# Using Guided Community Detection to Improve Existing Microservice Designs

Patric Genfer[1,2] and Uwe Zdun[1]

[1] Research Group Software Architecture, Faculty of Computer Science, University of Vienna, Vienna, Austria
{patric.genfer|uwe.zdun}@univie.ac.at
[2] UniVie Doctoral School Computer Science DoCS, University of Vienna, Vienna, Austria

**Abstract.** Breaking monolithic applications into microservices is well studied, but the redesign efforts often stop after the initial decomposition. However, microservice architectures evolve, and changing requirements demand ongoing effort to optimize and reduce the interservice communication to provide the best possible performance. Nevertheless, redesigning an existing system is challenging due to established domain or functional boundaries that limit flexibility. To address this issue, we present a novel approach for refining existing microservice architectures to reduce communication overhead while preserving original domain and data access constraints. Our method applies a community detection algorithm, guided by forces derived from domain boundaries, data consistency, and functional separation, to identify optimal service clusters. By running our algorithm with varying input scenarios, we generate a Pareto front of system redesign alternatives, evaluated on architectural metrics. In a case study using a large microservice reference system, our approach reduced interservice calls by 20% while keeping all constraints and up to 50% when partially easing some service boundaries. The approach is easily configurable and adaptable, offering a practical tool for evolving microservice architectures.

**Keywords:** Microservices · Leiden community detection · APIs

## 1 Introduction

One goal of microservice architectures is to split formerly large monolithic applications into smaller, easier-to-handle services, where each service is only responsible for a single aspect of the business domain [1, 2]. Various strategies for identifying the optimal service granularity have been proposed, like defining service boundaries based on domain concepts [3], or on functional decomposition [2, 4]. However, most of these approaches focus only on creating an initial design [3], without adequately addressing the fact that a microservice system is an evolving structure that must adapt to new requirements and features. Also, initially defined domain boundaries can often be subjective and need further adjustment as the problem domain is better understood or new use cases are identified and implemented [5]. These conceptual mismatches can even translate into practical inefficiencies when domain boundaries are not well aligned

with physical ones, leading to high network traffic and poor performance [6]. Nevertheless, these originally defined boundaries play an important role in the microservice design, as they express important domain knowledge necessary to understand the system. Breaking up these restraints to achieve better network performance would ignore all this distilled knowledge. Thus, any system redesign that aims to preserve the characteristics of the original architecture must carefully account for these boundaries, making the design process challenging.

To address this challenge, we present a novel approach for redesigning existing microservice systems to improve service granularity and reduce inter-service network calls. Our method accounts for key factors such as domain boundaries, data access restrictions, and functional partitioning. We apply a graph-based community detection algorithm to model API operations and guide partitioning by incorporating these forces through varying edge weights. By permuting the weight assignments, we generate hundreds of architectural alternatives with different levels of granularity. We assess the resulting solutions based on a selection of architectural metrics and runtime scenarios we collected using end-to-end (E2E) test cases [7]. Due to conflicting metrics, we derive a Pareto front to highlight optimal trade-offs, allowing architects to select the most suitable solution. Demonstrating our approach on a larger-scale microservice reference architecture revealed that even with the most conservative approach, i.e., keeping all existing system constraints in place, we could reduce the overall amount of interservice calls by 20%. Choosing a more progressive redesign, where more APIs are moved from their original domain into new clusters, reduced the network calls even further by up to 50%.

Thanks to its flexible configuration, our approach can be easily applied to other microservice systems and tailored to specific use cases[3] Architects can also add custom guiding forces with minimal effort.

With our research, we aim to answer the following research questions:

**RQ1** *How can we reduce interservice calls in microservices while preserving architectural boundaries?*
Community detection algorithms form well-partitioned graphs but often ignore domain and architectural boundaries. We show how to guide these algorithms to respect such boundaries.

**RQ2** *How can we evaluate the quality of our architectural solutions?*
Our multi-objective approach produces hundreds of design variants. We compute a Pareto front based on selected architectural metrics to identify the most suitable ones.

The paper is structured as follows: Section 2 reviews related work. Section 3 presents the use of community detection in microservice networks and its challenges. It introduces the granularity forces we use to guide the community detection process. Section 4 outlines the metrics for evaluating our architectural

[3] For reproducibility, we offer our study's whole source code and data in a data set published on Zenodo: `https://zenodo.org/records/15833788`.

redesigns. Section 5 presents a case study on a mid-scale reference system. Section 6 discusses results and limitations, followed by threats to validity in Section 7. Section 8 concludes our work and suggests future directions.

## 2 Related Work

While several studies focus on splitting monolith applications into microservices, relatively few investigate how an existing microservice system could be optimized by reorganizing its APIs between the services [3].

One such study is the work of Mendonça Filho and Mendonça [8], which investigates the impact of service granularity on the performance of microservice systems. They use the Service Weaver framework to deploy a microservice architecture on different virtual machines using varying granularity levels. Their study shows that the chosen service granularity significantly impacts the overall system's performance, and choosing the right level of granularity should be an informed decision, as it can have far-reaching effects on the system.

Homay et al. [9] introduced a decomposition method based on service granularity cost analysis to analyze whether decomposing a service provides a timely benefit compared to a larger system. Their approach requires collecting detailed runtime data, which was unavailable in our experimental setup. Gysel et al. developed *Service Cutter*, a tool that operates on different input data like domain models, ER diagrams, and use cases and applies clustering algorithms to generate decomposition models [10]. While they aim to create optimal decomposed services, we focus on reducing network traffic in existing microservice systems.

A study that uses community algorithms for splitting up microservice systems is the work of Rahmanian et al. [11]: They investigate how an existing system could be distributed between cloud and edge resources to avoid communication bottlenecks. Our research follows their approach to applying a community detection algorithm on an existing microservice architecture, but while they distribute complete services between communities, our method redesigns services by reorganizing their APIs.

Gaidels and Kirikova [12] also use the Leiden algorithm but follow a different approach: They form domains by grouping services into larger communities.

Another study that uses community algorithms and social network analysis to identify patterns in microservice communication is the work of Khodabandeh et al. [13]. They use call graphs from a large microservice runtime dataset to categorize services into communities and identify potential bottlenecks in service communication. In contrast to our work, they group services into new clusters, while we group individual APIs into new services.

Regarding the use of a Pareto front to find the best solutions in a multi-objective problem space, our work follows a similar approach to the research of Kinoshita and Kanuka [14]. They define several policies to evaluate the quality of their decomposition model and use reference lines to find optimal Pareto solutions. While their policies and violations align with our concept of granularity forces, the metrics we use for assessing our solutions consider both statically

extracted communication paths and runtime information collected through E2E tests. Also, similar to other research mentioned in this section, they focus more on transforming existing monoliths into microservices, while we concentrate on improving the communication structure of existing services.

## 3 Using Community Detection to Optimize Existing Microservice Designs

### 3.1 Background

Large networks often contain substructures of nodes that are more densely connected with their neighbors than other nodes. Identifying these communities can help uncover communication patterns, detect dependencies, or resource distributions within the network. However, these structures are often implicit and hidden in the overall network layout and, hence, not easily discoverable. Community detection algorithms are a way of identifying such clusters inside large networks by finding an optimal partitioning of a network graph so that there is a high density of connections between the nodes inside a community, compared to a lower connection density between nodes of different groups [15]. A benchmark for measuring the quality of a community partitioning is the *modularity* score, expressed by the following formula [16]:

$$\mathcal{H} = \frac{1}{2m} \sum\nolimits_c \left( e_c - \gamma \frac{K_c^2}{2m} \right) \tag{1}$$

Here, $e_c$ is the number of edges within a community, whereas $K_c$ is the sum of the degrees of all nodes in the given community $c$. These two terms are inversely related: To achieve a high modularity score $H$, we must maximize $e_c$ while keeping $K_c$ small, i.e., for reaching a large $e_c$, our partition should ideally have a large community that contains most of, if not all, edges. However, to minimize $K_c$, our partition should consist of many small communities with only a few nodes. Accordingly, reaching a good modularity score, i.e., a value close to 1, requires both terms to be balanced.

Several heuristic algorithms for optimizing modularity exist, with the Louvain [17] and the Leiden [16] algorithms being two of the most popular. We chose the latter for our research, as it is more efficient and avoids poorly connected communities [16]. The algorithm follows an iterative approach, creating a partition during its first step and refining this initial solution by moving nodes between communities until it reaches a partitioning that can not be optimized further. It then aggregates the communities of the refined partition to build new communities. From there, it reruns the procedure of moving nodes between communities until the solution cannot be further improved [16].

### 3.2 Graph-Based Microservice Modeling

Interestingly, the concept of modularity maps also very well with two major design principles of microservice architectures: *Cohesion* and *Coupling*: The goal

of a good microservice design is to reach highly cohesive functionality within a service, while at the same time having a low coupling, i.e., a low number of connections between services [1, 18].

Accordingly, if we model a microservice system as a directed graph $G = (V, E)$, with $V$ representing all public API operations in the system and $E$ a pairwise set of vertices $\{v_1, v_2\}$ indicating a cross-service network call from API $v_1$ to API $v_2$, we can further interpret all services of the system as a partitioning of communities $C = \{c_1, ..., c_n\}$, where each community $c$ contains all APIs that are part of that service. Figure 1 shows an example of such a graph constructed from a subset of the *train-ticket* microservice benchmark system[4], which we also use later for our case study.

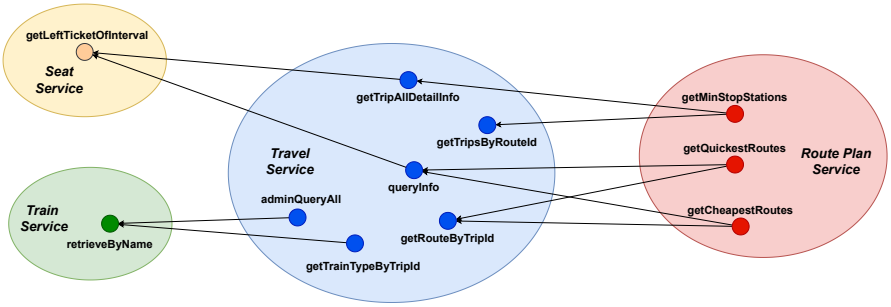


**Fig. 1.** Model of a microservice system. Each service represents a community of API operations. Connections between communities show network calls between APIs.

Since every edge $\{v_i, v_j\}$ stands for an actual network call between two APIs, having many edges can result in performance costs due to network latency, making the system also more vulnerable to network faults [19]. Finding a partitioning where both endpoints of an edge are within the same community would thus convert the network call to an in-memory method invocation, making it faster and more robust. Applying the Leiden algorithm to our example creates a much better partitioning with significantly fewer links between communities and highly connected APIs within the same community (see Figure 2). A microservice system modeled after such a graph would be much less susceptible to network issues, making the communication faster and more secure [20].

However, the graph we initially constructed has a huge drawback. While it correctly reassembles the connections between APIs, it does not model the relations between an API and its service. Consequently, the Leiden algorithm creates a partitioning that eradicates all former service boundaries. See, for instance, the blue API operations in Figure 2: In our initial setup, they were all part of the Travel domain and most likely shared implementation details and data structures, but now they are scattered throughout the system. The same happened

[4] `https://github.com/FudanSELab/train-ticket`

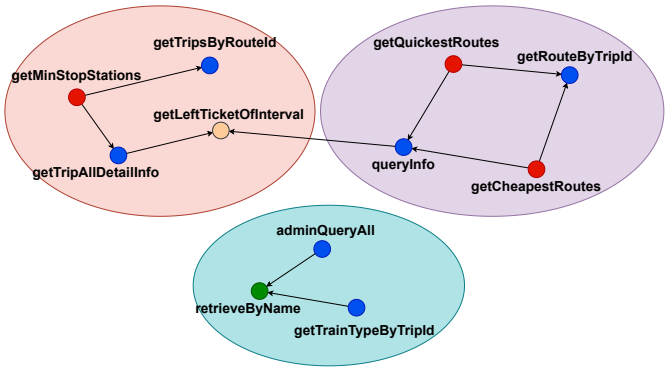


**Fig. 2.** Applying the Leiden community detection algorithm on our microservice system creates an optimal partition regarding cohesion and coupling—however, it also destroys former domain and implementation boundaries, making the solution very impractical.

with the *Route Plan Service* APIs, which are now split between two services. Reshaping our service architecture destroyed all the information and knowledge expressed by the relationship between APIs and their services.

### 3.3 Guiding Community Detection with Granularity Forces

To advise the Leiden algorithm to consider an API's semantic relations to its service, we must enrich our graph with this additional knowledge. We identified four key factors that form these relations: *Domain Boundaries*, *Data Access Constraints*, *Functionality-Based Partitioning*, and *Interservice Communication*. We call these factors Granularity forces, as they directly affect a service's granularity.

1. **Domain Boundaries** represent an essential force as they often define the conceptual perimeter of a service [19] and its APIs. To model this force, we add a semantic node representing the domain concept to each service and connect it to its APIs. We further add the weight $w_d$ to represent the new relation's strength. Since our reference architecture was initially decomposed by domains, each service maps exactly to one domain node.
2. To enforce **Data Access Constraints**, we add another semantic node for each data store, linking it to APIs that read or write to it. The weight $w_p$ ($p$ersistence) indicates the strength of the connection.
3. In addition to domain concepts, services can be structured around specific use cases. To capture this **Functionality-Based Partitioning**, we create one node per test case and connect it to each API called during E2E tests. We assign the weight $w_u$ ($u$se case) to express the relation between an API and its use case. We used runtime data from E2E tests [7] to connect APIs to their use cases.
4. For **Interservice Communication**, which the graph already represents, we enrich the existing edges by adding a communication weight $w_c$.

Incorporating these forces into our existing model leads to an enriched graph, depicted in Figure 3, where we added additional nodes for domain concepts and data storage. We omitted use-case-related forces for now to improve clarity, but considered them in our case study later.

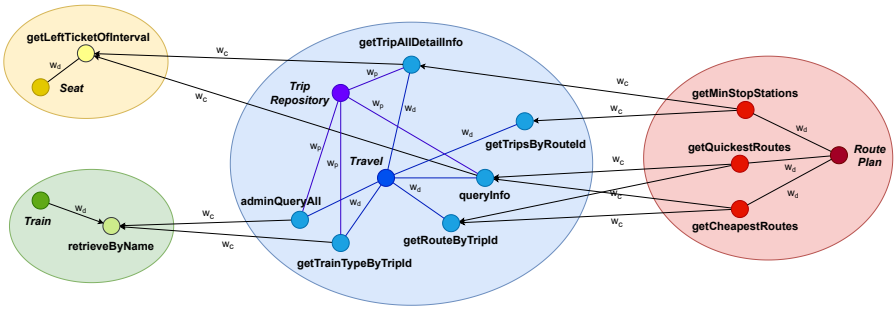


**Fig. 3.** All services contain additional nodes to increase the cohesion between their APIs. While all three services have an additional domain concept node, the Travel Service also has a data repository node linked to the APIs that need database access.

The Leiden algorithm characterizes edges with higher weights as stronger relations and places them more likely inside the same community[5]. Accordingly, adjusting each force weight allows us to steer the community detection. However, manually assigning weights can be cumbersome, especially for larger graphs. Instead, we implemented a process to generate permutated weight assignments automatically and use them as input vectors for the Leiden algorithm to create a large set of different partitionings. To further automate the process, we defined a set of architectural metrics to validate the generated solutions.

## 4 Metrics

To assess the quality and suitability of our generated solution, we defined a set of architectural metrics derived from our microservice graph and the granularity forces we introduced before. Our first metric, the ***N**etwork**C**all**R**atio* calculates the ratio of all API invocations over the network compared to the total amount of API calls.

$$NCR = \frac{\left|\left\{ \{v_i, v_j\} \in E : cluster(v_i) \neq cluster(v_j) \right\}\right|}{\left|\left\{ \{v_i, v_j\} \in E \right\}\right|} \tag{2}$$

In the unchanged setup, where all APIs are placed in their initial services and communication happens only between APIs of different services, this value is exactly 1. However, this value should be lower in a more optimized community

[5] `https://igraph.org/r/doc/cluster_leiden.html`

partitioning, indicating that formerly separated APIs were moved into the same service to reduce network traffic. Reaching a low value for this metric is one of the main goals of our approach.

The ***D**omain**M**ovement**R**atio* metric describes how many APIs moved out of their initial domain context due to our system redesign. For this, we compare the number of edges connecting APIs with their domain node with the number of domain edges connecting APIs in different clusters. Note that $E^d$ contains all semantic edges between an API operation and a domain node.

$$DMR = \frac{\left|\left\{ \{v_i, v_j\} \in E^d : cluster(v_i) \neq cluster(v_j) \right\}\right|}{\left|\left\{ \{v_i, v_j\} \in E^d \right\}\right|} \tag{3}$$

Since in our original graph, all APIs are grouped within their original domain context, this value will be 0. However, after the APIs are reorganized into new communities, an existing API may be moved out of its previous domain and into a new domain. Still, our goal is to keep this value as low as possible.

Analog to *DMR*, we define the ***D**ata**A**ccess**V**iolation**R**atio* as a metric that calculates the number of cross-service database calls compared to the total number of database calls (with $E^p$ containing all edges between an API and a persistence storage). Again, this number should be close to zero, as sharing data stores between services should best be avoided.

$$DAVR = \frac{\left|\left\{ \{v_i, v_j\} \in E^p : cluster(v_i) \neq cluster(v_j) \right\}\right|}{\left|\left\{ \{v_i, v_j\} \in E^p \right\}\right|} \tag{4}$$

Figure 4 gives an illustration of these three metrics. After reorganizing our small microservice setup introduced earlier with the Leiden algorithm, we see that the new partitioning significantly reduced the inter-service calls (the thick directed arrows) from ten to four invocations. However, it also moved three APIs into a new service, breaking former domain boundaries (the blue dotted lines), resulting in a worse *DMR* score. One of these APIs is now even dislocated from its local data store, resulting in a higher *DAVR* value.

Our final metric, the ***T**estcase**N**etworkcall**R**atio*, combines runtime data from E2E tests with the static API structure extracted from source code. Figure 5 illustrates how the metric is composed. We identify all APIs of each test case, including indirect API-to-API calls. By matching these with our static communication model, we reconstruct the full API call paths (colored arrows in the diagram) and count the network calls required to complete each test case. We then computed an average network call count across all tests, compared with the average number of API calls per test. A lower value indicates that our community detection algorithm has grouped the test-case-related APIs into a dedicated service, making the execution more efficient.

Finding a single solution that optimizes all metrics is unlikely, as they often conflict. For example, minimizing network calls (NCR) may move APIs into other services, leading to domain boundary violations and increased DMR. However, a naive fix like merging all APIs into one service would result in a low modularity score and, therefore, already be ruled out by the Leiden algorithm. Instead, our

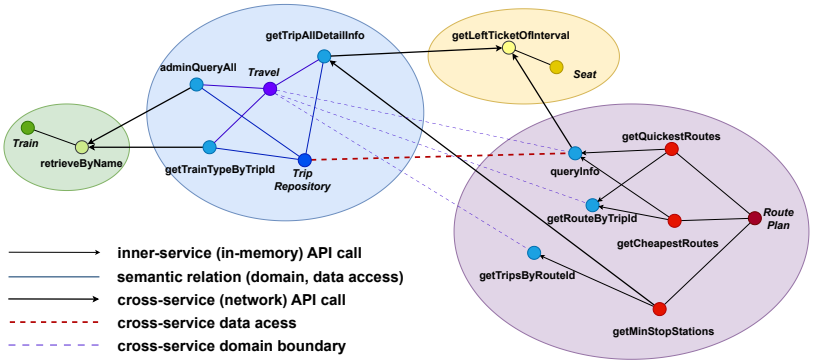


**Fig. 4.** Example of a reorganized microservice system. While the number of network connections could be reduced, the new partitioning also dislocated some APIs out of their domain boundaries and introduced a cross-service data-access call.

approach constructs a Pareto front of solutions where no metric can be improved without worsening another. Architects can then select the most appropriate design based on their priorities and the constraints of their use case.

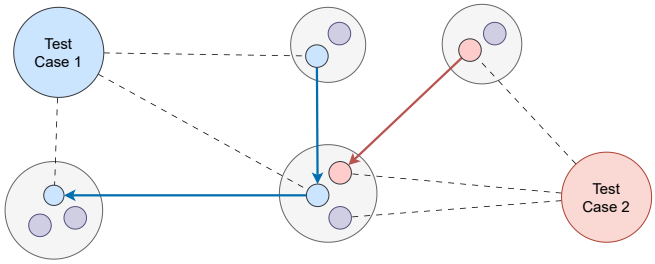


**Fig. 5.** The *TNR* metric expresses how many API network calls (the colored thick lines) are necessary to successfully process a given test task. Merging several of those APIs (the blue or red dots) into dedicated clusters could improve the execution time per test case, as fewer network calls would be necessary.

## 5 Case Study

### 5.1 Architectural Model Reconstruction

We assessed our community guiding process on the Train Ticket microservice benchmark application[6], a mid-size system with more than 30 microservices and

[6] `https://github.com/FudanSELab/train-ticket`

large enough to represent real-world applications [21]. Since isolated nodes do not impact the modularity score of a graph [22], we considered only services with at least one API operation that is either the source or the target of a cross-network call. We also ignored calls from incoming clients, such as from the user interface. We followed a three-stage transformation process to generate the resulting community graphs. In the first step, we used source code detectors [23], a lightweight Python-based parser technique, to reconstruct a component-based architectural model from Train Ticket's code artifacts (see Figure 6). In the next step, we converted the component model into an igraph-compatible network structure using the corresponding Python interface[7]. This transformation was necessary to use the graph algorithms available for this package, including a Python implementation of the Leiden algorithm[8], which we used in the last step to convert our graph in various community partitions by running the algorithm with different weights assignments. We also varied the maximum size of communities generated by the algorithm between two and eight APIs per cluster to see how different service granularities would affect the solutions.

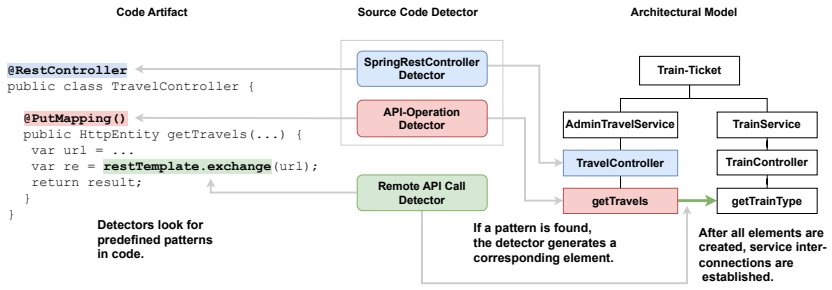


**Fig. 6.** We used *Source Code Detectors* to parse the underlying microservice code base and reconstruct an architectural model of the system. Those detectors are lightweight parsers that look only for specific patterns in code and, if found, create the corresponding model elements [23].

### 5.2 Detecting Communities in a Mid-size Microservice Application

For the case study, we picked 39 backend services we considered relevant for business and domain operations and ignored infrastructure services and those related to user interaction, resulting in 133 cross-service network API operation calls. Although our detectors could also track asynchronous connections, most of these calls were synchronous HTTP REST API invocations.

For later comparison, we created a graph representing the original partitioning by assigning each node to its initial service cluster and running the Leiden

[7] `https://python.igraph.org/en/stable`
[8] `https://leidenalg.readthedocs.io/en/stable/intro.html`

algorithm with zero iterations (Figure 7). Larger dots represent domain concepts, repositories, or use cases, while smaller dots show API operations. API operation labels have been omitted for clarity.

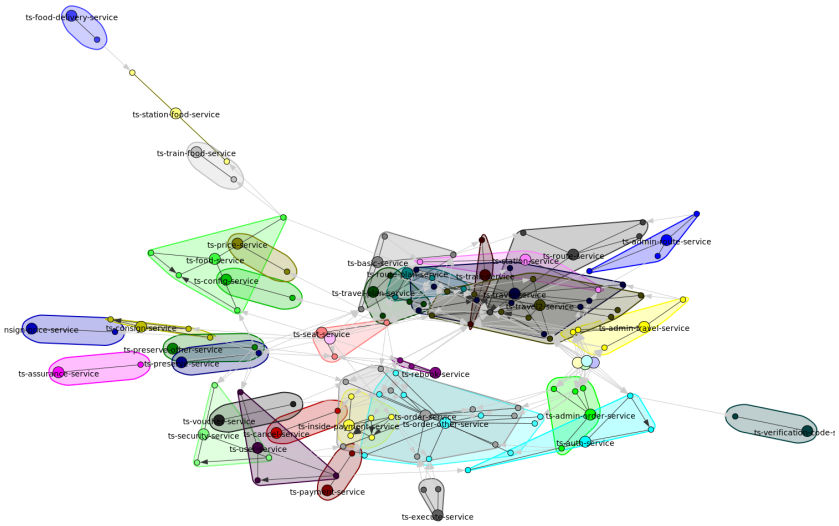


**Fig. 7.** The original microservice architecture drawn as a community graph. Each cluster represents a service, with the smaller dots inside each cluster being the API operations belonging to that service. Our graph shows only the APIs that are part of an invocation chain with more than one network call.

We generated permuting input assignments with weight values between 0 and 5 and, for each, ran the Leiden algorithm with infinite iterations and varying community size limits (2–10). Due to inherent randomness [16], results may slightly vary between runs. Our setup produced 2048 solutions, each based on a unique input vector and community size. To reduce the solution space, we computed a Pareto front[9], which aims to minimize all the metrics described in Section 4). The calculated front consists of 108 solutions, each representing an optimal result that cannot be further improved.

Table 1 highlights selected solutions, each excelling in a specific metric. While the Pareto front was based on our metrics *NCR*, *DMR*, *DAVR*, and *TNR*, we also list the number of services and average APIs per service for comparison. For reference, the original configuration is included, which, as expected, scores well in *DMR* and *DAVR* due to preserving the original service boundaries but poorly in *NCR* and *TNR* due to many small services. We also show a *Brute-Force* solution (with input vector $w_c : 1, w_d : 0, w_p : 0, w_u : 0$) that optimizes for connectivity and ignores all other semantic links, causing efficient networking

[9] By using the paretoset Python package `https://pypi.org/project/paretoset/`

performance but very poor domain and data cohesion. The remaining entries are Pareto-optimal, offering the best trade-offs per focus. Among them, *Solution 695* stands out as a balanced alternative: While it does not achieve the best result in any single metric, it consistently performs well across all of them, making it the least-worst option overall.

**Table 1.** Selected redesigns and their performance based on architectural metrics (lower is better). The brute-force partition, generated by the Leiden algorithm without considering any semantic nodes, is network-efficient but ignores domain and data boundaries. The other, Pareto-efficient solutions offer a better balance, improving network efficiency while preserving key constraints.

| Solution | # services | **NCR** | **DMR** | **DAVR** | **TNR** | APIs/ service |
|---|---|---|---|---|---|---|
| **original setup** | 44 | 1.0 | 0.0 | 0.0 | 1.0 | 2.83 |
| **brute force** | 65 | 0.17 | 1.0 | 1.0 | 0.02 | 6.19 |
| **391** | 11 | **0.33** | 0.53 | 0.55 | 0.23 | 9 |
| **1031** | 12 | 0.78 | **0.06** | 0.05 | 0.95 | 8.25 |
| **22** | 11 | 0.58 | 0.34 | **0.0** | 0.65 | 9 |
| **903** | 10 | 0.37 | 0.48 | 0.40 | **0.22** | 9.90 |
| **695** | 10 | 0.56 | 0.31 | 0.10 | 0.56 | 10 |

Another observation is that although we used varying community sizes, the best results always have relatively large services with nine or more APIs per service, which is already considered an upper limit [18], but may well depend on the actual implementation complexity of each API, which we did not measure.

Which solution an architect may finally pick depends on their objectives. If improving the network throughput is the primary goal, *Solution 391* provides the most benefits but requires a more substantial system redesign, as it dislocates many APIs from their domain and data store. *Solution 903* is similar, but reduces the network connections for the selected use cases even further. *Solution 1031* is the partitioning that violates the least boundaries, but still improves the network performance by roughly 20%. We would thus consider it a conservative solution that still provides good benefits. However, assuming the domain- and data constraints are lessened slightly further, *Solution 695* (Figure 8) may give an even better tradeoff, as it reduces the network calls by almost half while violating only a few domain and data boundaries. But still, an architect or domain expert must determine whether these violations are acceptable or need additional measures, like splitting up the data store.

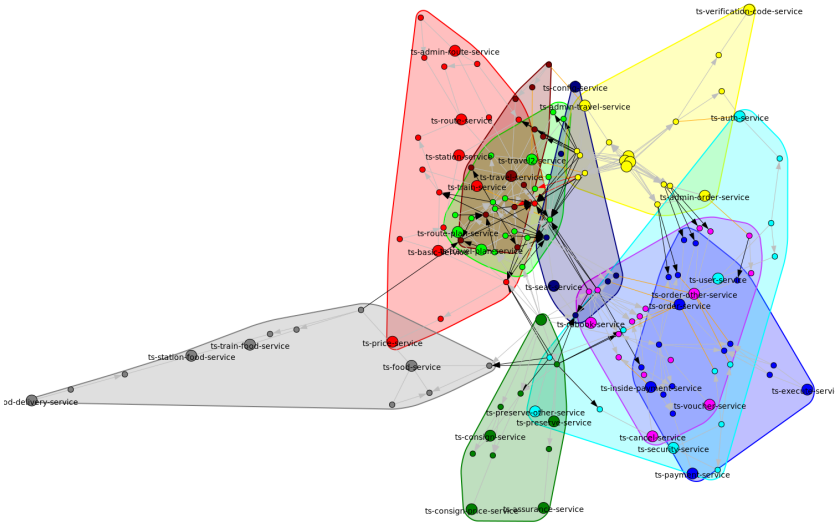


**Fig. 8.** Solution 695 is a good tradeoff between reducing network communication and keeping domain and data boundaries. Orange arrows indicate API operations moved away from their original domain, while red arrows indicate APIs that make cross-service database calls.

## 6 Discussion

Addressing our first research question **RQ1**, we could show that providing community detection algorithms like *Leiden* with additional guidance creates a better decoupled service graph while still keeping conceptual boundaries.

We thus identified four essential forces that affect a service's granularity and steer the community detection process: Domain boundaries, access to local data storage, use-case or feature-based functionality, and interconnectivity. Each of these forces plays a vital role in deciding which service an API should belong to. However, other forces may also affect service granularity. For instance, according to Conway's law, team organization could be a heavy driving force for service boundaries. Tracking these forces could be part of further research.

To assess the quality of our approach (**RQ2**), we defined four architectural metrics to track and evaluate the changes our process applies to the original architecture. Since these metrics present multi-objective goals, we calculated a Pareto front that contains only solutions that cannot be further improved without degrading at least one objective. Depending on how much domain or data boundary violations are tolerable, our solutions may reduce network connections between services by up to 50% or even more. Nevertheless, implementing such a reorganized microservice system would require architectural decisions, e.g., handling API calls to databases in another service or sharing implementation between APIs that are now in different services. While we can point architects

to the locations of the violations, they have to adapt the system manually. Providing additional guidance here would be an interesting extension of our work.

## 7 Threats to Validity

**Internal Validity**: If system chosen for our study does not represent the broader population of APIs and microservices, our results may be biased. To counteract this, we chose a reference system that conforms to common microservice best practices and is well-established among researchers. Also, other factors than the ones we mentioned could influence the community detection process.
**Construct Validity**: Regarding construct validity, we consider whether our model accurately represents the microservice system and if our forces properly generalize the factors shaping its architecture. Concerning the first case, our decomposition technology is well-established and has been applied successfully in several studies [23]. We also incorporated manual verification steps and traced the generated elements back to their code base. The identified granularity forces are based on extensive literature research and our experience with previous microservice systems. Still, they cover only a subset of influencing factors, and many others may exist.
**External Validity**: We chose a considerably large-scale and widely accepted reference system to show that our method would also be suitable for larger real-world applications. Thanks to the possibility of choosing individual weight assignments, our community detection analysis can easily be adapted to specific scenarios. Architects may adjust certain weights to their needs.

## 8 Conclusions and Future Directions

This work introduces a new approach using guided community detection to reduce interservice connections in existing microservice architectures. While applying these algorithms without modifications may reduce cross-service calls, they often create communities that do not align well with actual service boundaries, resulting in API operations scattered across various services. To address this, we identified granularity forces influencing the relationship between an API and its service. We used them in an automated approach to guide the community detection process to create more accurate partitionings. These forces include architectural concepts like domain boundaries, data store access, and runtime information from test cases. While we provide the best-found solution for every force through calculating a Pareto front, architects can fine-tune our approach by adjusting the weights individually. Our case study showed that our approach leads to partitionings with reduced interservice communication while respecting service boundaries and constraints, providing practical guidance for designing systems that can perform better, are less complex, and are easier to maintain due to less coupling. This research is still in its early stages. In our upcoming studies, we plan to incorporate other aspects, like a service's implementation complexity or the frequency of API calls, into our model to improve the guidance provided by our algorithm.

*Acknowledgments:* This work was supported by: FWF (Austrian Science Fund) projects API-ACE: I 4268 and IAC$^2$: I 4731-N.

## Bibliography

[1] T. Engel, M. Langermeier, B. Bauer, and A. Hofmann, "Evaluation of microservice architectures: A metric and tool-based approach," in *Information Systems in the Big Data Era: CAiSE Forum 2018, Tallinn, Estonia, June 11-15, 2018, Proceedings 30*. Springer, 2018, pp. 74–89.

[2] M. Ahmadvand and A. Ibrahim, "Requirements reconciliation for scalable and secure microservice (de) composition," in *2016 IEEE 24th International Requirements Engineering Conference Workshops (REW)*. IEEE, 2016, pp. 68–73.

[3] F. H. Vera-Rivera, C. Gaona, and H. Astudillo, "Defining and measuring microservice granularity—a literature overview," *PeerJ Computer Science*, 2021.

[4] D. Bajaj, A. Goel, and S. C. Gupta, "Greenmicro: identifying microservices from use cases in greenfield development," *IEEE Access*, pp. 67 008–67 018, 2022.

[5] N. Ford, M. Richards, P. Sadalage, and Z. Dehghani, *Software Architecture: The Hard Parts*. " O'Reilly Media, Inc.", 2021.

[6] S. Hassan, R. Bahsoon, and R. Kazman, "Microservice transition and its granularity problem: A systematic mapping study," *Software: Practice and Experience*, vol. 50, no. 9, pp. 1651–1681, 2020.

[7] A. S. Abdelfattah, T. Cerny, J. Y. Salazar, A. Lehman, J. Hunter, A. Bickham, and D. Taibi, "End-to-end test coverage metrics in microservice systems: An automated approach," in *European Conference on Service-Oriented and Cloud Computing*. Springer, 2023, pp. 35–51.

[8] R. C. Mendonça Filho and N. C. Mendonça, "Performance impact of microservice granularity decisions: An empirical evaluation using the service weaver framework," in *European Conference on Software Architecture*. Springer, 2024.

[9] A. Homay, A. Zoitl, M. de Sousa, M. Wollschlaeger, and C. Chrysoulas, "Granularity cost analysis for function block as a service," in *2019 IEEE 17th international conference on industrial informatics (INDIN)*, vol. 1. IEEE, 2019, pp. 1199–1204.

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

[11] A. Rahmanian, A. Ali-Eldin, B. Skubic, and E. Elmroth, "Microsplit: Efficient splitting of microservices on edge clouds," in *2022 IEEE/ACM 7th Symposium on Edge Computing (SEC)*. IEEE, 2022, pp. 252–264.

[12] E. Gaidels and M. Kirikova, "Service dependency graph analysis in microservice architecture," in *Perspectives in Business Informatics Research: 19th International Conference on Business Informatics Research, BIR 2020, Vienna, Austria, September 21–23, 2020, Proceedings 19*. Springer, 2020, pp. 128–139.

[13] G. Khodabandeh, A. Ezaz, and N. Ezzati-Jivan, "Network analysis of microservices: A case study on alibaba production clusters," in *Companion of the 15th ACM/SPEC International Conference on Performance Engineering*, 2024.

[14] T. Kinoshita and H. Kanuka, "Enhancing automated microservice decomposition via multi-objective optimization," *IEEE Access*, 2024.

[15] M. E. Newman and M. Girvan, "Finding and evaluating community structure in networks," *Physical review E*, vol. 69, no. 2, p. 026113, 2004.

[16] V. Traag, L. Waltman, and N. Van Eck, "From louvain to leiden: guaranteeing well-connected communities. sci. rep. 9, 5233," 2019.

[17] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of communities in large networks," *Journal of statistical mechanics: theory and experiment*, vol. 2008, no. 10, p. P10008, 2008.

[18] O. Al-Debagy and P. Martinek, "A metrics framework for evaluating microservices architecture designs," *Journal of Web Engineering*, vol. 19, pp. 341–370, 2020.

[19] S. Newman, *Building microservices: designing fine-grained systems*. " O'Reilly Media, Inc.", 2021.

[20] D. Shadija, M. Rezai, and R. Hill, "Microservices: granularity vs. performance," in *Companion Proceedings of the10th international conference on utility and cloud computing*, 2017, pp. 215–220.

[21] X. Zhou, X. Peng, T. Xie, J. Sun, C. Xu, C. Ji, and W. Zhao, "Benchmarking microservice systems for software engineering research," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 323–324. [Online]. Available: https://doi.org/10.1145/3183440.3194991

[22] U. Brandes, D. Delling, M. Gaertler, R. Görke, M. Hoefer, Z. Nikoloski, and D. Wagner, "On modularity-np-completeness and beyond," *ITI Wagner, Faculty of Informatics, Universität Karlsruhe (TH), Tech. Rep*, vol. 19, p. 2006, 2006.

[23] P. Genfer and U. Zdun, "Exploring architectural evolution in microservice systems using repository mining techniques and static code analysis," in *European Conference on Software Architecture*. Springer, 2024, pp. 157–173.