Pangaea: Semi-Automated Monolith Decomposition into Microservices

Simone Staffa, Giovanni Quattrocchi,
Alessandro Margara, and Gianpaolo Cugola

Politecnico di Milano
`name.surname@mail.polimi.it`

Abstract. As microservices become the reference architecture for many practitioners, decomposing an application into microservices remain a challenge. This paper tackles the problem with *Pangaea*, a semi-automatic tool to decompose a software system into microservices. *Pangaea* (i) takes in input a high-level model of the system; (ii) formulates decomposition as an optimization problem, and (iii) outputs a proposed placement of functionalities and data onto microservices, using a visual representation that helps reasoning on the overall architecture. *Pangaea* evaluates design concerns, communication overheads, data management requirements, opportunities and costs of data replication. Our evaluation on a real-world application shows that *Pangaea* consistently delivers more efficient solutions than simple heuristics and state-of-the-art approaches, and provides useful insights to developers.

Keywords: microservices architectures, service decomposition, service modeling, software architectures

1 Introduction and Motivations

The increasing need to evolve software systems quickly and efficiently made many IT practitioners migrate from monolithic to microservices architectures. Microservices architectures define an application as a composition of independent units. Microservices contain a subset of logically-related application functionalities, and are developed, deployed, and maintained independently from each others. Microservice can be developed using a different technology stacks, they run as independent processes that only interact through network protocols such as HTTP or MQTT, they can be scaled independently, and faults do not make the whole system unresponsive, since there is no single point-of-failure. A key challenge to embrace a microservices approach is how to decompose an application into microservices. Indeed, the adoption of microservices architectures encompasses both technical and managerial concerns, which should be carefully considered in the decomposition process. In general, we may classify the desired charachteristic of a successful decomposition as organization, communication, and data management aspects.

Organization. Microservices are organized around business capabilities: the decomposition needs to produce microservices that are highly cohesive and include all the data and processing components to implement a given capability.

Communication. Microservices are developed as independent executables that communicate using remote procedure calls or asynchronous propagation of messages. Frequent communication across microservices may increase the overall response time of the application: the decomposition needs to produce microservices that are loosely-coupled.

Data management. Microservices decentralize data management by design. Each microservice has its local, partial view of the application domain. Data integrity in the presence of concurrent operations and replication is enforced at application level, and may require coordination protocols that are complex, introduce coupling, and may degrade performance. An effective decomposition should be aware of integrity requirements and avoid costly coordination by colocating related data elements within the same microservice.

In summary, a decomposition always represents a compromise between heterogeneous and conflicting forces. Without any tool to support their reasoning, developers may incorrectly evaluate the possible alternatives, leading to inaccurate decompositions that affect development, operations, and maintenance costs. Given the complexity of this problem, some approaches were presented in the literature to assist engineers in the decomposition process [5, 6]. They range from theoretical frameworks that provide principles and guidelines [1, 3] to completely automated tools [2,4]. Manual tools still require considerable effort from developers. Automated tools are often limited to specific application types, and generate decompositions that may be inadequate to the developers actual needs. Moving from these premises, this paper introduces *Pangaea* (Sec. 2), a semi-automatic tool to decompose a monolith into microservices. *Pangaea* takes in input a high-level model of the application. It formulates an optimization problem that evaluates design concerns (coupling and cohesion), communication overhead, data management requirements, opportunities and costs of data replication, and searches for the optimal placement of data and operations across a set of microservices. Developers can prioritize certain requirements over the others through a set of parameters in the model. Our evaluation on a real-world application (Sec. 3) shows the effectiveness of *Pangaea* compared to simple heuristics, a manual decomposition, and a state-of-the-art decomposition approach.

2 *Pangaea*

This section presents *Pangaea* in details. Fig. 1 overviews its workflow, where developers provide (i) a *system model*, which defines the data entities and operations that build the application, together with their characteristics and mutual relations; (ii) a set of *input parameters* that configure the tool and steer the decomposition process based on user preferences. Given these inputs *Pangaea* works in three steps: (1) a *parser* translates the system model into an optimization problem; (2) a *solver* outputs a solution to the problem: a possible allocation

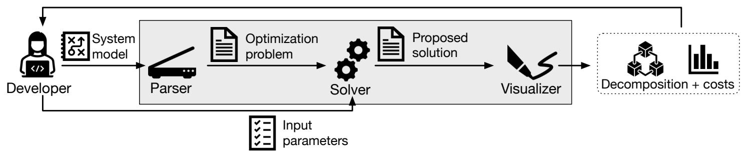

Fig. 1: *Pangaea*: overview of the workflow.

of data entities and operations onto microservices; (3) a *visualizer* produces a visual representation of the proposed decomposition together with a detailed analysis of the costs it incurs. Developers evaluate the decomposition and decide if accepting it or refining the system model and input parameters.

2.1 System model

Pangaea builds on an expressive yet easy-to-use modeling framework. Developers model an application in terms of *data entities* and *operations*, both characterized by a set of attributes. They specify data entities and operations as annotations in YAML, using the `@Entity` and `@Operation` tags, respectively. Tags can be placed in the source code of the application, as comments next to the definition of the data elements and functionalities they model, or they can be placed in a single or multiple dedicated files.

Data entities. Data entities are basic elements of data that *Pangaea* 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 framework to their needs. For instance, in a relational data model, a data entity can be used to model a single table: *Pangaea* will treat the table as an unbreakable unit and map it to microservices accordingly. Developers may also decide to model multiple related tables as a single data entity or to split a table into multiple data entities. In the first case, *Pangaea* will not distinguish individual tables and will consider them as a whole. In the second case, *Pangaea* will have the opportunity to assign the various parts of the table to different microservices. A data entity e is characterized by the following properties.

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

implementation: an optional string that developers can use to map e to concrete elements in the application (for instance, the database tables e refers to).

relations: a list of the other data entities e depends on. The use of relations makes *Pangaea* aware of semantic connections between data entities, which it may exploit to increase cohesion and reduce coupling. Developers may also specify the strength of each relation, which can be either *strong* or *weak*. For instance, in the case of relational tables, developers may model foreign key constraints between tables as strong relations.

replication overhead: a number indicating the expected overhead of replicating e within multiple microservices. Indeed, replication may involve a coordination overhead to keep replicas consistent, which depends on the desired level of consistency and the frequency of updates.

Operations. Operations represent units of execution, which are candidate to become logic functionalities exposed by microservices. Each operation accesses data entities and is associated to a single microservice. In *Pangaea*, an operation o is characterized by the following properties.

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

entities: the list of data entities accessed by o. For each data entity, developers can specify if the access is *read-only* or *read-write*. *Pangaea* 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 (greater for read-write access and lower for read-only access) and it may increase coupling, as it indicates that a microservice is requesting data with remote invocations to another microservice rather than accessing it locally.

frequency: a number that indicates how frequently o is invoked. In the decomposition process, *Pangaea* will prioritize reducing the costs associated to operations that are invoked more frequently.

integrity: represents the requirements of o in terms of data integrity. It can be either *low* or *high*. For instance, integrity may include isolation policies to coordinate concurrent invocations, such that developers may distinguish between a high level of isolation (stronger, but more expensive to enforce, such as serializable isolation) and a low level of isolation (weaker, but less expensive, such as monotonic atomic view isolation). Enforcing integrity requirements is more expensive in distributed settings, that is, when o needs to access remote data elements. Accordingly, *Pangaea* will favor decomposition choices that maximize local data access for operations that require (high) integrity.

forced entities: list of data entities that need to be located on the same microservice as o. For instance, developers may enforce a single microservice being responsible for updating a data entity e. Also, developers can use forced entities to encode application-specific concerns such as access control policies.

2.2 Optimization problem

Pangaea formulates an optimization problem that aims to find an allocation of data entities and operations onto a set of microservices that minimizes three costs: (i) *Coupling cost* is the (design) cost for placing non-related data entities in the same microservice, which decreases cohesion. (ii) *Communication cost* is the overhead of communication across microservices due to dependencies between operations and data entities that are not placed in the same microservice. (iii) *Replication cost* is the overhead of replication. While replication may reduce the communication cost, keeping replicated data entities consistent requires additional coordination and it may result in increased response times. We denote E the set of data entities, O the set of operations, and M a set of microservices. Two decision binary variables x and y encode the placement of operations and data entities onto microservices, respectively:

$$x_{o \in O, m \in M} = 1 \text{ iff } o \text{ is placed on } m, \qquad y_{e \in E, m \in M} = 1 \text{ iff } e \text{ is placed on } m$$

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

number of microservices: is the cardinality of M and indicates the maximum number of microservices that the decomposition can use. The solver may use only a subset of microservices, resulting in a decomposition into fewer microservices.

organization-communication ratio: an integer number α that indicates the importance developers attribute to organization concerns (coupling cost) over communication concerns (communication and replication costs), on a scale between 0 and 100 (default: 50).

relation weight: an integer number w_{rel} used to weight the cost of placing on the same microservice two *unrelated* data entities in comparison with the same cost for *weakly* related entities (default: 2).

access weight: an integer number w_{acc} that represents the overhead of *read-write* access with respect to *read-only* access to data entities (default: 2).

integrity weight: an integer number w_{int} that represents the overhead of enforcing *high* integrity with respect to *low* integrity for operations (default: 2).

Coupling cost. The coupling cost is the cost associated to placing two unrelated entities on the same microservice, defined for each microservice $m \in M$ as:

$$CPcost_m = \sum_{e1 \in E, e2 \in E} y_{e1,m} \cdot y_{e2,m} \cdot CP_{e1,e2}$$

where $y_{e1,m} \cdot y_{e2,m}$ is 1 if both e_1 and e_2 are placed on m, and 0 otherwise, while $CP_{e1,e2}$ is a measure of the dependencies between e_1 and e_2. A strong dependency leads to a small coupling cost: coupling the two entities in the same microservice is acceptable as it does not decrease the cohesion of the microservice. We compute CP_{e_1,e_2} based on the *relation* attributes expressed in the system model: it is 0 if e_1 and e_2 are the same entity or if there is a strong relation between them, it is 1 if there is a weak relation, and it equals w_{rel} if they are unrelated.

Communication cost. The communication cost measures the overhead of placing an operation o and a data entity e accessed by o on two different microservices, defined for each microservice $m \in M$ as:

$$COMMcost_m = \sum_{o \in O, e \in E} x_{o,m} \cdot (1 - y_{e,m}) \cdot COMM_{o,e}$$

where $x_{o,m} \cdot (1 - y_{e,m})$ is 1 if o is placed on m but e is not, and 0 otherwise, while $COMM_{o,e}$ evaluates the weight of communication between e and o, and is defined as: $COMM_{o,e} = acc_{o,e} \cdot int_o \cdot freq_o$ where $acc_{o,e}$ is the *access cost*, which is 0 if o does not access e, 1 if o accesses e in read-only mode and w_{acc} if o accesses e in read-write mode; int_o is the *integrity cost*, which is 1 if o has weak integrity requirements and w_{int} if o has strong integrity requirements; finally, $freq_o$ is the frequency of o, as indicated by the developers in the system model.

Replication cost. The replication cost is the overhead of replication, defined for each data entity $e \in E$ as:

$$REPLcost_e = \sum_{m \in M} y_{e,m} \cdot REPL_e$$

where the summation indicates that the cost for replicating a data entity is proportional to the number of replicas (the number of microservices that holds a replica of e), while $REPL_e$ is the replication overhead, as indicated by the developers in the system model.

Objective function. The goal is to minimize the total cost, expressed as the sum of coupling, communication, and replication costs, weighted by the ratio α:

$$TOTcost = \alpha \cdot \sum_{m \in M} CPcost_m + (100 - \alpha) \cdot (\sum_{m \in M} COMMcost_m + \sum_{e \in E} REPLcost_e)$$

under the constraints that an operation is assigned to a single microservice, while an entity may be replicated to multiple microservices:

$$\forall_{o \in O} \sum_{m \in M} x_{o,m} = 1, \quad \forall_{e \in E} \sum_{m \in M} y_{e,m} \geq 1$$

Notice that the above problem is not linear (the coupling cost requires multiplying y by y, and the communication cost requires multiplying x by y). To linearize the product of any two binary variables a, b, we introduce a new binary variable $c = a \cdot b$. We observe that $c \neq 0 \iff a = b = 1$, which can be expressed with the following linear constraints: $c \leq a$, $c \leq b$, $c \leq a + b - 1$.

2.3 Presenting the output

We conceive *Pangaea* as a decision support tool that should help developers reasoning on the system and evaluate the consequences of a given decomposition choice in terms of design and operational costs. Accordingly, we built a visualizer component that offers a graphical representation of the proposed decomposition as a dynamic Web page. The visualizer shows entities and operations associated to microservices, as well as remote invocations across microservices, labeled with their communication cost. In addition, *Pangaea* outputs a detailed report with the individual contrinutions to the total cost of the proposed solution. Developers may use the report to evaluate the trade-offs of the solution and to refine their system model or choice of input parameters.

3 Evaluation

We evaluated *Pangaea* on a real-world case study provided by Tutored (`https://www.tutored.me/`), a tech startup that works in the education sector. The case study consists of a REST API developed with Node.js, Express, and Typescript. Once modelled in *Pangaea*, it includes 45 data entities and 71 operations. The evaluation aims to answer the following research questions: **(RQ1)** How do the decompositions proposed by *Pangaea* compare with alternative ones? **(RQ2)** How do practitioners benefit from the usage of *Pangaea*?

Approach	**Cost (comm)**	**Diff**
Manual	140.6k (77.65k)	+206%
Pangaea (4)	52.4k (5.6k)	+14%
Pangaea (5)	45.5k (7k)	-
Monolith	193.15k (0)	+320%
Distributed	73.75k (71.5k)	+60%
SC GN (4)	160k (0)	+248%
SC GN (5)	152.1k (0)	+231%
SC Leung	217.2k (0)	+373%
SC CW	89.8k (27k)	+95%

Fig. 2: Costs with $\alpha = 50$

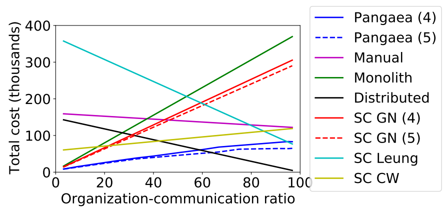

Fig. 3: Total costs by varying α.

To answer RQ1, we compared *Pangaea* with four alternative approaches: (i) a manual decomposition produced at Tutored; (ii) ServiceCutter (SC), a state-of-the-art tool for microservices decomposition that uses a graph clustering approach; (iii) a fully distributed solution (each entity is placed on a separate microservice); (iv) the original monolith. The manual solution was produced by software engineers at Tutored who work on the application. It is based on their knowledge of the domain without the help of any decision support tool. Tutored's software engineers also produced the input system model for *Pangaea*. As the manual solution included four microservices, we evaluated *Pangaea* with two configurations: $|M| = 4$ and $|M| = 5$. All our experiments are performed with the default input parameters presented in Sec. 2, unless otherwise specified. As a solver, we used Gurobi 9, with a maximum timeout of 7 minutes.

We compare the total cost of each solution using the cost function of *Pangaea*, based on the system model provided by the developers. Fig. 2 shows the results in terms of total cost of each solution and the fraction of it that is due to communication (the remaining part being organization). We configure SC with different graph algorithms (Girvan-Newman – GN, Leung, Chinese Whispers – CW). When a tool can be configured with an expected number of microservices, we indicate the number of microservices set as input in parenthesis. *Pangaea* (5) provides the solution with the lower total cost, and *Pangaea* (4), which uses the same number of microservices as the manual decomposition, has a cost that is only 14% higher. Interestingly, the total cost of the manual solution is about 3 times higher, and its communication cost is almost one order of magnitude higher than in *Pangaea*. Our interpretation is that developers tend to be more biased towards organization aspects, such as semantic affinities of data entities. As expected, the monolith solution incurs no communication cost but has a high total cost due to organization concerns (coupling), while the distributed solution results in a high communication cost. In terms of usability, SC provides disparate solutions depending on the selected algorithm, thus it requires developers to understand the details and differences between clustering algorithms. In absolute terms, SC solutions are between 95% and 373% more expensive than *Pangaea*.

Fig. 3 shows how the total cost of each solution changes with the organization-communication ratio α. Higher values of α linearly increase the cost of centralized solutions, such as the monolith and SC GN. Conversely, the cost linearly decreases for the distributed solution and SC Leung. The total cost of *Pangaea* is consistently lower than any other solution, the only exception being the distributed solution with low communication cost: however, this is an extreme case that artificially avoids coupling by creating an unrealistically high number of microservices. In conclusion, *Pangaea* solutions are the ones with the lowest total cost with balanced organization-communication ratio and outperform the other approaches even when the ratio changes.

To gether a better insight on the proposed decompositions, we manually analyzed their quality. The analysis offered a strong evidence that alternative approaches could not meet the expectations of developers, leading to decompositions that fall into two extremes: large microservices that cluster many functionalities with low cohesion or very small microservices that require frequent communication and do not justify a separate development and deployment.

To answer **RQ2**, we asked the developers at Tutored to provide an experience report. The time needed to produce the manual decomposition was between 6 and 8 hours, against the 2 hours needed to annotate the source code for *Pangaea*. In line with our objective, the developers described *Pangaea* as a support tool that can guide users in improving decomposition in an iterative fashion. The most important insights were the ones related to the communication cost, which is much harder to reason about and optimize with respect to organization aspects.

4 Conclusions

This paper introduced *Pangaea*, a semi-automated tool for decomposing a monolith into microservices. *Pangaea* uses a simple model of the application to formulate an optimization problem that balances organization, communication, and data management requirements. It outputs a graphical representation of the proposed decomposition together with detailed information on the costs it incurs. Our evaluation on a real-world application shows that *Pangaea* offers useful insights to developers. Our plans for future research include: (i) refine the model to enable more fine-grained modeling when suitable; (ii) evaluate alternative solving strategies to improve performance and scalability; (iii) extend the visualization tool to enable interactive adjustements of solutions.

References

1. Balalaie, A., Heydarnoori, A., Jamshidi, P., Tamburri, D.A., Lynn, T.: Microservices migration patterns. Software: Practice and Experience **48**(11) (2018)
2. Baresi, L., Garriga, M., Renzis, A.D.: Microservices identification through interface analysis. In: Europ. Conf. on Service-Oriented and Cloud Comput. ESOCC (2017)
3. Levcovitz, A., Terra, R., Valente, M.T.: Towards a technique for extracting microservices from monolithic enterprise systems (2016)
4. Mazlami, G., Cito, J., Leitner, P.: Extraction of microservices from monolithic software architectures. In: Intl. Conf. on Web Services (2017)
5. Selmadji, A., Seriai, A.D., Bouziane, H.L., Oumarou Mahamane, R., Zaragoza, P., Dony, C.: From monolithic architecture style to microservice one based on a semi-automatic approach. In: Intl. Conf. on Software Architecture. ICSA (2020)
6. Taibi, D., Systä, K.: From monolithic systems to microservices: A decomposition framework based on process mining. In: Intl. Conf. on Cloud Computing and Services Science. CLOSER (2019)