7. Proposed solution

In this chapter, we propose our solution for identification of microservice candidates in a monolithic application. The approach is based on the analysis of a dependency graph, that aggregates information from the static and evolutionary analysis of the source code.

7.1. Problem statement

The goal of this solution is to identify a set of microservice candidates that can be extracted from the source code of the given monolithic application, in order to automate the migration to a microservices architecture. The problem can be formulated as a graph partitioning problem, where the vertices correspond to the modules or classes in the monolithic application, and the edges represent the dependencies between them. The input of the algorithm is a representation M of the monolithic application, which exposes a set of functionalities M_F through a set of classes M_C, and history of modifications M_H. The triplet is described by Equation 1.

$$M_i = \left\{ M_{F_i}, M_{C_i}, M_{H_i} \right\} \tag{1}$$

The set of functionalities M_{F_i}, the set of classes M_{C_i}, and the set of historical modifications M_{H_i} are described by Equation 2.

$$\begin{aligned} M_{F_i} &= \{f_1, f_2, ..., f_j\} \\ M_{C_i} &= \{c_1, c_2, ..., c_k\} \\ M_{H_i} &= \{h_1, h_2, ..., h_l\} \end{aligned} \tag{2}$$

The output of the algorithm is a set of microservices S, according to Equation 3, where m is the number of microservices in the proposed decomposition.

$$S_i = \{s_1, s_2, ..., s_m\} \tag{3}$$

As each class belongs to exactly one microservice, the proposed decomposition S can be written as a surjective function f of M_{C_i} onto S_i as in Equation 4, where $f(c_i) = s_j$ if class c_i belongs to microservice s_j.

$$f : M_{C_i} \rightarrow S_i \tag{4}$$

A microservice that contains only one class is called a *singleton microservice*. Singleton microservices typically contain classes that are not used by any other class in the monolithic application. As an optimization of the microservice decomposition, these classes can be omitted from the final decomposition, as they do not have any functional contribution.

7.2. Design

We start by identifying the functional and non-functional requirements for the solution. Then, we propose a four-step approach to decomposition adapted from the microservice identification pipeline by Lopes and Silva [62].

- **Extraction**: the necessary information is extracted from the application and its environment.
- **Decomposition**: using the collected data, a decomposition of the application into microservices is proposed.
- **Visualization**: the proposed decomposition is visualized to facilitate the understanding of the architecture.
- **Quality assessment**: the proposed decomposition is evaluated according to a set of quality metrics.

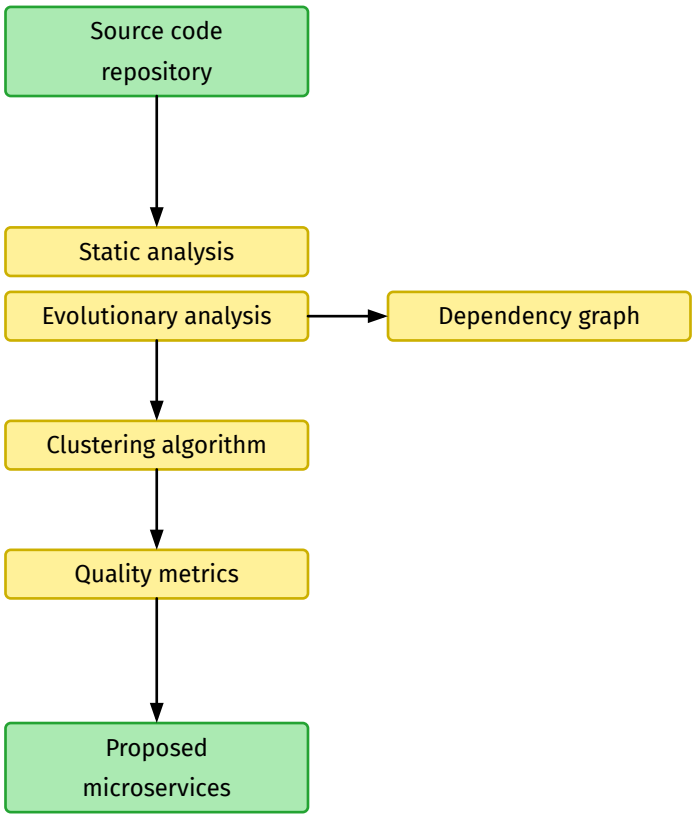


Figure 4: Overview of the architecture of the proposed solution

The next sections detail each of these steps, providing a comprehensive overview of the proposed solution. The process we describe is generic and not tied to any specific programming language or paradigm. We implemented a prototype of the proposed solution in Ruby, as the monolithic application we use for evaluation is written in Ruby. The implementation is available online[6].

[6]https://github.com/floriandejonckheere/mosaik

7.3. Requirements

Our approach needs to fulfill certain requirements. We make a distinction between functional and non-functional requirements. In software engineering, functional requirements describe requirements that impact the design of the application in a functional way . Non-functional requirements are additional requirements imposed at design-time that do not directly impact the functionality of the application .

The functional requirements we pushed forward for our proposed solution are as follows:

1. **Efficiency**: the solution decomposes the monolithic application into a microservices application with adequate efficiency
2. **Automation**: the solution automates the decomposition process as much as possible
3. **Technology**: the solution is written in a programming language that is compatible with the Ruby programming language[7]
4. **Visual**: the solution can output the proposed decomposition in a visual manner, to aid understanding of the process

The non-functional requirements identified in our solution are:

1. **Usability**: a software architect or senior software engineer can reasonably quickly get started with the solution
2. **Performance**: the solution performs the analysis, decomposition, and evaluation reasonably fast on the source code of a larger application
3. **Reuse**: The solution can successfully be reused for untested monolithic applications

[7] https://www.ruby-lang.org/

7.4. Extraction

Software development is typically done in multiple steps, either using the waterfall model , or using an iterative approach. Analysis and design are two steps of early software development which often yield software development lifecycle artifacts in the form of use cases, process models, and diagrams. However, after the completion of the development and the subsequent deployment, these documents are often not kept up to date, and sometimes even lost. Hence, it is not always possible to use design documents for the information extraction phase. A software development artifact that is usually available is the source code repository of the application. Hence, we chose the source code repository as the starting point of the information extraction.

Mazlami et al. [19] propose a microservice extraction model that includes three possible extraction strategies: *logical coupling* strategy, *semantic coupling* strategy, and *contributor coupling* strategy. In this thesis, we concentrate on the logical coupling strategy, and the contributor coupling strategy. The next sections describe in detail how these strategies are used for extracting information from the source code repository.

7.4.1. Structural coupling

Structural coupling is a measure of the dependencies between software components. The dependencies can take the form of control dependencies, or data dependencies . Control dependencies are dependencies between software components that are related to the flow of control of the software system (e.g. interleaving method calls). Data dependencies relate to the flow of data between software components (e.g. passing parameters). In our proposed solution we extract structural coupling information using static analysis of the source code . As the solution is intended to collect information from monolithic applications written in the Ruby programming language, the static analysis is limited to the information that is embedded in the source code. Ruby is a dynamic language, which means that only incomplete type information can be extracted using static analysis. In particular, some techniques like meta-programming and dynamic class loading may affect the accuracy of the extracted information.

Our solution analyzes the source code of the monolithic application using the `parser` library[8]. The library is written in Ruby and can be used to parse Ruby source code files and extract the AST of the source code. Iterating over the AST of the monolithic application, our solution extracts the references between classes.

Using this information, a call graph is constructed that represents the structural coupling of the monolithic application.

For each class in the monolithic application $c_i \in M_C$, a vertex is created in the call graph. References between classes are represented as directed edges between the vertices.

[8] https://github.com/whitequark/parser

A directed edge is created for each reference between two classes c_i, c_j. This edge describes three types of references: (i) static method calls between two methods m_i and m_j of the classes c_i, c_j *(method-to-method)*, (ii) references from method m_i to an object of class c_j *(method-to-entity)*, and (iii) associations between entities of class c_i and c_j *(entity-to-entity)* [24].

Hence, the structural coupling N_s for each pair of classes $c_i, c_j \in M_C$ is defined as the sum of the number of references between the classes, as described in Equation 5.

$$N_s(c_i, c_j) = \sum_{m_i \in c_i, m_j \in c_j} ref_{mm}(m_i, m_j) + ref_{mc}(m_i, c_j) + ref_{cc}(c_i, c_j) \tag{5}$$

The ref_{mm}, ref_{mc}, and ref_{cc} functions return the number of references between the two methods m_i and m_j, method m_i and class m_j, and classes c_i and c_j respectively.

As Luiz Carvalho, Alessandro Garcia, Colanzi, et al. [22] note, the choice of granularity is an important decision in the extraction of microservices. Existing approaches tend to use a more coarse-grained granularity (e.g. on the level of files or classes) rather than a fined-grained granularity (e.g. on the level of methods). Using a coarse-grained granularity can lead to a smaller number of microservices that are responsible for a larger number of functionalities. A fine-grained granularity can lead to a much larger number of microservices, which can decrease the maintainability of the system. Hence, a trade-off between the two granularities must be made. Our proposed solution uses a coarse-grained granular approach, using the classes of the monolithic application as the starting point for the extraction of microservices.

Consider the extraction algorithm in pseudocode in Algorithm 1.

Algorithm 1: Structural coupling extraction algorithm

```
calls ← array[][][]

for each ( class : classes )
   for each ( method : class.methods )
      for each ( reference : method.references )
         receiver ← reference.receiver
         type ← reference.type
         calls[class][receiver][type] ← 1

return calls;
```

Algorithm 1: Structural coupling extraction algorithm

7.4.2. Logical coupling

The logical coupling strategy is based on the Single Responsibility Principle [63], which states that a software component should only have one reason to change. Software design that follows the Single

Responsibility Principle groups together software components that change together. Hence, it is possible to identify appropriate microservice candidates by analyzing the history of modifications of the classes in the source code repository. Classes that change together, should belong in the same microservice. Let M_H be the history of modifications of the source code files of the monolithic application M. Each change event h_i is associated with a set of associated classes c_i that were changed during the modification event at timestamp t_i, as described by Equation 6 [19].

$$h_i = \{c_i, t_i\} \tag{6}$$

If c_1, c_2 are two classes belonging to the same change event h_i, then the logical coupling is computed as follows in Equation 7 [19].

$$\Delta(c_1, c_2) = \sum_{h \in M_H} \delta_{h(c_1, c_2)} \tag{7}$$

Where δ is the change function.

$$\delta(c_1, c_2) = \begin{cases} 1 \text{ if } c_1, c_2 \text{ changed in } h_i \\ 0 \text{ otherwise} \end{cases} \tag{8}$$

Then, Equation 7 is calculated for each change event $h_i \in M_H$, and each pair of classes c_1, c_2 in the change event. Thus, the logical coupling N_c for each pair of classes $c_i, c_j \in M_C$ is defined as the sum of the logical coupling for each change event $h_i \in M_H$.

$$N_c(c_1, c_2) = \Delta(c_1, c_2) \tag{9}$$

Consider the extraction algorithm in pseudocode in Algorithm 2.

Algorithm 2: Logical coupling extraction algorithm

```
cochanges ← array[][]

for each ( commit : git.log )
    parent ← commit.parent
    parent_diff ← diff ( commit, parent )

    for each ( file_one : parent_diff.files )
        for each ( file_two : parent_diff.files )
            cochanges[file_one][file_two] ← 1

return cochanges;
```

Algorithm 2: Logical coupling extraction algorithm

7.4.3. Contributor coupling

Conway's law states that the structure of a software system is a reflection of the communication structure of the organization that built it [64]. The contributor coupling strategy is based on the notion that the communication structure can be recovered from analyzing the source code repository [19]. Grouping together software components that are developed in teams that have a strong communication paradigm internally can lead to less communication overhead when developing and maintaining the software system. Hence, identifying microservice candidates based on the communication structure of the organization can lead to more maintainable software systems.

Let M_H be the history of modifications of the source code files of the monolithic application M. Each change event h_i is associated with a set of associated classes c_i that were changed during the modification event at timestamp t_i. The change event h_i is also associated with a set of developers $d_i \in M_D$, as stated in Equation 10 [19]. M_D is the set of developers that have contributed to the source code repository of the monolithic application.

$$h_i = \{c_i, t_i, d_i\} \tag{10}$$

$H(c_i)$ is a function that returns the set of change events that have affected the class c_i, and $D(c_i)$ returns the set of developers that have worked on the class c_i.

$$H(c_i) = \{h_i \in M_H \mid c_i \in h_i\} \tag{11}$$

$$D(c_i) = \{d_i \in M_D \mid \forall h_i \in H(c_i) : d_i \in h_i\} \tag{12}$$

Then, Equation 12 is calculated for each class $c_i \in M_C$ in the monolithic application.

Finally, the contributor coupling N_d for each pair of classes $c_i, c_j \in M_C$ is defined as the cardinality of the intersection of the sets of developers that have contributed to the classes c_i, c_j [19].

$$N_d(c_1, c_2) = |D(c_i) \cap D(c_j)| \tag{13}$$

Consider the extraction algorithm in pseudocode in Algorithm 3.

Algorithm 3: Contributor coupling extraction algorithm

```
coauthors ← array[][]

for each ( commit : git.log )
    parent ← commit.parent
    parent_diff ← diff ( commit, parent )

    for each ( file : parent_diff.files )
        coauthors[file] ← commit.authors

return coauthors;
```

Algorithm 3: Contributor coupling extraction algorithm

7.4.4. Dependency graph

As a final step in the information extraction phase, an edge-weighted graph $G = (V, E)$ is constructed, where V is the set of classes in the monolithic application, and E is the set of edges between classes that have an interdependency based on the discussed information extraction strategies. The weight for the edge e_i between classes $c_j, c_k \in V$ is calculated as the weighted sum of the call graph N_s representing the structural coupling, the co-change matrix N_c representing the logical coupling, and the co-authorship matrix N_d representing the contributor coupling. The weights $\omega_s, \omega_c, \omega_d \in [0, 1]$ are used to balance the contribution of the structural, logical, and contributor coupling respectively, as described in Equation 14. This makes the strategy adaptive and flexible [39].

$$w(e_i) = w(c_j, c_k) = \omega_s N_s(c_j, c_k) + \omega_c N_c(c_j, c_k) + \omega_d N_d(c_j, c_k) \tag{14}$$

An illustration of the graph G is presented in Figure 5.

$G = (V, E)$

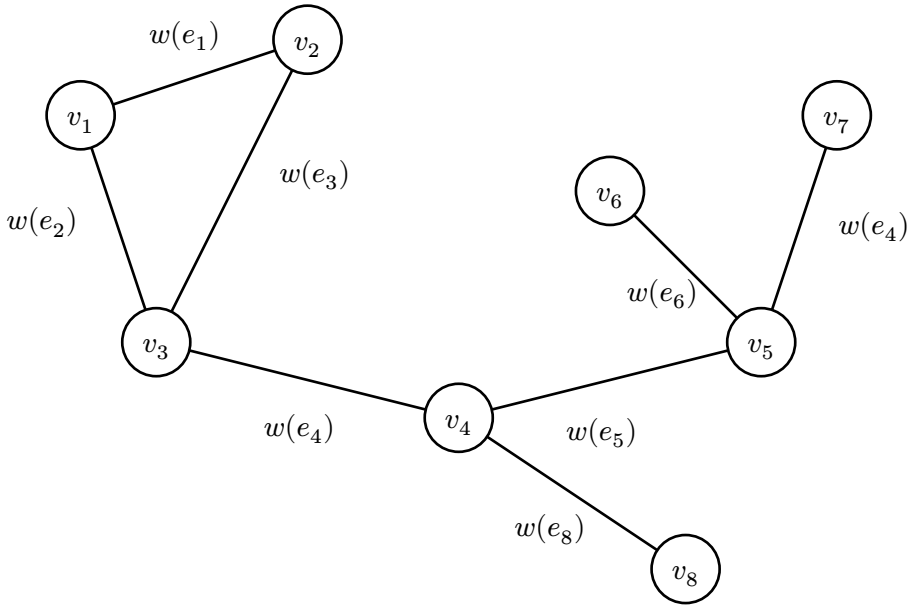


Figure 5: Dependency graph

7.5. Decomposition

Monolith decomposition is the process of identifying microservice candidates in a monolithic application. The goal of this process is to split the monolith into smaller, more manageable software components which can be deployed independently. Traditionally, monolith decomposition is a manual process that requires a deep understanding of the software architecture and business requirements. However, the burden of manual decomposition can be alleviated by using automated tools and algorithms. The knowledge of software architects should be leveraged where possible to guide the decomposition process, without imposing the requirement of a deep understanding of the software architecture. For example, Li et al. [12] propose a method that utilizes expert knowledge, however requires the recommendations to be written in a domain-specific language, increasing the burden on the architect.

Our solution proposes an automatic way to identify microservice candidates in a monolithic application using clustering algorithms. The decomposition process can be fine-tuned by assigning an importance to the different types of coupling strategies. This way, the software architect can decide which coupling strategies are most relevant to the decomposition process.

Clustering algorithms group similar elements together based on one or multiple criteria. Generally these algorithms work iteratively either top-down or bottom-up. Top-down algorithms start by assigning all elements to one big cluster, and then progressively split it into smaller clusters until a stopping criterion is met. Bottom-up algorithms on the other hand, start by assigning each element to its own cluster, then merge suitable clusters together in succession, until a stopping criterion is met.

Selection of algorithm

In Chapter 6.3.2, we performed an analysis of the state of the art in clustering algorithms used for microservice candidate identification. We considered the following criteria when selecting the most suitable algorithm for our task:

1. **Automation**: The algorithm should not require architectural knowledge up-front (e.g. number of clusters)
2. **Performance**: The algorithm should be able to handle large datasets efficiently

The first criteria disqualifies algorithms that require specifying the number of clusters up-front, such as Spectral Clustering, K-Means, and Agglomerative Clustering. Search-based algorithms (e.g. genetic, linear optimization) were considered as well, due to their inherent ability to optimize multiple objectives [22]. However, they require a lot of computing resources, and proper fine-tuning of parameters such as population size, mutation rate, and crossover rate, which makes them less suitable. Similarly, graph algorithms such as Kruskal's algorithm and Label Propagation were considered, but they may not always work on this type of graph due to the nature of the algorithm.

We found that the Louvain [51] and Leiden [52] algorithms are the most suitable for this task, as they are designed for optimizing modularity in networks. The algorithms are iterative and hierarchical, which makes them fast and efficient.

Similarly, in 2019, S. Rahiminejad, M. R. Maurya, and S. Subramaniam [65] performed a topological and functional comparison of community detection algorithms in biological networks. They analyzed six algorithms based on certain criteria such as appropriate community size (not too small or too large), and performance speed. The authors found that the Louvain algorithm [51] performed best in terms of quality and speed.

The Louvain/Leiden algorithm

The Louvain algorithm, introduced by Blondel et al. [51], is an algorithm for extracting non-overlapping communities in large networks. The algorithm uses a greedy optimization technique to maximize the modularity of the network.

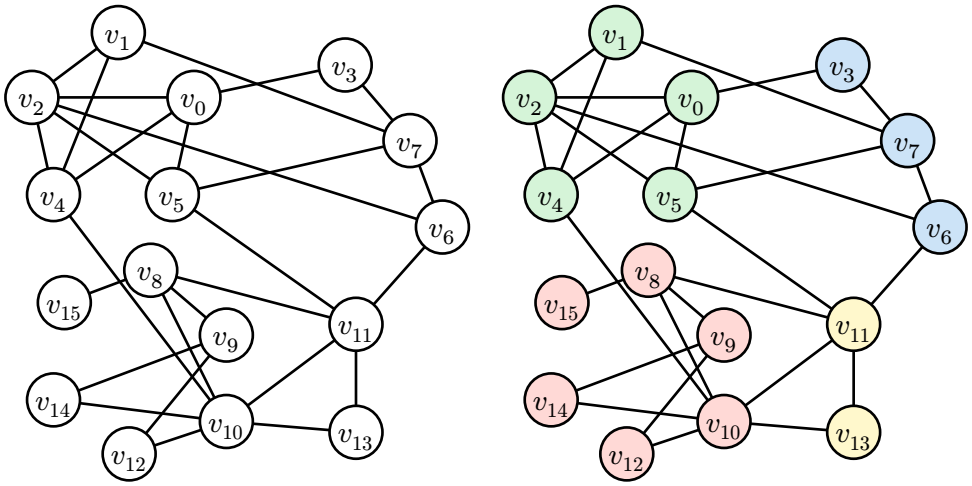
Modularity is a measure of the strength of division of a network. Networks with high modularity have dense connections between the internal vertices of a community, and sparse connections between vertices of different communities. The domain of the metric is between −0.5 (non-modular clustering) and 1 (fully modular clustering). Optimizing the modularity theoretically results in the best possible clustering of the network, though for numerical computing reasons, the algorithm uses heuristics to approach the optimal solution.

The modularity of a network is defined as follows [66]:

$$Q = \frac{1}{2m} \sum_{i=1}^{N} \sum_{j=1}^{N} \left[A_{\text{ij}} - \frac{k_i k_j}{2m} \right] \delta(c_i, c_j) \tag{15}$$

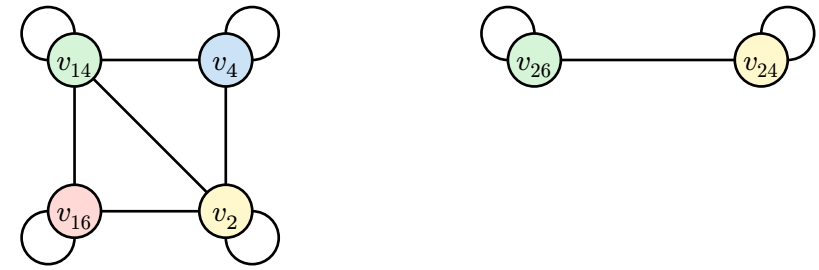
Where:

- A is the adjacency matrix
- k_i and k_j are the degrees of the vertices i and j respectively
- m is the number of edges in the network
- N is the total number of vertices in the network
- c_i and c_j are the communities to which vertices i and j belong
- $\delta(c_i, c_j)$ is 1 if c_i and c_j are in the same cluster, and 0 otherwise



Original network

Modularity optimization



Community aggregation

Second iteration

Figure 6: Louvain algorithm intermediate steps

Algorithm 4: Louvain algorithm

graph ← original network

loop
  for each *vertex* **in** *graph*
    Put *vertex* in its own community

  // Phase 1: local modularity optimization
  for each *neighbour* **in** *vertex.neighbours*
    Move *vertex* to community of *neighbour*
    if modularity gain
      break

  // Phase 2: community aggregation
  for each (*community* : *graph*)
    Reduce *community* to a single vertex

  if modularity no longer increases
    break

Algorithm 4: Louvain algorithm pseudocode

The Louvain algorithm operates in two phases. In the first phase, the algorithm optimizes the modularity locally by moving each vertex into the community of their neighbour that yield the best modularity gain. This step is repeated for each vertex until a local maximum is reached.

Then, the algorithm aggregates each community in a single vertex, while preserving the network structure. The algorithm can then be applied iteratively to the new network, until the modularity cannot be further increased.

A visualization of the intermediate steps of the Louvain algorithm is shown in Figure 6.

A major disadvantage of the Louvain algorithm is that it can only detect non-overlapping communities [51]. This means that a software component can only belong to one microservice, which is not in line with the principle of reuse in software engineering. The algorithm has also been proven to generate small and disconnected communities [52], which is not desirable in the context of microservices [67].

In 2019, Traag et al. [52] introduced the Leiden algorithm, an improvement of the Louvain algorithm that addresses the disconnected community problem. Similarly to the Louvain algorithm, the Leiden algorithm optimizes the quality of the network using the Constant Potts Model [68]:

$$\mathcal{H}(G, \mathcal{P}) = \sum_{C \in \mathcal{P}} |E(C, C)| - \gamma \binom{\|C\|}{2} \tag{16}$$

The Leiden algorithm operates in three phases. The first and last phases equal those of the Louvain algorithm (i.e., local modularity optimization and community aggregation).

In the second phase, the algorithm performs a refinement of partition on each small community. The refinement ensures that the algorithm does not get stuck in a local optimum using a probability distribution. The Leiden algorithm has been shown to outperform the Louvain algorithm in terms of quality and speed [52].

Although the Leiden algorithm is more complex than the Louvain algorithm, it is more suitable for our task due to its ability to detect overlapping communities.

Algorithm 5: Leiden algorithm (refinement)

```
// Phase 2: partition refinement
for each ( community : graph )
  partition ← community
  for each ( well connected vertex : partition )
    if vertex is a singleton
      assign vertex to new community
      using probability distribution P
```

Algorithm 5: Leiden algorithm (refinement)

7.6. Visualization

Our proposed solution is able to generate visualizations of the microservice decomposition process. The information extraction step can be visualized as a dependency graph where the vertices represent the classes of the monolithic application, and the edges represent the dependencies between the classes. The edges are weighted according to the calculated coupling factor between the classes. Depending on the weights assigned to the structural, logical, and contributor coupling, the generated graph can visualize a different view of the monolithic application.

Visualization of the decomposition process yields a dependency graph where the vertices represent the microservice candidates, and the edges represent the dependencies between the microservice candidates.

The graphs generated by the different steps in the decomposition process are saved as CSV files. The visualizations are generated using a custom implementation, that converts the graphs into the DOT language[9], before rendering them as images using the `dot` tool from the Graphviz package[10].

[9] https://graphviz.org/doc/info/lang.html
[10] https://graphviz.org/

7.7. Evaluation

Several quality metrics were considered to evaluate the quality of the microservice decomposition. Luiz Carvalho, Alessandro Garcia, K. G. Assunção, et al. [69] conducted an analysis of the criteria used to evaluate the quality of microservice-based architectures and found that the most common metrics are cohesion and coupling, as does our literature review win Chapter 6. In addition to this, they conclude that in many cases the decomposition process is guided only by these two metrics. In other cases, the software architects use other metrics (e.g. network overhead, CPU usage), although coupling and cohesion remain the dominant ones. According to experts, using four quality metrics is a good balance between the number of metrics and the quality of the solution [69].

Candela et al. [58] studied a large number of metrics to evaluate the quality of microservice-based architectures, including network overhead, CPU, and memory. They affirmed the need for more than two metrics to ensure the quality of the decomposition.

We distinguish between functional and non-functional metrics.

7.7.1. Functional metrics

Coupling

Coupling is a measure of the degree of interdependence between modules in a software system [70]. In the context of microservices, individual coupling is defined as the sum of static calls from methods within a microservice candidate M_c in a solution S to another microservice candidate $M_c \in S$ [22].

$$coup(M_c) = \sum_{v_i \in M_c, v_j \notin M_c} calls(v_i, v_j) \tag{17}$$

Where v_i and v_j are methods belonging and not belonging to M_c respectively, and $calls$ returns the number of method calls present in the body of method v_i made to method v_j.

The total coupling of a solution is the sum of the individual couplings of all microservice candidates M_c.

$$Coupling = \sum_{M_c \in S} coup(M_c) \tag{18}$$

A lower total coupling indicates a better decomposition.

Cohesion

Cohesion is a measure of the degree to which internal elements of a module in a software system are related to each other [70]. Microservice candidates cohesion is defined as the number of static calls between methods within a microservice candidate M_c in a solution S, divided by all possible

existing static calls in S [22]. The metric indicates how strongly related the methods internal to a microservice candidate are.

To compute the individual cohesion of a microservice candidate M_c, we first introduce the boolean function ref, which indicates the existence of at least one static call between methods v_i and v_j in D.

$$ref(v_i, v_j) = \begin{cases} 1 \text{ if } calls(v_i, v_j) > 0 \\ 0 \text{ otherwise} \end{cases} \tag{19}$$

The cohesion of a microservice candidate M_c is then calculated as described in Equation 20.

$$coh(M_c) = \frac{\sum_{v_i \in M_c, v_j \in M_c} ref(v_i, v_j)}{|M_c| \ \frac{|M_c|-1}{2}} \tag{20}$$

The total cohesion of a solution is the sum of the individual cohesion of all microservice candidates M_c.

$$Cohesion = \sum_{M_c \in S} coh(M_c) \tag{21}$$

A higher total cohesion indicates a better decomposition.

Complexity

Complexity is a measure of the number of operations performed by a method in a class [13]. We use the number of operations to compute the individual complexity of a microservice candidate M_c in a solution S. The metric is based on the Weighted Methods per Class (WMC) metric, which is the sum of the complexities of all methods in a class [71]. Classes and methods with lower complexity are associated with better maintainability and understandability.

$$numops(M_c) = \sum_{v_i \in M_c} ops(v_i) \tag{22}$$

Where *ops* returns the number of operations performed by method v_i.

The total complexity of a solution is the sum of the individual complexities of all microservice candidates M_c.

$$Complexity = \sum_{M_c \in S} numops(M_c) \tag{23}$$

A lower total complexity indicates a better decomposition.

7.7.2. Non-functional metrics

Network overhead

Microservices communicate over a network with one another, which introduces overhead in terms of latency and bandwidth. In order to keep this overhead low, it is important that method calls between microservices are kept to a minimum, and that the size of the data exchanged (e.g. parameters and return values) is kept small. Using the source code of the monolith application, we can estimate the network overhead of a method call by inspecting the primitive types of the parameters and return values of the methods involved in the call [24].

The heuristic function $h(v_i, v_j)$ estimates the network overhead of a method call from a method v_i to a method v_j, as described by Equation 24. The value for $h(v_i, v_j)$ is calculated by summing the size of the primitive types of the parameters and return values of the method call. The heuristic does not take into account the overhead of the communication protocol (e.g. HTTP headers) or data management overhead (e.g. (de-)serialization).

$$h(v_i, v_j) = \sum_{p \in \mathcal{P}(v_i)} size(p) + \sum_{r \in \mathcal{R}(v_j)} size(r) \tag{24}$$

$\mathcal{P}(v_i)$ returns the set of parameters of method v_i, and $\mathcal{R}(v_j)$ returns the return value(s) of method v_j. The $size$ function returns the size of the primitive type of the given parameter.

The individual network overhead of a microservice candidate M_c can be written as the sum of the network overheads of all method calls between methods in M_c and methods not in M_c.

$$ovh(M_c) = \sum_{v_i \in M_c, v_j \notin M_c} h(v_i, v_j) \tag{25}$$

The total network overhead of a solution is the sum of the individual network overheads of all method calls between microservice candidates M_c and M_c in solution S.

$$Overhead = \sum_{M_c \in S} ovh(M_c) \tag{26}$$

However, this method has severe limitations in a dynamic language such as Ruby, where the types of the parameters and return values are not explicitly declared and may vary at runtime. In some cases, the primitive types can be inferred from the method body, but in general, this is a difficult problem to solve.