other hand, microservice architectures are clearly more complex in nature, as they require micromanaging multiple code units, orchestrating and load balancing them. The organization needs to implement the communication system to make them interact and being able to deal with partial failure. In general, most of the advantages listed so
far are a double-edged blade, introducing more complexity and expertise to be handled. Due to this added complexity, microservice architectures are more expensive and require and organizational culture change, as a company which operated for a long time on a monolithic architecture might find challenging changing the paradigm upon which its system is based.

One of the main challenges in designing a functional and efficient microservice decomposition is achieving a reasonable level of data consistency, as we will detail in the next chapters.

0.3. History of microservices

Although the term microservice is of recent creation, the technological steps which exploited the advantages of smaller services encapsulating a specific business logic is much older. An example of this are Enterprise Java Beans, released in 1997 by IBM, which standardized the development of small and re-usable web-related components. Even though some characteristics of modern microservices were present, they only worked with Java and were still very far from achieving the full benefits of contemporary microservices.

The next step was the ideation of Service oriented architectures, which defined the "service" as a self-contained program able to perform a specific task, loosely coupled and independent from other services. Initially, the SOAP standard, released in 1999 by Microsoft was the first popular implementation of service oriented architectures, but it still suffered from poor scalability and capabilities of error handling.

SOAP’s evolution, the REST paradigm, became increasingly popular after 2008 and has become the standard for the realization of services. The high re-usability, decoupling, scalability, and ease of use of REST services became the fertile ground for the evolution of microservices. The term "microservice" was first used in 2011 by engineers participating in a workshop near Venice, to describe an architectural style that some of them were developing to build continuously deployed systems, while incorporating the DevOps philosophy. This form of architecture quickly gained popularity and the term became widespread and "official" in 2012. Today, the sector is growing at an extremely fast rate, as The Cloud Microservices Market Research Report of February of 2020 has predicted the size of the global microservice architecture market will increase with a compound annual growth rate of 21.37 percent from 2019 to 2026 with the market reaching a value to $3.1 billion by 2026 [3].
1 | Background: decomposition of a monolithic architecture

The decomposition of a monolithic architecture is a long and important process that most organizations have to undertake once their web architecture reaches a critical size. In this chapter we discuss the common problems organizations face when deciding how to split their monolithic architectures and the state of the art.

1.1. Challenges of splitting a monolithic application

We define a system to be partitioned whenever it is split in multiple subsystems with a limited degree of independence, meaning that some sort of communication is necessary between the partitions for the system to provide its functionalities. A partitioned system is inherently more complex to realize and handle than a centralized one, and as summarized by the well known first rule of distributed systems:

"Don't distribute your system"

— Martin Fowler [4]

Even a monolithic architecture should be kept in this form unless it has become too complex and unmanageable. There are several properties an organization must be mindful of to design a functional partitioned system which outperforms its monolithic counterpart or is at least not detrimental: its availability (although not in the traditional sense of software engineering), latency, partition tolerance and consistency. This clearly applies to microservice architectures, which are indeed partitioned systems.

Bailis et al. [5] define high availability as the capability of a partitioned system to guarantee its operations to be "always online", and, as a side-effect, to have low latency. It has been well documented since the early 2000s that some properties which can be guaranteed in a centralized application cannot be guaranteed in a high availability distributed application. Among these properties are:
Background: decomposition of a monolithic architecture

- **sequential consistency**, that can be defined in the context as the property where the result of any subsystem is the same as if the operations of all the subsystems were executed in some sequential order, and the operations of each individual subsystem appear in this sequence in the order specified by its program;

- **serializability**, which can be defined as the property where the outcome of a series of overlapping transactions is the same of the series of transactions being executed serially and without overlapping. In particular, serializable transactions are not available in a microservice application of high availability.

These properties require blocking coordination among the different subsystems and thus introduce latency, therefore a highly available distributed system cannot guarantee them: in other words, in order to obtain the advantages of a partitioned system with high availability one must necessarily abandon strong consistency semantics and partition the monolith in such a way that cross-service communication is as little as possible: this can either be done by creating less subsystems of bigger size at the cost of reducing the benefits of the microservice paradigm, or create such subsystems so that they are numerous and as cohesive as possible in the inside, and loosely coupled with the others.

This is the main tradeoff to be considered when decomposing a monolithic architecture. Our solution proposes to let the user modulate this tradeoff to obtain a customized solution through the use of weights which influence how distributed or monolithic the result will be.

An important element to be considered when realizing a microservice application are **distributed transactions**, as they pose a significant problem due to their requirements of **atomicity and isolation**, all without violating certain **integrity constraints**, and need cross-service communication whenever the pieces of data they access are placed in different services.

For instance, let’s imagine the microservice architecture of an *e-commerce* application, with two different microservices, one handling the business logic pertaining to orders and the other managing the customer data. Whenever performing an order, the *Order* microservice must interact with the *Customer* microservice to check if the user who’s performing the order has enough money in balance. It is absolutely vital for the correct functioning of the process to make sure that:

- The operation either fails or succeeds completely: the balance is reduced only if the order has fully succeeded, otherwise it stays the same (*Atomicity*).

- Concurrent execution of transactions leaves the data in the same state as if the
transactions would have been executed sequentially (*Isolation*).

- Certain constraints depending on the nature of the operation (such as the balance not becoming negative) must be enforced (*Consistency*).

These are the A, C and I of the well known **ACID properties**. We will refer to consistency as *integrity* from now on. We will refer to the combination of these three requirements as **transactionality**.

To make sure these properties are respected, distributed transaction systems often employ an algorithm called **two-phase commit (2PC)**, which implements a consensus system between the participating nodes so that the order of updates is safe and there is a global commit or abort to guarantee atomicity. The problem with 2PC is that it is a blocking protocol, and when the involved nodes are numerous, the latency and communication overhead becomes prohibitive. Moreover, 2PC requires the data entities distributed in the nodes of the system to belong to the same database, but this poses some disadvantages in the context of microservice architectures.

**Database per service and safe transactions**

Consequently, in our solution, we assumed the user desires a pattern called **Database per service** [6], where each microservice owns a standalone database. This has pros:

- Helps ensure that the services are loosely coupled. Changes to one service’s database do not impact any other services.

- Each service can use the type of database that is best suited to its needs.

But also some cons:

- Implementing distributed transactions can in fact be tricky as it needs to be done in a customized manner at the application layer, as we cannot use algorithms such as 2PC.

- Also distributed joins (joins between data entities that are located in different nodes) can become hard to implement.

With this pattern we either have to avoid cross-service distributed transactions altogether, which can be done by ensuring that the involved operations possess all the data they access in the distributed transaction in their same microservice (and thus there’s no need for cross-node communications). At the same time, the distributed transactions must be placed in the same service as the leader replicas of the data entities they access (which
Background: decomposition of a monolithic architecture

are the only ones written, fulfilling the single writer principle).

In the previous example, this would be done by grouping the Customer and Order operations in the same microservice along with all their accessed data entities, so that they work on the same data server. This comes at the expenses of traditional availability, and thus also high availability, as fault tolerance is reduced, and in certain scenarios even latency, as some distributed transaction algorithms might introduce a longer response time especially in presence of multiple participants.

The other technique allows the separation of the operations belonging to the distributed transaction in different services, and it is called Saga pattern [7], where each business transaction that spans multiple services is a sequence of local transactions. Each local transaction updates the database and publishes a message or event to trigger the next local transaction in the saga. If a local transaction fails because it violates a business rule then the saga executes a series of compensating transactions that undo the changes that were made by the preceding local transactions. This obviously requires the engineer to develop these compensating transactions, by no means a trivial task in certain scenarios.

Our model will treat atomicity, integrity and isolation as a property of the operations rather than the data to better represent the case of distributed transactions, and will be treated as the guarantee of the atomicity and isolation properties in case of write operations.

We will adopt an hybrid approach where the user can specify which operations are constrained to be in the same microservices as the leader replicas (the leaders) of the data entities they access in order to ensure those operations are guaranteed transactionality.

1.1.1. Bounded contexts and Domain Driven Development

Cross-service communication is reduced when services hold operations which are related to each other, and as different as possible from operations in other services. This happens because in such a scenario it is less likely that an operation will access data outside its service, as it is more probable that all or most of the data such operation accesses is already grouped in its same service, along with other similar operations.

A methodology focused on this characteristic, called Domain Driven Development, argues that the characteristics of the code should match its business domain. In other words, the architecture should be designed and should evolve in function of the different business domains it is composed of, in order to encourage modularity and cross-team communication between workers of different expertise, other than the advantages listed
Background: decomposition of a monolithic architecture

in the previous paragraph. As such, the architecture can be seen as a set of different bounded contexts that are independent from each other and that possess unified models. Domain Driven Design proposes various ways to model the relationships of different bounded contexts, and this can be mapped in a practical sense onto the workings of a microservice architectures.

In conclusion, treating operations as the smallest possible unit of the architecture, and finding a way to aggregate them on the basis of the similarity of their business domain, can be seen as an instrumental method to ensure high availability.

1.1.2. Replication

Another characteristic of the system which influences the latency of the system is the number of replicated data entities, and how their updates are managed. More replicated caches means more communication overhead between services to keep them up to date. On the other hand, more replicas also increase availability and reduce latency, as services become able to access their local replica instead of a remote data store.

With the term "replica", from now on, we will mean the local cache in a microservice of some data entities its operation/s accesses. Each data entity will be assigned a "leader replica", with criteria which will be explained in the next chapter.

Whenever an data entity needs to be written, we need to write the leader replica of the same data entity in an asynchronized fashion to achieve eventual consistency. This requires a communication overhead whenever such write operation happens in a microservice other than the one containing the leader replica.

1.2. State of the art

"Defining an application’s microservice architecture is more an art than a science."

— Chris Richardson, Microservices Patterns [8]

The decomposition of the monolithic architecture is usually done manually by engineers of the organization, but this is an extremely long process which progresses mostly by trial and error. Many researchers have proposed several automatic or semi-automatic alternatives to this process in order to reduce costs and time required to perform this task.

The approaches can be split in two different kinds: static decomposition whenever it
Background: decomposition of a monolithic architecture

is performed according to some description of the architecture and not during runtime, and dynamic decomposition whenever it is performed during runtime according to information such as online execution traces of the services. Dynamic decomposition tends to work better in DevOps environments as it exploits the capabilities for scaling and load balancing due to the fact that the microservices can be arranged according to dynamic data gathered at runtime, while static decomposition is generally better at providing a first decomposition of the monolith, which can be integrated subsequently with execution traces or similar information to provide additional advantages.

1.2.1. Static decomposition

Levcovitz et al. [9] proposed a manual method inspired on the three layer structure of most web applications, composed of presentation, application, and data layer. The presentation layer is represented by a set $F$ of facades $f_i$, which are the endpoints of the application, while the application layer is represented by the operation performing the business logic, by the set $B$ of business functions $b_i$. The data layer is represented by the database tables $t_{b_i}$ of the set $D$.

Then, six steps are manually executed to obtain a decomposition:

1. Database tables are grouped into different subsystems $ss_i$ which represent different business areas $a_i$;
2. Create a graph between the elements of the $F,B,D$ sets where the edges represent call relationships from facades to business functions, and access relationships from business functions to database tables;
3. Identify the pairs $(b_i, t_{b_i})$ connected in the graph;
4. For each subsystem, list the related $(b_i, t_{b_i})$ pairs;
5. Potential microservices are found by inspecting each path from facades to database tables and checking the code of the facades and business functions on such path. The inspection’s purpose is to find a meaning to the interaction, its features, input and output, and every bit of information that could classify it inside a candidate service or another;
6. Complete the microservice migration by implementing the API gateways which allow them to communicate.

This method was then used for the decomposition of a monolithic bank system composed of 750,000 lines of code.
Background: decomposition of a monolithic architecture

This process is intuitive and straightforward and provides a baseline upon which the architecture can be built systematically and eventually easily expanded with the growth of the system. The disadvantages are that it is still manual and time consuming, and does not provide any metric to evaluate the decomposition quality, and does not take into account any measurement regarding the communication costs.

Another work by Laigner et al. [10] expands this approach by assuming a REST based architecture and introducing different elements of a web application: a set I of REST interfaces, a set B of business functions, a set R of repository functions, and a set D of database tables:

1. Just like the previous approach, a dependency graph is built, with paths starting from interfaces, going through business functions and repository functions, and ending in the database tables;

2. The frequency of invocation of each interface is estimated and used as a weight for the graph edges;

3. Coupling is defined by the relationships between database tables, by using foreign keys to determine if tables are associated;

4. The interface frequency, coupling between tables, and a workload limit given by the user, an optimization problem is formulated and solved to find the optimal allocation of the resources.

This method expands the work provided by Levcovitz et al. by introducing a numerical metric of communication costs through the access frequency of interfaces, and also introduces an automatic component represented by the optimization problem. This approach is actually quite similar to the one we will introduce.

A static approach which makes use of logs files produced at runtime is the one proposed by Taibi and Systa [11]. The log files are analysed through a process-mining tools which allows to extract the processes and represent them as the business functions and database tables involved in them, under the form of a graph. These call frequency is also extracted from the log. From this graph, circular dependencies are removed, and the output is a refined graph which can then be used by experts to provide a manual decomposition. Moreover, a metric for the evaluation of the decomposition is proposed, represented by coupling (measured using the number of calls to external services from each service), the number of classes per service, and the number of classes in need of replication. This method is particularly interesting because it uses process mining as a support for the
final decisional process which is left to the organization’s experts, while adding evaluation metrics as an aid in finding the best solution. Process mining also allows to identify dead code and investigate the usage of each component. However, the main issue with this approach is the necessity of a great quantity of log traces to be parsed in an ad hoc way, which is a strong assumption.

Another interesting take on the subject was provided by Baresi et al. [12]: starting from the OpenAPI specification of the web application, an automatic decomposition can be provided by gathering the terms and matching them with a reference vocabulary from Schema.org. Then, using a fitness function based on DISCO, a precomputed database of similar words which measures similarity based on their co-occurrence, we can identify domains and reference concepts that the terms share and provide the candidate microservices. This way, different reference concepts are produced and associated each to a different microservice.

This approach focuses on semantic similarity of the resources, which is something akin to what a human would intuitively do when providing a decomposition. Furthermore, the granularity of microservices, the coupling and cohesion relationship are taken into account in allowing the user to tune some parameters to provide some customization.

This method is mostly automatic and often immediately executable as OpenAPI specifications are widespread, but can sometimes (20% of the times, according to the authors), result in a decomposition with few coarse-grained services as the reference concepts found are few and shared.

A particularly popular tool that has been taken as inspiration by many related works, especially for what concerns its thorough representation and formalization of its coupling criteria, is Service Cutter by Gysel et al. [13]. Service Cutter represents the elements of the application with three kinds of nanoentities: data, operations (that do not necessarily handle data, differently from our model), artifacts (snapshots of data and operation results). The architecture description is provided through a JSON file (called System Specification Artifact) specifying all the use cases, which detail operations and the nanoentities they read or write, similarly to our model, and various other fields that let the user specify user representations (information like pre-existing services, security zones, access groups, some peculiarities about the coupling criteria, etc.).

Service Cutter also adopts 16 different requirements for which two nanoentities might or
Background: decomposition of a monolithic architecture

might not be in the same service, called coupling criteria, grouped into four main areas:

- Cohesiveness, listing common properties that qualify why two related nanoentities should be put together;
- Compatibility, specified by the user in the user representation area, explaining diverging characteristics of nanoentities and which should not be grouped together;
- Constraints, unavoidable requirements imposing the presence of two nanoentities in the same service, or their distribution in different services;
- Communication, detailing communication costs like synchronization and cost of remoting for groups of nanoentities.

After parsing the JSON, Service Cutter builds a weighted undirected graph to be fed to a clustering algorithm, either Girvan-Newman [14] or Chinese whispers [15] depending on the user’s choice, to produce the final microservice candidates. Service Cutter is particularly groundbreaking for the great detail used on the definition of coupling requirements and has been applied to multiple real problems with success, but suffers from the weaknesses inherent to the clustering algorithms and still requires a rigorous specification file.

1.2.2. Dynamic decomposition

Dynamic decomposition has found more space in actual industrial research than academic research due to the need of having one, or better multiple, running web applications for the online data gathering.

A fairly recent and successful tool is Mono2Micro [16], developed by a team at IBM, that dynamically collects runtime traces of specific use cases and applies a form of hierarchical clustering to partition the application classes. In particular, it considers two dimensions:

- the space dimension as the partition of the application classes that results in the service candidates, seen as a set of well-defined functionalities with high cohesion and low coupling, such that they could be easily explained to a business user. To build those, module dependencies and direct/indirect call between use cases are taken into account. Business logic is emphasized to avoid grouping dependent use cases belonging to different functionalities.
- the time dimension is extracted from runtime call traces to produce temporal relations and co-occurrences between classes. This allows to extend already existing
Background: decomposition of a monolithic architecture

methods that only consider direct calls by also including indirect calls using pattern recognition. Furthermore, it is used to derive both short term and long term temporal call relationships between classes to understand a functional decomposition in terms of communication costs.

The evaluation of the solution is performed through five metrics:

- Inter-call percentage, as a measure of cross-service communication;
- Business context purity, as a measure of loose coupling and high cohesion;
- Structural modularity, as a measure of resilience to modifications;
- Interface number, as a measure of the implementation cost of the decomposition;
- Non-extreme distribution, as a measure of how evenly sized the microservices are.

Mono2Micro outperforms four different algorithms it was compared to and has been evaluated as effective in a survey workers of different specializations who sued it. It is also highly efficient in identifying dead code. The weakness is that being still fairly recent, it has not yet seen much production usage, and it does not take into account database transactions and database interaction patterns, which have been deemed as "extremely important" by almost two thirds of the survey participants. Another general issue is the need to generate the execution traces of the use cases manually as many organizations do not possess sufficient automated tests.

1.3. Motivations

From our review of the state of the art we found several issues in common with most works:

- most static decompositions require detailed manual work or are complex enough to need a deep understanding of the algorithm, or are entirely based on a manual approach;
- most techniques do not take into account the issue of distributed transactions;
- some techniques require input which is not easily accessible (such as the execution traces);
- some techniques lack metrics to quantify the solution’s validity;
- data replication is assumed as full unregulated replication or not considered as a possibility;
• most approaches lack the customization needed for the user to modulate how distributed/monolithic the solution will be.

We aim to realize an algorithm which requires an easy to produce architecture description, which tries to emulate the real functioning and needs of a microservice architecture. As such, we need a system that is able to understand and weigh the issues related to data consistency, and make a rational use of replication.

Therefore, our solution will try to accomplish these goals:

• model distributed transactions and offer metrics to evaluate cross-service communication;

• model replication as a tool that is used only when mathematically useful;

• require accessible and easy to produce input;

• require minimal manual work;

• allow the user to decide how many services and how distributed/monolithic the solution will be.
Our solution *Cromlech* considers a model which represents a software application as a system of interacting operations and data entities. The design of the modeling framework balances two requirements:

1. expressivity, to capture organization, communication, and data management concerns
2. simplicity, to limit the effort for developers to build the model.

### 2.0.1. 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 be used to model a single table: Cromlech will treat the table as an unbreakable unit and map it to microservices accordingly. Depending on their view of the system, developers may also decide to model multiple related tables as a single data entity or to split a table into multiple data entities. The user can also decide to represent the single columns as data entities, to reach a finer level of granularity. As *Cromlech* represents the requirements of the system as a property of operations rather than data, data entities are characterized simply by a string label to identify them.

### 2.0.2. Operations

Operations represent units of execution of the application, which are candidate to become logic functionalities of microservices. Each operation accesses (reads and/or writes) data entities and is associated to a single microservice. As such, operations can be seen as any action present in the software that can access the data entities.

In Cromlech, 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 a dependency relation between operations and data entities, and 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 due to the remote data access the operation needs to perform.

- **frequency**: an integer that indicates how frequently \( o \) is invoked. Operations which are invoked more frequently have a larger impact on the system’s performance, thus in the decomposition process *Cromlech* will prioritize reducing the costs associated to operations that are invoked more frequently, for instance the communication costs for accessing remote data entities discussed above.

- **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 operations will be always located on the same microservice where all the data entities it accesses are deployed.

- **colocated operations**: list of names of other operations that need to be located on the same microservice as \( o \). Using this data entity, developers can indicate operations that belong to the same business unit and that they want to deploy together on the same microservice. Accordingly, colocated operations help developers to express organizational constraints that *Cromlech* cannot break. At the same time, the presence of colocated operations reduces the number of acceptable solutions and may reduce the computation time needed by the solver.
2.1. Optimization model

Cromlech combines the system model and input parameters provided by developers from a configuration file to formulate an optimization problem, which aims to balance two conflicting forces:

1. The **cohesion metric** of the architecture, which represents the cohesion within individual services and the decoupling between different services. It pushes towards decentralized solutions, where each service is only responsible for a limited and highly-cohesive set of data entities and operations.

2. The **communication costs** that derive from inter-service communication and push towards solutions that centralize data entities and operations.

The overall problem defines an objective function that sums the two components above. It weights them by the *cohesion weight*, that developers can set to define the relative importance of each component for the specific architecture at hand. In the end, the goal of the solver is to find an optimal decomposition, that is, an allocation of operations and entities onto a set of microservices, that maximizes cohesion without incurring in excessive communication costs.

*Cromlech* allows for data entities to be replicated at different microservices, a common practice in real microservice architectures. It also assumes a **single writer principle**, meaning that a single microservice is responsible for all updates (write accesses) to a given data entity $e \in E$. This implies that each entity must have one and only **leader replica**, located in such service.

The elements manipulated by the optimizer will be:

- the set of entities $E$
- the set of operations $O$
- the set of microservices $M$.

2.1.1. Input parameters

Cromlech takes in input a small number of parameters that guide the decomposition process based on the requirements of developers:

- **maximum number of microservices**: is the cardinality of $M$ and 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 de-
composition into fewer microservices.

- **cohesion-communication ratio**: a real number that indicates the importance developers data entity 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 0.5, which suggests to Cromlech that organization and communication concerns are equally important. Increasing the value would favor solutions where microservices are highly cohesive (potentially at the cost of increased 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 cohesion-communication ratio provides a single parameter to steer the decomposition towards design concerns or runtime concerns.

### 2.1.2. Decision variables

Decision variables are manipulated by the optimizer to provide solutions with different objective values and are the building blocks of constraints and objective functions. Moreover, variables alone are sufficient to encode a complete solution. *Cromlech* defines three variables:

- $x_{o \in O, m \in M} = 1$ if and only if operation $o$ is located in microservice $m$, 0 otherwise
- $y_{e \in E, m \in M} = 1$ if and only if entity $e$ is replicated in microservice $m$, 0 otherwise
- $l_{e \in E, m \in M} = 1$ if and only if entity $e$ has its leader replica located in microservice $m$, 0 otherwise

This three variables allow us to understand all the necessary characteristics of the decomposition: where the operations are located, where the entities are replicated, and where their leader replica is located to fulfill the single writer principle.

### 2.1.3. Objective function

The objective function describes the expression derived from the decision variables that the optimizer aims, in *Cromlech’s* case, to maximize, and as such it denotes the quality of the solution. As outlined in the previous sections, our objective function is a combination of the cohesion metric and the communication cost.
Cohesion metric

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:

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

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

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

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 it can be derived by looking at the set of data entities each operation needs to access. Accordingly, 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 in 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.

$$\forall o \in O \forall e \in E \leftrightarrow acc_{e,o} = 1$$

$$S(o1, o2) = \frac{|E_{o1} \cap E_{o2}|}{\min(|E_{o1}|, |E_{o2}|)}$$

The cohesion metric for the entire architecture is the sum of the cohesion metrics for each service $m \in M$, weighted by the number of operations in $m$ with respect to all operations.

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

The communication costs measure the overhead of inter-service communication. Our model enables data entities to be replicated at different microservices. Replication is widely used in microservices architecture to improve service availability and reduce latency, as operations may access locally replicated data without incurring the cost of remote data access.

Accordingly communication costs include two aspects: the costs of remote data access and the cost of replication.

Remote data access occurs when an operation \( o \in O \) needs to access a data entity \( e \in E \) but the two are placed on different microservices.

Replication costs involve keeping remote replicas up-to-date.

Recall that we assume a single writer principle, meaning that, for a given data entity \( e \in E \), a single service (the leader replica) is responsible for all updates (write accesses) to \( e \).

Formally, operations incur the following communication costs. \( R_{o,e} \) is 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 of invocation of \( o \).

For each data entity \( e \in E \) and for each operation \( o \in O \), let us denote \( accR_{e,o} \) and \( accRW_{e,o} \) boolean variables that hold 1 if and only if \( o \) accesses \( e \) in read-only or in read-write mode, respectively. We extract their values from the data access property of operations in the input model.

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

\( W_{o,e} \) is the cost that \( o \in O \) incurs for writing a data item \( e \in E \). Due to the aforementioned single writer principle, 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.

\[
W_{o,e} = \sum_{m \in M} accRW_{e,o} \cdot f_o \cdot x_{o,m} \cdot (1 - l_{e,m})
\]
Finally, an entity $e \in E$ incurs a replication cost for keeping replicas up to date, which is proportional to the number of (non leader) replicas of $e$ and to the frequency at which $e$ is updated.

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

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

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

**Computation of the objective function**

The goal of the optimization problem is to maximize the cohesion metric $Coh$ while avoiding the communication costs $Com$.

Recall that the cohesion metric is a real number between 0 and 1: for ease of use, we also normalize the communication costs to a scale between 0 and 1.

To do so, we consider the decomposition where the communication costs for the given software application are minimum and the one where the communication costs are maximum.

The minimum value of communication costs is 0, and this happens in a centralized monolithic architecture (with no remote data access and no replication).

The maximum is 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 (colocated operations and transactional operations accessing at least one entity in common must be located in the same microservice). Let us define the maximum communication costs as $Com_{max}$.

The objective function becomes:

$$Obj = \alpha \cdot Coh - (1 - \alpha) \cdot \frac{Com}{Com_{max}}$$
2.1.4. Constraints

Constraints are expressions which guide the optimizer in setting allowed values for decision variables to find a valid solution. Our formulation identifies six main constraints:

1. **Lone operation constraint**: each operation is deployed on exactly one microservice;
2. **Entity deployment constraint**: each entity is deployed on at least one microservice;
3. **Lone leader replica constraint**: each entity has exactly one leader replica;
4. **Leader replica is a replica constraint**: the leader replica is a replica;
5. **Colocated operations constraint**: colocated operations must be placed in the same microservice;
6. **Transactional operations constraint**: transactional operations are in the same microservice of the leader replica of entities they access.

**Lone operation constraint**

We want each operation to be deployed exactly once on a single microservice. To express this, we can simply say that the sum of all placements which are true for the operations is equal to 1:

\[
\forall o \in O \sum_{m \in M} x_{o,m} = 1
\]

**Entity deployment constraint**

Each entity needs at least one replica in the decomposition, and can potentially be replicated in each service. The formulation is similar to the previous constraint:

\[
\forall e \in E \sum_{m \in M} y_{e,m} \geq 1
\]
Lone leader replica constraint
Since our model abides by the *single writer principle*, we want a single leader replica for each entity. This is expressed in a manner which is akin to the lone operation constraint:

\[
\forall e \in E \sum_{m \in M} l_{e,m} = 1
\]

Leader replica is a replica constraint
We must enforce that the placement of a leader replica in a microservice implies that a replica in the general term is in the same service, as the leader replica itself is a replica. The opposite might not necessarily be true, as a replica might not be the leader. This can be expressed as:

\[
\forall e \in E, m \in M y_{e,m} \geq l_{e,m}
\]

Colocated operations constraint
Deployment needs to enforce the constraints expressed by the developers with respect to the colocation of multiple operations. These constraints serve to indicate that two operations belong to the same business domain and that the solver must not separate them on different microservices. For any pair of operations \(o_1 \in O\) and \(o_2 \in O\), let us define a boolean variable \(\text{coloc}_{o_1,o_2}\) that is 1 if and only if developers requested the colocation of \(o_1\) and \(o_2\) (colocated operations data entity of operations in the input model):

\[
\forall o_1 \in O, o_2 \in O \sum_{m \in M} x_{o_1,m} \cdot x_{o_2,m} \geq \text{coloc}_{o_1,o_2}
\]

Transactional operations constraint
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, 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 $o \in O$, let us define a binary variable $tr_o$ that 1 if and only if developers requested $o$ to be executed with transactional semantics (transactional data entity of operations in the input model).

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

2.1.5. Linearizing the problem

We chose a linear optimizer for Cromlech, called Gurobi[17]. In a linear optimization problem, all expressions of constraints and of the objective function need to be linear. Our constraint formulation is not linear due to: Notice that the above problem is not linear for two reasons.

- Some formulas include a multiplication of binary variables. For instance computing colocated operations involve multiplying $x$ by $x$, and computing the communication costs involve multiplying $x$, $y$, and $l$.

- Some formulas include a multiplication of a binary variable and a continuous variable. For instance, computing the cohesion metric involves multiplying $x$ by the overall number of operations in a given microservice (an integer number).

- Some formulas include the multiplication (or division) of a continuous variable and an integer variable, for instance the $Coh_m$ formula: there’s a division the variable storing the similarity between two operations if they are in the same microservice, and the variable storing the number of operations in such service. Both are derived from decision variable $x$.

Linearization of product of binary variables

Let us first consider two binary variables $b1$ and $b2$. To linearize their product, we introduce a new binary variable $b3$ that represents the value $b1 \cdot b2$. 

\[
\forall b1, b2, b3 
\]
\[ b_3 = b_1 \cdot b_2 \neq 0 \iff b_1 = b_2 = 1 \]

Which can be expressed with the following linear constraints:

\[ b_3 \leq b_1 \]
\[ b_3 \leq b_2 \]
\[ b_3 \leq b_1 + b_2 - 1 \]

**Linearization of product of a binary and a continuous variable**

Let us now consider a binary variable \( b \) and an integer variable \( n \). To linearize their product, we introduce a new integer variable \( p \) that represents the value of \( b \cdot n \). This product can otherwise be seen as:

\[ p = n \quad \text{if} \quad b = 1, \quad 0 \quad \text{otherwise} \]

Which can be expressed with the following linear constraints:

\[ (p - n) \leq (1 - b) \cdot M \]
\[ -(p - n) \leq (1 - b) \cdot M \]
\[ p \leq b \cdot M \]

The \( M \) represents an arbitrarily large number, which is common in linearization formulas, and appears in the technique commonly called **big M method**.
Linearization of the product of a continuous variable and an integer variable

The framework we chose to model the optimization problem, called Pyomo[18], and the solver we chose, called Gurobi[13], automatically linearize the product of a continuous variable and an integer variable whenever the integer variable can expressed as a sum of binary variables.

Let us now consider a continuous variable $c$ and an integer variable $i$. Provided that $i$ is computable as the sum of binary $b$ in a given set $S$:

$$i = \sum_{s \in S} b_s$$

Then we can linearize the product of $c$ and $i$ as it becomes a linearizable product of a continuous variable and a sum of booleans:

$$p = c \cdot \sum_{s \in S} b_s$$

All of the instances where such a product appears in our formulation, can be linearized in this way:

- In the computation of replication costs, with the formula:

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

  We have a product of the binary $accRW_{e,o}$ and the integer $f_o$ which can be linearized and saved into an auxiliary continuous variable, which is then multiplied legally with a sum of booleans.

- In the computation of $Coh_m$, with the formula:

  $$Coh_m = \frac{\sum_{o1 \in O_m,o2 \in O_m} S(o1,o2)}{|O_m|^2}, if|O_m| > 0, Coh_m = 0, otherwise$$
We can express it in the optimization problem as:

\[ \text{Coh}_m \cdot \sum_{o_1 \in O_m, o_2 \in O_m} x_{o_1,m} \cdot x_{o_2,m} = \sum_{o_1 \in O_m, o_2 \in O_m} S_{o_1,o_2} \cdot x_{o_1,m} \cdot x_{o_2,m} \]

As \( \sum_{o_1 \in O_m, o_2 \in O_m} x_{o_1,m} \cdot x_{o_2,m} \) represents the number of operation pairs in service \( i \), and containing a product of binaries it can be easily linearized in an auxiliary variable. The same auxiliary variable can be used for the product of the similarity (a continuous variable) in the right hand side of the equation. Then, such product is saved into another auxiliary variable after linearization. Moving the denominator to the left hand side allows to turn the division into a product which is linearized with the technique described in this section.

- In the computation of \( \text{Coh} \), obtained by multiplying the continuous variable \( \text{Coh}_m \) by the number of operations in \( m \) and dividing it by the total number of operations, the number of operations in \( m \) can be expressed as this sum of binaries:

\[ O_m = \sum_{o \in O_m} x_{o,m} \]

and thus we end up with a legal product of a continuous variable and an integer expressed as a sum of binaries. The subsequent division by the total number of operations is also perfectly legal and linear by default as that number is fixed and not a variable.
3 | Cromlech

3.1. Cromlech’s workflow

All the input data used by Cromlech is user supplied. To provide an automatic decomposition of the monolithic architecture given in input, Cromlech uses a linear optimization model which allows to obtain an optimal solution in the proposed representation of the problem. The algorithm starts from a YAML file which details the monolithic architecture. The user will also be prompted to supply a maximum number of microservices for the decomposition, and a parameter from 0 to 1 called cohesion-communication ratio, whose importance we will explain later.

From this specification, a .dat file is extracted which contains all the inputs necessary for the optimization model, after a preprocessing phase which arranges the architecture in an initial state to allow for more accurate optimization. This state is then fed to a LP solver, which will process it to provide the user with a graphical representation of the microservice architecture, in styled HTML format.

![Figure 3.1: Cromlech’s workflow](image-url)
3.1.1. Design of the input file

The input file the algorithm needs to extract the specification of the architecture needs to be produced by the user. We chose the YAML format as it is a versatile and concise markup language. The EBNF grammar describing the file structure is (axiom is FILE):

\[
\begin{align*}
\text{<FILE>} & ::= \text{operations:}<\text{OPERATION}>^+ \\
\text{<OPERATION>} & ::= \text{<NAME>}\ <\text{CONS}><\text{FREQ}><\text{FORCED-OP}><\text{DB-ACCESS}> \\
\text{<NAME>} & ::= \text{'- name:' name} \\
\text{<TRXN>} & ::= \text{'transactional: Y'} | \text{'transactional: N'} \\
\text{<FREQ>} & ::= \text{'frequency:' integer} \\
\text{<COLOC-OP>} & ::= \text{'colocated_operations:' -} | (- \text{name })^+ \\
\text{<DB-ACCESS>} & ::= \text{'database_access:' <ENTITY>}^+ \\
\text{<ENTITY>} & ::= \text{'- entity_name:' name <ATTRS>}^+ \\
\text{<ATTRS>} & ::= \text{['read_data entities: ' <ATTR}>^+ | \text{['write_data entities: ' <ATTR>}^+] \\
\text{<ATTR>} & ::= \text{'- name'}
\end{align*}
\]

The following excerpt gives a glimpse of how the file can be filled by a potential user.

```
operations:
- name: login
  transactional: N
  frequency: 6
  colocated_operations:
    - database_access:
      - entity_name: User
        read_attributes:
          - id
          - email
          - password
- name: SignUp
  transactional: N
  frequency: 2
  colocated_operations:
    - database_access:
      - entity_name: User
        write_attributes:
          - email
          - id
          - password
```
Rules for the compilation of the input file

The user needs to explicitly list all the data entities involved in an operation for the algorithm to provide a useful output. Moreover, he also needs to adhere to these rules:

- a data entity cannot be read and written at the same time by the same operation: this is because the model supposes the write access includes the read access.
- all the components are mandatory except the colocated operations list, which can be left empty.

3.2. Architecture extraction

This section describes the process the program undertakes to transform the YAML input into a .dat file ready to be processed by the optimization model.

3.2.1. Processing of the input file

This file is processed by a Python script to produce a set of dictionaries representing all the associations between operations and data entities and their characteristics. The microservice architecture is represented as a list of lists, where each list contains the IDs of the operations and data entities which are present in the corresponding microservice. Then, some operations are performed:

1. data entities which are accessed by a single operation are removed, as they only introduce noise to the measurements: they never introduce any communication cost (being present only in the microservice of the operation that reads them and consequently not requiring any cross-service communication) and they influence the similarity numbers. After this step, if some operation remains without accessed data entities (e.g. it only accesses data entities which are not accessed by any other operations), it is dropped from the architecture as it can be featured in any microservice of the final result without changing the costs, as it is unrelated to the other operations;
2. operations which are bound by the colocated operations relationship are permanently locked in the same microservice;
3. operations with transactionality requirements which write the same data entity are grouped in the same service, to fulfill the single writer principle. The maximum
cohesion architecture will be the decomposition with every operation in a separate service, except for colocated operations and operations with transactionality requirements (which are subject to the aforementioned process);
4. the minimum and maximum architecture cohesion are computed, respectively on the monolith and on the maximum cohesion architecture, to provide the bounds of such variable which will be used by the LP solver;
5. the maximum communication cost is computed on the maximum cohesion architecture, and it will be used to normalize the communication costs by the LP solver.

3.2.2. Creation of the solver file

After these processing steps, the Python script compiles a .dat file, which is the standard input of most popular LP solvers. It contains almost all the input parameters for the programming model.

3.3. Optimization problem solver

The model has been realized with the Python-based modeling language Pyomo, which allows, among other things, to interpret a .dat file as an abstract model. The chosen solver was Gurobi [17], a modern and fast LP solver which is also able to internally linearize some forms of quadratic constraints with great saving of computational time. The solver is able to assign the $x$, $y$ and $l$ variables in order to maximize cohesion and minimize communication costs with priority established by the cohesion/communication ratio. The variable values are stored by the optimizer in matrices which are then used by a script to produce a list of microservices, each holding operations and data entities with a marker used to understand their replication status.

3.4. Output files

From the $x$, $y$, and $l$ variables included in the LP solver’s solution, a Python script is ran to provide the user a graphical solution. The first output file is a graph realized in styled HTML which simply shows the computation by the preprocessing algorithm which yields the bound operations, without the operations removed by step 1 of the preprocessing. This files serves to let the user know which operations have been bound due to the colocated operations option or due to them being accessing the same data entity while requiring high transactionality requirements. The final output file is another styled HTML file which represents the whole architecture
as a graph which connects operations to the replicas they access in the same microservice. Operations are represented as dots of different colors:

- **navy blue** for transactional write operations;
- a **lighter blue** for non-transactional write operations;
- a **dark green** for transactional read operations;
- a **lighter green** for non-transactional read operations.

The size of the dots increases with the operations’ frequency, to indicate that the operation has more weight with regards to communication costs. Replicas are named following this convention:

\[ <entity\_name>.<data\_entity\_name>@<service\_id>/<num\ of\ replicas\ of\ attr.\ in\ arch.> \]

For example, "User.password@2/4" means that it is a replica of data entity "password" of entity "user", in service number two, and that there are 4 replicas of such data entity in the whole microservice architecture. Graphically, replicas are represented by squares of different colors:

- **red** if the replica is leader;
- **orange** if it is not leader, but cached;
- **yellow** if it is not leader, and uncached.

This way the users can understand which data entities he should not replicate and in which services, and which replicas should be the propagation leaders.

Two other output files will be produced:

- The textual version of the services, listing the names of the operations contained in each so that the user can eventually modify it and compute the new scores;
- A textual file listing all the data entities and operations removed by the preprocessing step.
Figure 3.3: Example of decomposition of an architecture.
4 Experiments and analysis

The purpose of this chapter is to illustrate the results obtained through *Cromlech* with multiple use cases. We applied *Cromlech* on two architectures: the one kindly offered by *Tutored*, and a microservice benchmark architecture called *Trainticket* [19, 20].

Concerning *Tutored*, our aim is to:

1. Measure the performance of *Cromlech* in terms of cohesion, communication costs, and absolute score (difference between cohesion and communication costs, regardless of alpha) against *Pangaea* and *ServiceCutter*, with *Cromlech*’s own metrics
2. Measure the similarity of solutions computed by *Cromlech* with the manual solution with an ad-hoc algorithms, along with the similarity of solutions of other algorithms
3. Evaluate *Cromlech*’s performance using *Pangaea*’s metrics

Concerning *Trainticket*, we want to:

1. Again evaluate *Cromlech*’s performance to see if it compares with the provided benchmark, which should be a high quality solution, with testing performed both with some transactional operations enabled and without them
2. Compare *Cromlech*’s solution with *ServiceCutter*’s

4.1. Decomposition similarity algorithm

To measure the similarity between two microservice decompositions of the same architecture, which is needed to compare the decomposition of *Cromlech* with the benchmark offered by *Trainticket* and with the manual decomposition of *Tutored* and the results of *Pangaea*, we decided to adopt a simple method that uses pairs of operations associated to a binary value, which is true if the operation pair is in the same service and false otherwise.

The algorithm is structured in a way which allows the comparison of decompositions with different number of services. It works this way:

1. Create two triangular matrices, A and B, for the two decompositions we want to compare. This will contain operation identifiers as both rows and columns to identify an operation pair. The cells will be set as 1 if the operations are both in the same service, 0 otherwise. The matrices will be need to be triangular, so we will discard
any cell where the row number is greater or equal than the column number (therefore also same operation pairs are excluded).

2. Then, we count all the cells where:
   • both matrices contain 1, we will call this count \( \text{both\_true} \);
   • both matrices contain 0, we will call this count \( \text{both\_false} \);
   • the first decompositions matrix contains 1 and the seconds 0, we will call this cont \( \text{only\_first} \);
   • the second decompositions matrix contains 1 and the firsts 0, we will call this cont \( \text{only\_second} \).

3. The total similarity of the two decomposition will be:

\[
S = \text{both\_true} + \text{both\_false} - \text{only\_first} - \text{only\_second}
\]

The measurement aims to valorize the pairs which are placed in the same service by both decompositions and the ones which are not placed together by both, and penalize the situation in which a pair is in the same service in one decomposition and not in the other. The algorithm works reasonably well even when comparing decompositions with extremely different number of services: for example, when comparing an almost monolithic decomposition with few services with a highly distributed one, we would have a very high \( \text{only\_first} \) number (as the almost-monolith has many more true pairings than the distributed architecture) and thus the score would be low.

In our results the similarity score will be normalized using the number of possible couplings as a maximum.

### 4.1.1. Decomposition similarity algorithm example

To better understand how the comparison algorithm works, here is an example.

We have two architectures:

\[
\text{Decomposition}_A = ([0, 1, 2], [3, 4, 5])
\]

\[
\text{Decomposition}_B = ([0, 2], [4, 5], [1, 3])
\]

Bear in mind that the numbers represent the operation indices, and the square brackets enclose each microservice. The algorithm starts computing the triangular matrices ("-" represents an invalid pair):
4 | Experiments and analysis

Then we compute the required parameters: both_true = 2, both_false = 8, only_first = 4, only_second = 1. According to these numbers, the final score will be

\[ S = 2 + 8 - 4 - 1 = 5 \]

If we want to normalize this number, we can divide it by the number of possible pairings, which is actually the number of assignable positions in one of the matrices, computable with:

\[ total\_couples = \frac{(op\_num - 1)^2 + op\_num - 1}{2} \]

Which is 15 in this case, so the normalized similarity will be 5/15.

4.1.2. Similarity in terms of data

This algorithm can be applied to pairs of leader replicas of data entities to establish a similarity metric in terms of data: essentially, this measures how often two solutions grouped the leader replicas of a pair of data entities in the same service, or, in other words, how much both decompositions tend to treat two pieces of data as related.

Due to the construction of our model, which expresses cohesion on the basis of operation similarity, and treats the presence of data entities in a service as a side-effect of the presence of operations accessing it, we decided to prioritize the comparisons performed on operations pairs, but we will also compute similarity on data entity presence for completeness.

The technique will be applied only to pairs of leader replicas, as their number is fixed (each data entity must have a single leader replica) and thus allows for easy normalization and reduces the variability associated to the possible presence (or absence) of a data entity in a service.
4.2. **Tutored**

*Tutored’s* architecture can be considered medium sized, as it features 71 operations and 271 data entities. We will compare *Cromlech’s* results with the manual decomposition by Tutored’s engineers, then with the results given by our predecessor *Pangaea*.

4.2.1. **Architecture characteristics**

The architecture has been given 7 transactional operations, mostly related to user session and user operations. The preprocessing step cuts out operations *Logout* and *retrieveExperienceTypes*, reducing the number of operations to 69, and several data entities, reducing the number of data entities to 166. The maximum communication cost, computed on the most distributed architecture possible, is 1568. In absence of user specified forced operations, the only bound operations are the ones relating to social network user operations, which are 4 transactional writes. The worst possible cohesion score, computed on the monolith, is 0.1579.

<table>
<thead>
<tr>
<th>operations</th>
<th>data entities</th>
<th>max num services</th>
<th>monolith cohesion</th>
<th>worst comm. cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>69</td>
<td>166</td>
<td>66</td>
<td>0.1579</td>
<td>1568</td>
</tr>
</tbody>
</table>

*Figure 4.1: Characteristics of *Tutored*’s architecture after preprocessing.*
4.2.2. Manual decomposition

The engineers at *Tutored* were asked to provide a decomposition of their monolithic architecture while being mindful of both cohesion and communication costs. The manual decomposition was provided in between 6 to 8 hours and is composed of 4 microservices. This number was chosen beforehand as they identified 4 main business domains.

The workers found that identifying business domains and thus focusing on cohesion was the easy part, but once the business domains were identified, distributing certain "fringe" operations to achieve lower cross-service communication felt much less straightforward, especially considering that multiple factors have to be kept in mind.

These problems become more pronounced for a human as the number of microservices grows, since complexity increases and keeping track of cross-service communication becomes exponentially harder. In fact, as we will show in the next subsections, 4 microservices is a very low number for an architecture of 69 operations. Moreover, it is challenging for a human worker to keep track of more than 150 different columns with their relative leader replicas: in fact, the decomposition was performed by keeping track of tables, at a coarser grade of granularity, instead of columns. This is because the decomposition was requested by our predecessor *Pangaea*, which considers tables in its model, instead of columns.

Their 4 microservices delineate distinct business domains:

1. the first one, which we can call the "**Content and activities service**", focuses on operations to deliver content such as streams, webinars, and also feed elements such as posts from employers. Also, it contains operations relating to activities and events.

2. the second one contains most user actions, so we can call it "**User service**", as it contains the traditional actions that allow the user to manage his account, and retrieve various related information. It also contains all the transactional write operations relating to social network functionalities.

3. the third service can be called "**Job service**", as it contains the operations related to job offers and interviews.

4. the last service contains mostly operations related to the users curriculum vitae, thus we can call it "**Curriculum service**", as it allows the user to manage everything related to its education, skills, experience, etc. It should be noted that a small subgraph related to "experience" operations is not connected to the main component.
Figure 4.2: From top left, clockwise: Content and activities, User, Job, Curriculum.
4 | Experiments and analysis

In depth quantitative analysis

The metrics reflect the issues they encountered: their cohesion metric is 0.3715, which is quite high for such a low number of services, as we will show, but their communication cost is 642 out of 1568 = 0.4094, which is bad considering such a low number of services.

<table>
<thead>
<tr>
<th>comm. cost</th>
<th>cohesion</th>
<th>absolute score</th>
<th>services</th>
<th>ops/services ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.4094</td>
<td>0.3715</td>
<td>-0.0379</td>
<td>4</td>
<td>17.25</td>
</tr>
</tbody>
</table>

Figure 4.3: Characteristics of Tutored’s manual decomposition.

The decomposition looks, from a qualitative standpoint, very sensible and well structured. All the services have a specific business domain and with low overlapping, except for the data entities of the entity Education, which is present in all four services and accounts for most of the communication costs:

- 55 (8.6% of the total) write communication cost, out of which 55 (100%) are related to Education data entities;
- 339 (52.8% of the total) replication costs, out of which 268 (79%) are related to Education data entities;
- 248 (38.6% of the total) read communication cost, out of which 137 (55.2%) related data entities of the entity Education.

This means that this decomposition has to exchange many messages to keep the secondary replicas of data entities of Education up to date.

Figure 4.4: Distribution of the communication costs for each entity.
Moreover, this widespread distribution of these data entities also hurts the cohesion of the architecture, as many operations which access the data entities of *Education* and are similar to one another, end up in different services. The services have these cohesion scores:

- the "Content and activities" service scores 0.0677 with 19 services out of 69;
- the "User" service scores 0.1145 with 14 services out of 69;
- the "Job" service scores 0.088 with 25 services out of 69;
- the "Curriculum" service scores 0.1013 with 11 services out of 69.

The "User" service scores the highest, thanks to the social network operations which access the same data entities, strongly improving the overall cohesion. The "Content and Activities" and "Job" service, however, negatively impact on the decomposition’s cohesion due to their size and diminished cohesion.
4.2.3. **Cromlech**’s solution

With Tutored’s architecture, we tried to

- find the most similar solution to the manual decomposition, with the maximum number of services capped at 4;
- find the best scoring solution without microservice number limit.

To do so, we ran different executions with varying alpha, with the runtime of the solver capped at 18000 seconds (5 hours) as the score of the solution starts growing very slowly after a few hours.

4 Service solutions overview

We ran different executions from 0 to 1 alpha, with steps of 0.1. These are the results we obtained (*similarity* is intended as the normalized similarity to the manual solution):

<table>
<thead>
<tr>
<th>alpha</th>
<th>cohesion</th>
<th>comm. cost</th>
<th>absolute score</th>
<th>similarity (op)</th>
<th>similarity (data)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>0.219</td>
<td>0.0</td>
<td>0.219</td>
<td>0.632</td>
<td>0.1316</td>
</tr>
<tr>
<td>0.1</td>
<td>0.3154</td>
<td>0.0</td>
<td>0.3154</td>
<td>0.632</td>
<td>0.234</td>
</tr>
<tr>
<td>0.2</td>
<td>0.3154</td>
<td>0.0</td>
<td>0.3154</td>
<td>0.631</td>
<td>0.319</td>
</tr>
<tr>
<td>0.3</td>
<td>0.3154</td>
<td>0.0</td>
<td>0.3154</td>
<td>0.631</td>
<td>0.327</td>
</tr>
<tr>
<td>0.4</td>
<td>0.3154</td>
<td>0.0</td>
<td>0.3154</td>
<td>0.631</td>
<td>0.187</td>
</tr>
<tr>
<td>0.5</td>
<td>0.334</td>
<td>0.0179</td>
<td>0.3161</td>
<td>0.66</td>
<td>0.249</td>
</tr>
<tr>
<td>0.6</td>
<td>0.334</td>
<td>0.0198</td>
<td>0.3142</td>
<td>0.673</td>
<td>0.14</td>
</tr>
<tr>
<td>0.7</td>
<td>0.384</td>
<td>0.072</td>
<td>0.312</td>
<td>0.753</td>
<td>0.518</td>
</tr>
<tr>
<td>0.8</td>
<td>0.3907</td>
<td>0.118</td>
<td>0.279</td>
<td>0.824</td>
<td>0.453</td>
</tr>
<tr>
<td>0.9</td>
<td>0.3935</td>
<td>0.12</td>
<td>0.2823</td>
<td>0.832</td>
<td>0.445</td>
</tr>
<tr>
<td>1.0</td>
<td>0.4047</td>
<td>0.5803</td>
<td>-0.1756</td>
<td>0.845</td>
<td>0.366</td>
</tr>
</tbody>
</table>

**Figure 4.5:** Main normalized metrics of solutions on Tutored’s architecture.

From the gathered data we can notice how cohesion score and communication costs both grow according to the alpha parameter, following the intended purpose for which the parameter was designed. The solutions up to alpha = 0.4 are extremely similar, and they all manage to use all 4 service slots by finding connected components in the complete graph representation of the architecture.

In all the solutions, the issue of the manual decomposition regarding the distribution of Education data entities is solved by grouping together more of the operations which access them. Although it may seem counter-intuitive, this increases the cohesion and lowers the
communication costs, effectively creating a quantitatively better decomposition. Another detail worth noticing is that after the alpha = 0.6 mark *Cromlech’s* solutions start outperforming the manual one in terms of cohesion, and every solution except the one with alpha = 1 (where communication costs are not taken into consideration) significantly outperform the manual solution in terms of communication costs, with even the 0.9 solution having a communication cost around 3.4 times smaller (0.12 vs 0.4094).

**Figure 4.6:** Evolution of communication cost in function of alpha.

**Figure 4.7:** Evolution of cohesion metric in function of alpha.
It is important to notice how the "sweet spot" for the absolute score is in the middle of the alpha range, and declines at the extremes. This means that the alpha parameter and the cost model have been correctly designed to discourage extreme solutions and favour solutions that balance the two sides of the tradeoff (cross service communication and cohesion/decoupling).

These findings corroborate the difficulties encountered by Tutored’s workers in lowering the cross-service communication, and the fact that the most similar architectures have alpha parameters exceeding 0.7 shows that Cromlech was able to produce a comparable decomposition only with extreme prioritization of cohesion requirements.

Data similarity, on the other hand, displays more erratic behavior due to being highly dependent on the very variable way with which leader replicas are chosen, but still hits a sweet spot on the 0.7-0.9 range, again confirming these findings.

Figure 4.8: Evolution of similarities in function of alpha.
We will now detail the four services provided by Cromlech in the decomposition with alpha equal to 0.9, which we believe to be a good solution due to its very high cohesion (0.3935) while still maintaining a very reasonable communication cost of 12% out of the maximum, and that is also very similar to the manual:

1. the "User service" identified by *Cromlech* is much larger than the corresponding one of the manual decomposition (32 operations vs 14), as it includes the main functionalities associated with the data entities of the entity *Education*. In fact, it also holds operations from the *Webinar* block and of the *Curriculum* block, which are strongly intertwined with the user operations through the entities of *Education*. This allows to save cross-service communication costs while maintaining an extremely high score of 0.1881, much more than the "User" services of the manual decomposition (0.1145);

2. the second service is a peculiar one, containing many different blocks which are cohesive among themselves but disconnected to each other (the *Skills*, *Experience*, *Academia* and *Languages* blocks). Overall, it looks like the optimizer created three cohesive services and left the small which were difficult to place in the other services together, to avoid hurting the cohesion of the bigger services. This service holds 17 operations and we will call it "Miscellaneous" service, and it is the worst with only 0.039 cohesion.

3. the third one is similar to the "Content and activities" of the manual, as it contains the same business logic, but mostly read operations. It is 12 operations large and scores 0.0871 in cohesion, which is still significantly better than the 0.0677 of its counterpart in the manual decomposition.

4. the last one contains operations pertaining to the "Job" business logic (interviews, job applications etc.). It is quite similar to the manual one for a portion, but holds less operations (8 vs 25). It scores 0.0793 in cohesion.
While the "User" service contributes in a manner that is proportionate to its size, the "Job" and "Content and activities" services contribute very positively to the overall cohesion even with their smaller size. The "Miscellaneous" is by far the worst and hurts the overall cohesion. Even if the "User" service is not the most useful for cohesion, its existence is important for the reduction of communication costs caused by the data entities of Education.

Figure 4.9: From top left, clockwise: User, Job, Content and Activities, Miscellaneous.
Qualitative comparison with the manual solution

From comparing the manual and the alpha = 0.9 *Cromlech* decomposition, we can remark some points in commons:

• Both kept a service for the business domain of user actions, although *Cromlech* is bigger, in fact the manual decomposition’s "User" service is a subset of *Cromlech*’s. Also, pretty much all of the "Curriculum" service of the manual decomposition except for the *experience*, *skills*, *academia* and *languages* block are included in the "User" service of *Cromlech*;

• The "experience", "skills", "languages" "academia", "webinar" blocks where grouped together in both, although in different services;

• The "Job" services are very similar to each other, but the manual’s includes a few read-only operations which *Cromlech* put in the "Content and activities" services;

• The "Content and activities" services have some overlap but they also differ because of the "webinar" block, which *Cromlech* placed in the "User" service due to its frequent access of *Education* data.

The biggest discrepancy is surely in the User service, as it varies from 14 in the manual to 32 in *Cromlech*’s. As outlined by the numerical evidence, the driving factor for the difference between the two is mainly the huge cross-service communication caused by the *Education* entity.

All in all, we can surely say that the manual decomposition identifies more distinct business domains and is semantically more sensible, and one could argue that qualitatively it makes more sense. However, since it was produced without keeping track of replication (assuming a fully replicated microservice architecture) and operation frequencies, as it is impossible, even for a team, to take all the different 69 frequencies into consideration, it performs poorly in cross-service communication, which is prohibitive as already explained. Although it is harder to identify distinct domains in the algorithmic decomposition, we are absolutely confident in its lower latency.

Can *Cromlech*’s solution be manually improved?

It seems clear from the solution proposed by our algorithm, is that 4 microservices are just too few for an application this large. In fact, in the decomposition there are 7 distinct connected components (without including the *retrieveSkillLevels* which is a single operation component): this means that all these different subgraphs can be placed in different services without suffering any consequence on cross-service communication, and actually improving cohesion. These connected components all belong to the "Miscellaneous" service, which can be split into 4 different services: one containing the "Experience" block,
one containing the "Skill" block, one containing the "Language" block, another containing the "Academia" block.

By applying this move we obtain a new decomposition of 7 services, where the communication cost is the same but the cohesion skyrockets to 0.5038, a 28% improvement on the previous 0.3935.

The small downside is a slight reduction in both similarity metrics, but the score remains quite high.

<table>
<thead>
<tr>
<th>comm. cost</th>
<th>cohesion</th>
<th>absolute score</th>
<th>services</th>
<th>similarity (op)</th>
<th>similarity (data)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.12</td>
<td>0.5038</td>
<td>0.3838</td>
<td>7</td>
<td>0.81</td>
<td>0.397</td>
</tr>
</tbody>
</table>

Figure 4.10: Metrics of the manually improved *Cromlech* decomposition

Figure 4.11: Improved solution, with 7 services, 0.5038 cohesion and 0.189 comm. cost.
4.2.4. *Cromlech*’s 15 service solution

We ran *Cromlech* on Tutored’s architecture with 15 services cap, and 0.9 alpha to emulate the prioritization of cohesion by Tutored’s workers. The result is an architecture with 0.8131 cohesion and 0.464 communication costs. The solution was obtained in around a day of computation, since imposing an higher service number increases the size of the matricial computations of the simplex algorithms. It is interesting to note that this decomposition scores a very high operation similarity to the manual solution, higher than the previously found best.

<table>
<thead>
<tr>
<th>comm. cost</th>
<th>cohesion</th>
<th>absolute score</th>
<th>similarity to manual (op)</th>
<th>similarity to manual (data)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.464</td>
<td>0.8131</td>
<td>0.3491</td>
<td>0.858</td>
<td>0.423</td>
</tr>
</tbody>
</table>

**Figure 4.12:** Metrics of *Cromlech*’s decomposition (15 services).

This decomposition still has more than half of its cost depending on the *Education* entity (423 out of 728) and around 30% depending on the *Application* entity (235 out of 728). The other entities have little to no communication costs. Surprisingly enough, even with 11 service more than the manual solution, this decomposition still has less communication costs associated to the *Education* entity, but overall the profile for the communication costs shifts noticeably. Nonetheless, the communication costs are not so far from the manual solution (0.464 vs 0.4094) even though this decomposition has almost quadruple the number of services.

These are the operations inside these services:

- Academia: `retrieveInstitutes, retrieveQualifications, retrieveMarks`
- Applications: `applyJobOffer, deleteStory, retrieveApplication, retrieveStoryVideo, startApply, uploadStory`
- Content: `RetrieveCommunityArticles, retrieveEmployerPosts, retrievePostDetail, retrieveStreamBySlug, retrieveStreamDetail, retrieveStreamsHome`
- Curriculum: `employerRetrieveCurriculum, employerRetrieveUserInfo, retrieveCurriculum, retrieveLanguageLevels, retrieveLanguages, retrieveSkillLevels, retrieveSkills, usersMe`
- Education: `createEducation, deleteEducation, eventCheckIn, registerToCareerDay, updateEducation, updateOrderEducation`
- Experience: `createExperience, deleteExperience, updateExperience, updateExperienceOrder`
4| Experiments and analysis

- Interview: deleteInterview, scheduleInterview
- Language: createLanguage, deleteLanguage, updateLanguageOrder
- Job Offer: applyJobOffer, deleteStory, retrieveApplication, retrieveStoryVideo, startApply, uploadStory
- Skills: createSkill, deleteSkill, updateSkillOrder
- Social: AppleV1, Facebook, ForgotPassword, Google, Signup, retrieveCurrentUniversityInfo, retrieveStudyTags
- User actions: ChangePassword, Login, ResetPassword, updateContacts, uploadCV
- User profile: deleteUser, retrieveCompanies, retrieveEmployerInfo, updateProfile, updateUserProfile
- Webinar: participateWebinar, retrieveWebinarParticipation, webinarOffline, webinarRegistration
- Webinar2: RetrieveContentsIdFromSlug, retrieveEmployerWebinars, retrieveWebinarDetail
Figure 4.13: *Cromtech*’s decomposition, with 15 services.
4.2.5. Pangaea decomposition

Pangaea is the predecessor to our work. It is also a semi-automatic tool which uses a linear programming model to solve the decomposition problem, but its workings are much different from Cromlech’s.

First of all, Pangaea represents data at the granularity of entities, thus coarser than data entities, for the Tutored architecture. Secondly, the input file is quite similar but the concept of forced operations is replaced by forced entities, meaning entities which are bound to stay in the same service as the referred operation.

Pangaea uses a much simpler metric model for the evaluation of costs:

- Communication cost as the sum of frequencies of operation which access entities outside their service, multiplied by a factor if it’s a write access, and another factor if it’s an high consistency access;
- Coupling cost is completely different, as it is evaluated as a property of entities rather than operations: it expresses the relationships between entities as three tiers (unrelated, weakly related, strongly related) and these tiers impose a customizable cost whenever two unrelated or weakly related entities are grouped together;
- It also introduces a replication cost as a value chosen by the user which is multiplied by the number of replicated entities. In our solution, replication costs are internal to the communication costs, as the presence or absence of a replica is influences cross-service interaction.

Therefore, rather than basing its metrics on the assumption of a single-writer protocol as Cromlech does, Pangaea assumes a set of values upon which its costs are calculated, and these factors are chosen by the user to modify the proposed solution. However, just like Cromlech, Pangaea offers a factor to modify the objective function in order to prioritize the reduction of either coupling costs, or communication and replication costs.
If we analyze the 4 service solution computed by Pangaea, it scores an average similarity to the manual, but it underperforms in both cohesion (0.306) and communication costs (0.2323). Moreover, the workers at Tutored found difficult assigning a business domain to each of the services.

<table>
<thead>
<tr>
<th>comm. cost</th>
<th>cohesion</th>
<th>absolute score</th>
<th>simil. to manual (op)</th>
<th>simil. to manual (data)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.2323</td>
<td>0.306</td>
<td>0.0737</td>
<td>0.8183</td>
<td>0.24</td>
</tr>
</tbody>
</table>

**Figure 4.14:** Metrics of Pangaea’s decomposition.

It can be noticed that, just like Cromlech, Pangaea includes various Education operations in the User service to lower the cross-service communication drastically.

**Figure 4.15:** Distribution of communication costs by entity in Pangaea’s decomposition.

Semantically, it is not very straightforward to identify the business domains in Pangaea’s solution:

- The "User" service includes many of Education’s data entities and also includes "skill" and "languages" blocks;
- There’s a service with "Experience", "CareerDay" and "Qualification" blocks, which we will call "Experience" service;
- A very large service including everything content-related and job offers, which we will call "Content and job offers" service;
4) Experiments and analysis

- A difficult to identify service with the "qualification" block, and operations `eventCheckIn` and `uploadCV`.

In terms of cohesion:
- the "User" service scores 0.1465 with 30 operations out of 69;
- the "Content and job offers" service scores 0.092 with 27 operations out of 69;
- the "Experience" service scores 0.0411 with 8 operations out of 69;
- the unidentified service scores 0.0264 with 4 operations out of 69.
Figure 4.16: Pangaea’s decomposition, with 4 services.
4.2.6. Evaluation of *Cromlech* with *Pangaea’s* metrics

In order to test the validity of our model against different measurements to mitigate the inherent bias in using our own metrics upon which the model was built, we decided to apply the metrics used by *Pangaea* to our 4 service, alpha 0.9 solution. Since *Pangaea* operates at table-level granularity, while we decided to operate at column-level granularity, we had to apply some slight changes in order to make the decomposition viable for *Pangaea’s* metrics:

- the pre-processing step where certain columns were trimmed off the architecture was reversed and such data entities were reinstated in the service where the only operations which accessed them was present
- each column was transformed into its parent table in the services where it is present, and duplicates of the same table in the same service were deleted.

This way we were able to represent *Cromlech’s* decomposition at table-level without moving any operation or data entity from its original microservice.

After testing, these were the results we obtained with all weights (replication, communication and coupling weights) set to the default value of 1:

<table>
<thead>
<tr>
<th>communication cost</th>
<th>coupling cost</th>
<th>replication cost</th>
<th>total cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>1646</td>
<td>54</td>
<td>1712</td>
</tr>
</tbody>
</table>

Figure 4.17: *Cromlech’s* decomposition (4 services, alpha 0.9) evaluation with *Pangaea’s* metrics.

We hereby provide the results of other decompositions obtained through various algorithms computed through *Pangaea’s* metrics, in order to compare them with *Cromlech*:

<table>
<thead>
<tr>
<th>SOLUTION</th>
<th>COMMUNICATION</th>
<th>COUPLING</th>
<th>REPLICATION</th>
<th>TOTAL COST</th>
<th>OPTIMAL SOL. DISTANCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manual</td>
<td>1553</td>
<td>1208</td>
<td>51</td>
<td>2812</td>
<td>1894</td>
</tr>
<tr>
<td>Pangaea (4)</td>
<td>112</td>
<td>840</td>
<td>46</td>
<td>1048</td>
<td>130</td>
</tr>
<tr>
<td>Pangaea (5)</td>
<td>140</td>
<td>731</td>
<td>47</td>
<td>918</td>
<td>0</td>
</tr>
<tr>
<td>Monolith</td>
<td>0</td>
<td>3818</td>
<td>45</td>
<td>3863</td>
<td>2615</td>
</tr>
<tr>
<td>Distributed</td>
<td>1430</td>
<td>0</td>
<td>45</td>
<td>1475</td>
<td>557</td>
</tr>
<tr>
<td>S. Cutter G. Newman (4)</td>
<td>0</td>
<td>3155</td>
<td>45</td>
<td>3200</td>
<td>2282</td>
</tr>
<tr>
<td>S. Cutter G. Newman (5)</td>
<td>0</td>
<td>2997</td>
<td>45</td>
<td>3042</td>
<td>2124</td>
</tr>
<tr>
<td>S. Cutter Leung</td>
<td>3267</td>
<td>672</td>
<td>45</td>
<td>4344</td>
<td>3426</td>
</tr>
<tr>
<td>S. Cutter Chinese W.</td>
<td>541</td>
<td>1210</td>
<td>45</td>
<td>1796</td>
<td>878</td>
</tr>
</tbody>
</table>

Figure 4.18: Various algorithms compared through *Pangaea’s* metrics.
As expected, *Cromlech* is outperformed by *Pangaea* with its own metrics, but actually performs quite well overall by placing second best out of all the algorithms, with one variation of *ServiceCutter* placing close in third position.

What is worth noting is that *Cromlech* has the highest replication cost, but this allows for a greatly reduced communication cost of only 12, even smaller than *Pangaea*’s solution, and beaten only by monolithic decompositions of *ServiceCutter*.

In the end, it was also predictable that *Cromlech* would perform quite poorly in coupling cost, as its decompositon was not based at all on the data-based parameters of *Pangaea* which are centered on common concepts of the ER paradigm such as composition and aggregation, but providing a completely different model based on the similarity of operations.
4.2.7. ServiceCutter decomposition

ServiceCutter [13] is a popular decomposition tool whose functioning we already described in Section 1.2. Like Pangaea (which is in fact based upon it), ServiceCutter identifies data elements (both at an entity and nanoentity) as the main elements to group to obtain the microservice result, instead of operations.

By using the entities of the Tutored architecture, and how they are related in traditional relational database terms (such as aggregation and composition), the author of Pangaea was able to obtain a decomposition from ServiceCutter. Starting from this decomposition that only specified entity placement, we were able to reverse engineer the placement of operation by placing them in the service where the entities they accessed were present, and choose the most optimal placement whenever an operation could be placed in multiple locations. It is important to specify that we omitted all data entities and operations which were trimmed by our preprocessing stage, so our interpretation of this decomposition can be compared with the others. Luckily, all these omitted elements made services on their own, so they generally decreased the plausibility of the solution.

Out of the multiple clustering algorithms which ServiceCutter proposes, the only one which did not produce an almost completely monolithic solution was Chinese Whispers [15], which consequently was the best scoring out of all the options. In fact, the Girvan-Newman[14] version proposed a solution which was very close to the monolith:

<table>
<thead>
<tr>
<th>comm. cost</th>
<th>cohesion</th>
<th>absolute score</th>
<th>simil. to manual (op)</th>
<th>simil. to manual (data)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.1636</td>
<td>0.1636</td>
<td>0.485</td>
<td>0.1263</td>
</tr>
</tbody>
</table>

Figure 4.19: Metrics of ServiceCutter’s decomposition (Girvan-Newman).

The main difference from the monolith is that this manages to separate the Experience connected component, with a small increase in cohesion.

Another algorithm, Leung [21], provided a more distributed solution, but still very expensive in cross-service communication:

<table>
<thead>
<tr>
<th>comm. cost</th>
<th>cohesion</th>
<th>absolute score</th>
<th>simil. to manual (op)</th>
<th>simil. to manual (data)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.3241</td>
<td>0.3701</td>
<td>0.0560</td>
<td>0.761</td>
<td>0.2196</td>
</tr>
</tbody>
</table>

Figure 4.20: Metrics of ServiceCutter’s decomposition (Leung).
The best solution by far between the ones provided by ServiceCutter is the one obtained through the Chinese Whispers algorithm: the result is a 5 service decomposition that scores average, 0.3505 on the cohesion and 0.1714 on the communication costs.

<table>
<thead>
<tr>
<th>comm. cost</th>
<th>cohesion</th>
<th>absolute score</th>
<th>simil. to manual (op)</th>
<th>simil. to manual (data)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1714</td>
<td>0.3505</td>
<td>0.1791</td>
<td>0.828</td>
<td>0.304</td>
</tr>
</tbody>
</table>

Figure 4.21: Metrics of ServiceCutter's decomposition (Chinese Whispers).

The main issue with this decomposition is the extreme variance in service sizes: it creates two large services of 33 and 29 operations, and three small ones of 4, 2, and 1 respectively. This is their description:

- the "User" service is made of 33 operations and is actually quite similar to the one proposed by Cromlech's 4 service decomposition, but it also includes the "skills" and "languages" blocks. It is almost identical to Pangaea's "User" service;
- another 29 operation service containing the "job offers", "webinars", "streams", "stories", "posts", "events" blocks, and thus can pretty much be referred to as "Content" service;
- a 4 operation service representing the "experience" block, so we will refer to it as "Experience" service;
- a 2 operation service containing the "retrieveMarks" and "retrieveQualifications" operations, thus we will call it "Qualifications" service;
- a single spare operation ("registerToCareerDay") which was probably left alone due to the entity CareerDay being unrelated to all the other entities.

All in all, the main weakness of this solution is its weak distribution, as it resembles a two service split of the original architecture, but nonetheless, it manages to score well in cohesion in spite of this factor.

For what concerns the communication costs, 0.1714 is still lackluster for a decomposition primarily made of only two sizeable services. the distribution of communication costs follows the usual pattern for this architecture, with all of the costs responding to the Education and User entities.
Let's now analyze the cohesion characteristics of this decomposition:

- the "User" service scores at 0.1549 with 33 operations out of 69;
- the "Content" service scores at 0.0956 with 29 operations out of 69;
- the "Experience" service scores at 0.058 with 4 operations out of 69;
- the "Qualifications" service scores at 0.0275 with 2 operations out of 69;
- the "CareerDay" service scores 0.0145 with 1 operation out of 69.
It can be noticed that the "Content" service is nowhere near as cohesive as the "User" service, and the smaller service have very high cohesion (as expected by services of such small size).

Figure 4.23: *ServiceCutter’s* 5 service decomposition.
4.2.8. Recap of *Tutored*’s decompositions

We will now compare the metrics associated to each of the presented decompositions.

**Figure 4.24:** Comparison of main metrics for every decomposition of *Tutored*’s architecture. Absolute score is the difference between cohesion and communication costs.

**Figure 4.25:** Comparison of cohesion metrics for every decomposition of *Tutored*’s architecture.
Figure 4.26: Comparison of communication costs for every decomposition of Tutored’s architecture.

Figure 4.27: Comparison of total score for every decomposition of Tutored’s architecture.
There are several main key takeaways that we can extract from the study of Tutored’s architecture:

- **4 services are not enough** for this architecture: the 7 and 15 services solutions greatly outperform the other solutions in cohesion (as expected) with minimal impact on communication costs. In fact, the 7 service solution, which is a slight manual optimization of the 4 service algorithmic one, preserves the same communication costs and increases the cohesion enormously. The 15 service solution has a similar the same communication costs as the manual solution, even with almost 4 times the number of services;

- the **manual solution** is outperformed in communication costs by almost every other decomposition, supporting the fact that humans are able to identify business domains easily, but are not skilled in arranging operations to facilitate the upkeep of consistent data;

- the **manual solution** will most likely produce very high latency due to cross-service communication;

- the **manual solution heavily favours and prioritizes cohesion** and Cromlech was able to produce similar results only with high alpha weights;

- communication costs were strongly dependent on the distribution of Education data entities;

- *Cromlech*’s sweet spot for solutions is in the middle range of alpha, and extreme solutions are discouraged;

- *Cromlech* is efficient at reaching low communication costs while keeping reasonable cohesion values, but it takes time to produce good solutions;

- *Pangaea* and *ServiceCutter* achieve close metrics and high similarities, outlining the similarity between the approaches;

- the **7 and 15 services solutions** are semantically valid and could actually be used, the 7 being the less distributed alternative;

- **algorithmic decompositions benefit from fine tuning by expert workers**, as they often include some minor dubious placements which can be changed to tweak the overall quality of the solution;

- overall, *Cromlech* outperforms the other algorithms in communication costs when the number of services is similar, except the case in which alpha is close to 1 and communication costs are overlooked.
4.3.  

Trainticket

Trainticket [19, 20] is a web application deployed on 41 microservices developed to study common faults and debugging of microservice systems. The authors developed the architecture to study and test different debugging methods after replicating common faults found in multiple industrial systems. Furthermore, it is also a well known and used benchmark representation of microservice systems, as it is large enough and distributed enough to provide a valuable example of a microservice architecture realized with best practices and constantly updated to reflect the evolution of the state of the art. It represents a mock application for buying and managing train tickets, including user registration and session management.

4.3.1.  Obtaining the YAML input file

The backend of Trainticket is realized mostly with the Java framework Spring, which is renowned for its great ease of use and modularity in developing distributed web applications, hence it was straightforward for us to extract the operations and accessed data from the source code and build the corresponding YAML representation. Of the original 41 services, we excluded from our representation:

- Services which did not possess operations that interact with persistent data, and thus are out of scope for our study as they do not possess any inherent communication cost nor a way to compute similarity to other operations. These services use external APIs to provide their functionalities, or perform some action like sending mails without working with data (these are, for example, ts-avatar-service, ts-news-service, ts-notification-service, ts-verification-service);
- Services which are a copy of other services as they provide a "specialized" version of other services, but are virtually identical, actually containing the same logic, operations and data (for example ts-travel-service and ts-travel2-service);
- Services which contain admin operations, and thus are not open to the users, and can be considered out of scope as they do not need load balancing and must be forcefully grouped in their own services.

We are left with 27 services and 124 operations. For what concerns the frequency of operations, we decided to set them all at the same value of 1, since we possessed no data about their usage.

We decided to fully utilize Cromlech’s capabilities by assigning some operations the transactionality requirement. These operations were mostly the ones involved in payment
actions, as it is a rational assumption that those operations are most probably transactions with a strong need for consistency semantics. Those operations are: `getTickets`, `updateUser`, `cancelOrderByUser`, `calculateRefund`, `processDelivery`, `pay`, `createPaymentAccount`, `addMoney`, `drawBack`, `payDifference`, `checkOrderValidity`, `preserve`, `rebook`, `payDifferenceRebook`, `distributeSeat`, `getLeftTicketsOfInterval`.

We also tested *Cromlech* on a version of the input without any transactional operations in order to provide a comparison with the original benchmark and with *ServiceCutter*’s solution that was not influenced by a priori grouping of transactional operations accessing the same data entities.

### 4.3.2. Metrics of *Trainticket*’s architecture, no transactional operations

The preprocessing step only removes one operation: there is only one unique data entity, and this also means that the operations are more interconnected than the ones of the original *Tutored* architecture. Moreover, the architecture has 124 operations and 137 data entities, that means there are on average 1,104 data entities per operation, again signifying that a lot of data entities are in common with many operations, many more than *Tutored*’s architecture, where each operation had 2,406 data entities on average.

The worst possible communication cost of 807: it is lower than *Tutored*’s even if it is larger, because we left operations at a default frequency of 1. The monolith’s cohesion is 0.0961, and since there are no bound operations the maximum number of services is equal to the number of operations.

<table>
<thead>
<tr>
<th>operations</th>
<th>data entities</th>
<th>max num services</th>
<th>monolith cohesion</th>
<th>worst comm. cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>124</td>
<td>137</td>
<td>124</td>
<td>0.0961</td>
<td>807</td>
</tr>
</tbody>
</table>

Figure 4.28: Characteristics of *Trainticket*’s architecture after preprocessing, without transactional operations.
4.3.3. Metrics of *Trainticket*’s architecture, with transactional operations

When considering the 16 previously listed operations which transactional requirements, the metrics of the architectures change slightly. Due to the groupings of transactional operations writing the same entities, the maximum number of services drops to 112. Clearly, the monolith cohesion does not change, but the worst communication cost gets slightly better, down from 807 to 663.

<table>
<thead>
<tr>
<th>operations</th>
<th>data entities</th>
<th>max num services</th>
<th>monolith cohesion</th>
<th>worst comm. cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>124</td>
<td>137</td>
<td>112</td>
<td>0.0961</td>
<td>663</td>
</tr>
</tbody>
</table>

Figure 4.29: Characteristics of *Trainticket*’s architecture after preprocessing, with transactional operations.
4.3.4. **Trainticket’s benchmark decomposition**

The original decomposition of the architecture is indeed well performing, as it shows a very high **cohesion of 0.8822** even with as many as 4.59 operations per service on average. If we consider the architecture without transactional operations the **communication cost is around 0.5328**, or a little above 53% of the max, which is also quite low for such a high number of services.

<table>
<thead>
<tr>
<th>comm. cost (no transact.)</th>
<th>cohesion</th>
<th>services</th>
<th>absolute score</th>
<th>services to ops ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.5328</td>
<td>0.8822</td>
<td>27</td>
<td>0.3494</td>
<td>4.59</td>
</tr>
</tbody>
</table>

Figure 4.30: **Trainticket’s original decomposition cohesion-related metrics.**

These numbers tell us **there are many highly distinct business domains in this architecture, and these bounded contexts have very high cohesion internally and decoupling from the others.** This is due to the fact that this architecture was conceived in a microservice format, with the precise purpose of representing the quintessential microservice architecture, while **Tutored’s** is born as a monolith like most web applications to this day. It also means that the algorithm should have an easier time finding the bounded contexts and producing a similar solution to the original one.

Concerning the communication costs, they total 430 and are split in 279 read costs (64.9% of the total), 117 write costs (27.3% of the total), and 34 replication costs (8% of the total). The most expensive entities are **Order** (155) and **Trip** with 105 followed by **Route** (64).

Figure 4.31: **Distribution of the communication costs for each entity.**
The following are the textual and graphical representations of \textit{Trainticket}'s decomposition.

These are the operations contained in the services:

- **Assurance**: getAllAssuranceTypes, getAllAssurances, findAssuranceById, findAssuranceByOrderId, createAssurance, deleteAssuranceById, deleteAssuranceByOrderId, modifyAssurance;
- **Auth**: saveUser, getAllUser, findById, findByUsername, deleteUserById, updateUser;
- **Basic**: queryForTravel, queryForStationId;
- **Cancel**: cancelOrderbyUser, calculateRefund;
- **Config**: createConfig, updateConfig, queryConfig, queryAllConfigs;
- **Consign-Price**: getPriceByWeightAndRegion, queryConsignPrice, createAndModifyPrice, getConsignPrice;
- **Consign**: insertConsignRecord, updateConsignRecord, queryConsignByAccountId, queryConsignByOrderId, queryConsignByConsignee;
- **Contacts**: findContactsById, findContactsByAccountId, createContact, deleteContact, modifyContact, getAllContacts;
- **Delivery**: processDelivery;
- **Execute**: ticketExecute, ticketCollect;
- **Food**: createFoodStore, createTrainFood, listFoodStores, listTrainFood, listFoodStoredByStationId, listTrainFoodByTripIds;
- **Food-Map**: createFoodOrdersInBatch, createFoodOrder, deleteFoodOrder, findFoodOrderByOrderId, findAllFoodOrders, updateFoodOrder, getAllFoods;
- **Order**: getSoldTickets, createOrder, cancelOrder, deleteOrder, updateOrder, modifyOrderStatus, getOrderPrice, payOrder, initOrder, checkOrderValidity;
- **Payment**: pay, createPaymentAccount, addMoney, queryPaymentAccount, queryPayments, drawBack, payDifference, queryMoney, initPayment;
- **Preservation**: preserve;
- **Price**: createNewPriceConfig, findPriceConfigById, findByRouteIdAndTrainType, findAllPriceConfig, deletePriceConfig, updatePriceConfig;
- **Rebook**: rebook, payDifferenceRebook;
- **Route**: createAndModifyRoute, getAllRoutes, getRouteById, getRouteByStartAndTerminal;
- **Route-Plan**: searchCheapestRouteResult, searchMinStopRouteResult, searchQuickestRouteResult;
- **Seat**: distributeSeat, getLeftTicketOfInterval;
Experiments and analysis

- Station: createStation, existStation, updateStation, deleteStation, queryStations, queryStationById;
- Ticket-Office: getAllOffices, getSpecificOffice, addOffice, deleteOffice, updateOffice;
- Train: createTrain, retrieveTrain, queryTrains, updateTrain, deleteTrain;
- Travel: getRouteByTripId, getTrainTypeByTripId, retrieveTrip, updateTrip, deleteTrip, getTickets, createTrip;
- Travel-Plan: getCheapestTravelResult, getQuickestTravelResult, getMinStopTravelResult;
- Voucher: addVoucher, queryVoucher.

Figure 4.32: Trainticket’s decomposition, with 27 services.
4.3.5. *Cromlech’s solution, without transactional operations*

We noticed that running the architecture without transactional operations required a much longer computation time due to the less constraints posed on the optimizer, thus we decided to run the architecture on *Cromlech* for a whole week to be sure to obtain significant results and try to improve on the benchmark score. We set an alpha of 0.925 to produce a solution similar to how a human would reason due to the high prioritization of cohesion shown by our study of *Tutored*, and capped the maximum number of services to 27, like the original decomposition.

The result is a decomposition with a slightly better cohesion of 0.8923 and greatly reduced communication costs of 0.20695, less than 40% of the original communication costs. Some services are completely identical to the original ones, and overall this solution has a very high similarity to the benchmark.

<table>
<thead>
<tr>
<th>comm. cost</th>
<th>cohesion</th>
<th>services</th>
<th>absolute score</th>
<th>op. similarity</th>
<th>data similarity</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.20695</td>
<td>0.8923</td>
<td>27</td>
<td>0.68535</td>
<td>0.9027</td>
<td>0.8523</td>
</tr>
</tbody>
</table>

Figure 4.33: Metrics of *Cromlech’s* decomposition of the *Trainticket* architecture

This decomposition reduces communication costs by aggregating a group of write operations which perform routing, orders and ticketing inside a service with many of *Order*, *Route* and *Trip* data entities leader replicas, while maintaining a very high cohesion as many of these operations share the same data entities.

![Graph showing communication cost distribution by entity of *Cromlech’s* decomposition.]

Figure 4.34: Communication cost distribution by entity of *Cromlech’s* decomposition.
We can immediately notice that communication costs associated to the *Order* entity were reduced by more than 4 times.

This is the list of services identified by *Cromlech*:

- **Assurance**: findAssuranceById, findAssuranceByOrderId, createAssurance, deleteAssuranceById, deleteAssuranceByOrderId, modifyAssurance;
- **Assurance-Type**: getAllAssuranceTypes, getAllAssurances;
- **Auth**: saveUser, getAllUser, findById, findByUserId, findByUsername, deleteUserById, updateUser;
- **Config**: createConfig, updateConfig, queryConfig, deleteConfig, queryAllConfigs;
- **Consign**: insertConsignRecord, updateConsignRecord, queryConsignByAccountId, queryConsignByOrderId, queryConsignByConsinee;
- **Consign-Price**: queryConsignPrice, getConsignPrice;
- **Contacts**: findContactsById, findContactsByAccountId, createContact, deleteContact, modifyContact, getAllContacts;
- **Execute**: ticketExecute, ticketCollect;
- **Food-Map**: processDelivery, createFoodOrdersInBatch, createFoodOrder, deleteFoodOrder, findById, findByOrderId, findAllFoodOrders, updateFoodOrder;
- **Food-Store**: createFoodStore, listFoodStores, listFoodStoredByStationId, getAllFoods;
- **Routing-Order**: queryForTravel, cancelOrderbyUser, getSoldTickets, createOrder, updateOrder, modifyOrderStatus, payOrder, preserve, rebook, searchMinStopRouteResult, searchQuickestRouteResult, distributeSeat, getLeftTicketOfInterval, createTrain, queryTrains, deleteTrain, getCheapestTravelResult, getQuickestTravelResult, getMinStopTravelResult, createTrip, getTrainTypeByTripId, updateTrip, getTickets;
- **Order**: cancelOrder, deleteOrder, getOrderPrice, initOrder;
- **Payment1**: calculateRefund, pay, createPaymentAccount, addMoney, queryPaymentAccount, drawBack, payDifference, queryMoney, payDifferenceRebook;
- **Payment2**: queryPayments, initPayment;
- **Price**: createNewPriceConfig, findById, findByRouteIdAndTrainType, findAllPriceConfig, deletePriceConfig, updatePriceConfig;
- **Route1**: searchCheapestRouteResult, getALLRoutes, getRouteByOrderId, getRouteByStartAndTerminal;
- **Route2**: createAndModifyRoute, getRouteByTripId;
- **Security**: findAllSecurityConfig, addNewSecurityConfig, modifySecurityConfig, deleteSecurityConfig;
- **Station**: queryForStationId, createStation, existStation, updateStation, deleteSta-
Experiments and analysis

- Ticket-Office: getAllOffices, getSpecificOffice, addOffice, deleteOffice, updateOffice;
- Train: retrieveTrain, updateTrain;
- Train-Food: createTrainFood, listTrainFood, listTrainFoodByTripIds;
- Trip: retrieveTrip, deleteTrip;
- Validity: checkOrderValidity, checkSecurityConfig;
- Voucher: addVoucher, queryVoucher;
- createAndModifyPrice;
- getPriceByWeightAndRegion.

Figure 4.35: Cromlech’s decomposition of the Trainticket architecture
Qualitative comparison between the benchmark decomposition and **Cromlech**’s

There are several differences between the services in our decomposition and the ones of the original, but there are also several identical services: *Auth, Config, Consign, Consign-Price, Contacts, Execute, Price, Ticket-Office, Voucher*. Some over services are very similar and only incurred in slight modifications: *Security* is almost identical except for one operation which was put in the new *Validity* service; the two services relevated to *Food*, which featured 4 subgraphs, were more conveniently split into three services (one very similar to *Food-Map*, one with food store operations, and one with foods served on trains); *Assurance* was split into two different services as a couple of operations interact with the *AssuranceType* entity rather than the *Assurance* one; the *Station* service has only one discrepant operation.

Overall, out of the original 27 services, we have 9 identical ones in our solution, and other 5 which were split into more services for higher cohesion, but with very slight modifications. In other words, more than half of the decomposition remained identical or almost, which is a positive result.

For what concerns the heavier differences:

- what we called the *Routing-Order* services is a highly cohesive service aggregating several operations (mostly write) from all the services which were involved in the routing, travel planning, and order services in the original solution. This service is extremely important as it is the greatest actor in the reduction of communication costs;

- some minor services (such as *Preservation* and *Delivery*) were aggregated into bigger ones which maintain good cohesion;

- services with a reasonable "semantic" sense but an ambiguous business domain at the data level, such as *Cancel* and *Seat*, which mostly interacted with entities from other domains, ended up in other services, mostly in the *Routing-Order* one.

Overall, the result can be deemed satisfactory as **Cromlech** was able to maintain an exceptionally high cohesion and reduce communication costs by more than half, all while highlighting distinct and semantically sound business domains.

The main drawback of the decomposition is the size of the *Routing-Order* service (23 operations, almost 19% of the total size), but it may be the cost to pay to have such low cross-service communication.
4.3.6. *ServiceCutter* solution

We ran *ServiceCutter* on *TrainTicket*’s architecture, providing an input file describing all the entities and their relationships of aggregation, composition and inheritance, and an additional optional file detailing all the operations (called *use cases* by *ServiceCutter*) and the data entities they read or write.

To compete on an even playfield with the work we did with the *Cromlech* input file, we decided to avoid specifying any detail regarding transactional requirements and operation frequency.

Upon the different choices of clustering algorithms which *ServiceCutter* provides, we chose *Girvan-Newman* [14] as it was the only one that allowed to choose a maximum number of services. Unfortunately, *ServiceCutter* was not able to produce 27 services, but maxed out at 21. Nonetheless, it provided a reasonable and well-performing decomposition, with 0.8436 cohesion, and 0.3135 communication cost.

<table>
<thead>
<tr>
<th>comm. cost</th>
<th>cohesion</th>
<th>services</th>
<th>absolute score</th>
<th>op. similarity</th>
<th>data similarity</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.3135</td>
<td>0.8436</td>
<td>21</td>
<td>0.5304</td>
<td>0.824</td>
<td>0.939</td>
</tr>
</tbody>
</table>

Figure 4.36: Metrics of *ServiceCutter*’s decomposition of the *TrainTicket* architecture

In cohesion, this solution is beat both by *Cromlech* and the original, but this is to be expected, as this decomposition possesses 6 less services. What is unexpected, seen the lower number of services, is it being also outperformed in terms of communication costs by *Cromlech*, and by a significant amount.

The solution is significantly less similar to the original in operation similarity, and much more similar in data similarity, compared to *Cromlech*’s.

Overall, the solution performs quite well, especially in the cohesion department, and many services have been built identically to the original decomposition, as we will see in the next pages.

Concerning communication costs, due to the similarity to the original one, the solution follows pretty much its same pattern, although scaled down by around 30 to 50% for most entities, with *Order*, *Trip* and *Route* taking up the majority of the cost.
Figure 4.37: communication cost distribution by entity of ServiceCutter’s decomposition.

The content of the 21 services is as follows:

- **Assurance**: `getAllAssuranceTypes`, `getAllAssurances`, `findAssuranceById`, `findAssuranceByOrderId`, `createAssurance`, `deleteAssuranceById`, `deleteAssuranceByOrderId`, `modifyAssurance`;
- **Auth**: `saveUser`, `getAllUser`, `findByUserId`, `findByUsername`, `updateUser`;
- **Auth2**: `deleteUserById`;
- **Config**: `createConfig`, `updateConfig`, `queryConfig`, `deleteConfig`, `queryAllConfigs`;
- **Consign**: `getPriceByWeightAndRegion`, `queryConsignPrice`, `createAndModifyPrice`, `getConsignPrice`, `insertConsignRecord`, `updateConsignRecord`, `queryConsignByAccountId`, `queryConsignByOrderId`, `queryConsignByConsignee`;
- **Contacts**: `findContactsById`, `findContactsByAccountId`, `createContact`, `deleteContact`, `modifyContact`, `getAllContacts`;
- **Delivery**: `processDelivery`;
- **Food-Map**: `createFoodOrdersInBatch`, `createFoodOrder`, `deleteFoodOrder`, `findFoodOrderByOrderId`, `findAllFoodOrders`, `updateFoodOrder`;
- **Food-Store**: `createFoodStore`, `listFoodStores`, `listFoodStoredByStationId`, `getAllFoods`;
- **Main**: `cancelOrderbyUser`, `ticketExecute`, `ticketCollect`, `getSoldTickets`, `createOrder`, `cancelOrder`, `deleteOrder`, `updateOrder`, `modifyOrderStatus`, `getOrderPrice`, `payOrder`, `initOrder`, `checkOrderValidity`, `preserve`, `rebook`, `distributeSeat`, `getLeftTicketOfInterval`, `checkSecurityConfig`, `getTickets`;
• Payment1: calculateRefund, pay, createPaymentAccount, addMoney, queryPaymentAccount, queryPayments, drawBack, payDifference, queryMoney, payDifferenceRebook;
• Payment2: initPayment;
• Price: createNewPriceConfig, findPriceConfigById, findByRouteIdAndTrainType, findAllPriceConfig, deletePriceConfig, updatePriceConfig;
• RoutePlan-Train: searchCheapestRouteResult, searchMinStopRouteResult, searchQuickestRouteResult, createAndModifyRoute, getRouteById, getRouteByStartAndTerminal, createTrain, retrieveTrain, queryTrains, updateTrain, deleteTrain, getAllRoutes;
• Security1: findAllSecurityConfig, addNewSecurityConfig, deleteSecurityConfig;
• Security2: modifySecurityConfig;
• Station: queryForStationId, createStation, existStation, updateStation, deleteStation, queryStations, queryStationById;
• Ticket-Office: getAllOffices, getSpecificOffice, addOffice, deleteOffice, updateOffice;
• Train-Food: createTrainFood, listTrainFood, listTrainFoodByTripIds;
• Travel-Route: queryForTravel, getCheapestTravelResult, getQuickestTravelResult, getMinStopTravelResult, createTrip, getRouteByTripId, getTrainTypeByTripId, retrieveTrip, updateTrip, deleteTrip;
• Voucher: queryVoucher, addVoucher.

Qualitative comparison between the benchmark decomposition and ServiceCutter’s

This solution has 7 identical services (Assurance, Config, Contacts, Delivery, Price, Ticket-Office, Voucher) to the original one, and interestingly enough, 5 of these are also identical in Cromlech’s solution (Config, Contacts, Price, Ticket-Office, Voucher). 4 services underwent slight modifications: Security has been split into two services (one contains only modifySecurityConfig), Auth also has been split into two (one contains only deleteUserId), and the same fate was reserved to Payment, again split into two with one being only initPayment.

It is interesting to notice how the Station service is identical to Cromlech’s, and possesses one more operation than the original. Moreover, except for processDelivery which is put in its own service just like the original, the three food related services are identical to Cromlech’s: overall, these common patterns allow for both solutions to somewhat reciprocally validate themselves.

In the end, this decomposition features two services, Travel-Route and RoutePlan-Train,
which are essentially the fusion of the homonymous services in the original, and a very large Main service which handles orders and common ticketing operations, approximatively similar to Cromlech’s Routing-Order service.

Figure 4.38: ServiceCutter’s decomposition of the Trainticket architecture
4.3.7. *Trainticket*'s solution, with transactional operations

Due to the added constraint of the 16 transactional operations we were able to repeat the experiments for varying levels of alpha, as obtaining a quality solution only required 20 hours instead of the several days taken by the input without transactional operations. The purpose of this round of testing was studying how the alpha parameter is able to influence the final results, and evaluate if the conclusions made about the analogous study of *Tutored* are also present in this evaluation.

Solutions overview

Just like we previously did for *Tutored*, we ran different executions from 0 to 1 alpha, with steps of 0.1, and with an execution time of 20 hours. This time we omitted similarity due to the fact that with the transactional operations, already a large number of operations are colocated and this strongly affects similarity, making this measurement out of scope for this study. These are the results we obtained:

<table>
<thead>
<tr>
<th>alpha</th>
<th>cohesion</th>
<th>comm. cost</th>
<th>services</th>
<th>absolute score</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>0.1367</td>
<td>0.0</td>
<td>2</td>
<td>0.1367</td>
</tr>
<tr>
<td>0.1</td>
<td>0.389</td>
<td>0.004</td>
<td>8</td>
<td>0.3886</td>
</tr>
<tr>
<td>0.2</td>
<td>0.483</td>
<td>0.021</td>
<td>14</td>
<td>0.462</td>
</tr>
<tr>
<td>0.3</td>
<td>0.569</td>
<td>0.04</td>
<td>22</td>
<td>0.565</td>
</tr>
<tr>
<td>0.4</td>
<td>0.7054</td>
<td>0.1056</td>
<td>27</td>
<td>0.599</td>
</tr>
<tr>
<td>0.5</td>
<td>0.8216</td>
<td>0.2</td>
<td>27</td>
<td>0.6216</td>
</tr>
<tr>
<td>0.6</td>
<td>0.8575</td>
<td>0.2473</td>
<td>27</td>
<td>0.6102</td>
</tr>
<tr>
<td>0.7</td>
<td>0.864</td>
<td>0.2579</td>
<td>27</td>
<td>0.6061</td>
</tr>
<tr>
<td>0.8</td>
<td>0.9041</td>
<td>0.3906</td>
<td>27</td>
<td>0.5135</td>
</tr>
<tr>
<td>0.9</td>
<td>0.9136</td>
<td>0.4494</td>
<td>27</td>
<td>0.4642</td>
</tr>
<tr>
<td>1.0</td>
<td>0.9305</td>
<td>0.728</td>
<td>27</td>
<td>0.2025</td>
</tr>
</tbody>
</table>

Figure 4.39: Main normalized metrics of solutions on *Tutored*'s architecture. (with 16 transactional ops)

It is immediately clear from the test results that, due to the larger size and thus much larger solution space of the *Trainticket* architecture compared to *Tutored*'s, there is a much more proportional and linear growth of communication costs and cohesion score with the increase of alpha.
The most interesting insights are:

- after alpha=0.3, *Cromlech* starts exploiting all the available service slots;
- the absolute score grows up to alpha=0.5 and stays somewhat similar up to alpha=0.7, then starts plummeting suggesting that the subsequent solutions’ cohesion score increase is not justified in proportion to their increase in communication costs;
- the benchmark is beaten in communication costs in all measurements except alpha=1;
- the benchmark is beaten in cohesion from alpha>=0.8;
- the benchmark is beaten in absolute score for all alpha except the extremes;
- *ServiceCutter* is beaten in communication costs for alpha <= 0.7;
- *ServiceCutter* is beaten in cohesion for alpha >= 0.5;
- *ServiceCutter* is beaten in absolute score for 0.3 <= alpha <= 0.7.

![Figure 4.40: Evolution of communication cost in function of alpha.](image)
Figure 4.41: Evolution of cohesion metric in function of alpha.

Figure 4.42: Evolution of absolute score in function of alpha.

Again, we find the same arguments elaborated during the Tutored study, especially:

- Overall it is again clear that <i>Cromlech</i> is more adept at reducing communication costs than its competitors and can produce, through the modulation of alpha, a wide array of very different solutions to satisfy the needs of different users;
• the sweet spot for the better solutions is again in the middle range of alpha. Compared to Tutored, this architecture provided more structured and linear measurements due to its mode distinct business domains and probably also due to the equal frequency assigned to each operations, which allowed other characteristics of the algorithm to show up in the final measurements.
Conclusion and future works

The testing results make us feel confident in Cromlech’s ability to produce plausible and functional results in architectures of varying sizes and domain number. The main differences between the Tutored and Trainticket cases are in fact in size and "context interconnectivity": while Tutored presents a more homogeneous architecture where contexts are not as easy to delineate, Trainticket presents a native microservice architecture with much clearer contexts. Nonetheless, in both scenarios Cromlech performed reasonably well.

The main motivation for a work like this is to make up for the difficulty encountered by human workers in finding a decomposition without excessive cross-service communication, while retaining some of the human-like skill to easily identify business domains. The algorithm succeeded in both tasks in all the solutions presented.

Overall, Cromlech can be used as a tool to produce an initial decomposition upon which experienced workers with knowledge of the architecture can start to provide the final microservice decomposition: it makes more sense to not take Cromlech’s proposal as the final choice, but to use it to guide the appointed workers. For example, the manually improved solution of the 4 service decomposition (at 4.9) is a slight modification of the algorithmic solution which greatly outperforms it (although introducing 3 more services). Nonetheless, there are several limitations in our approach that should be taken into account by anyone willing to improve it:

- our algorithm is time consuming and certainly not online: the time taken by the solver increases very sharply depending on the service cap, which involves most of the matricial computations of the optimization algorithm;
- there is still a manual component to the process which can be time consuming for bigger architectures, and the result still relies on the workers’ accuracy in determining operation frequency and transactionality;
- there are several distributed system paradigms to realize microservice architectures that might function in a different manner from single writer principle we assumed;
- the solver which we chose, Gurobi [17], is well performing and considered to be state of the art, but is a proprietary software requiring license.

For future works, we still have not seen sophisticated hybrid approaches combining aspects
of both static and dynamic decomposition: a work like *Cromlech* could be expanded to derive operation frequency from execution traces, and maybe incorporate some concept like the *time dimension* of *Mono2Micro* [16] to capture the time dependencies between operations and decrease cross-service communication.

Concerning the manual part of the process, it could be dropped by extracting the required information from widespread specification formats like *OpenAPI* (like Baresi et al. did [12] which are available *a priori* for some REST architectures.

In the end, the linear optimization approach can be remodeled to fit pretty much any existing paradigm for the realization of distributed systems, thus any formulation like *Cromlech’s* can be modified to suit the needs of any system and organization.
Bibliography


[11] Davide Taibi and Kari Systä. From monolithic systems to microservices: A decompo-


