Ingeniería y
Competitividad
Vol 26 (1) 2024; doi: 10.25100/iyc.v26i1.13145

Vol 26 (1)

ISSN
0123-3033
e- 2027-8284

Article of investigation

Cognitive complexity points: a metric to evaluate the design of microservices-based applications

Puntos de complejidad cognitiva: una métrica para evaluar el diseño de aplicaciones basadas en microservicios

Fredy H. Vera–Rivera[1]

[1] Grupo de investigación en Inteligencia Artificial (GIA), Universidad Francisco de Paula Santander, Cúcuta, Colombia.

Abstract

The complexity of the software allows us to analyze how difficult to understand, implement and maintain the program can be. The metrics allow us to measure and estimate certain characteristics of the software to make decisions and corrective or preventive actions. The definition of the complexity of the microservices-based applications design is fundamental since it directly affects the performance of the application, development, testing, maintainability, storage (transactions and distributed queries), and the use and consumption of computational resources. In this paper, a cognitive complexity metric is proposed to evaluate the design and granularity of microservices-based applications, which define the required effort, or degree of difficulty to understand the microservices that make up the system. Typical cases were analyzed, which can appear in the design of microservices-based applications, the calculation of cognitive complexity was correct and consistent with the difficulty of understanding, maintaining, and developing a microservice system, therefore it is a viable option for analyzing complexity in microservices-based architecture.

Resumen

La complejidad del software permite analizar lo difícil que puede ser entender, implementar y mantener el programa. Las métricas nos permiten medir y estimar ciertas características del software para tomar decisiones y acciones correctivas o preventivas. La definición de la complejidad del diseño de aplicaciones basadas en microservicios es fundamental, ya que afecta directamente el rendimiento de la aplicación, los tiempos de desarrollo y prueba, la mantenibilidad, el almacenamiento (transacciones y consultas distribuidas), el uso y consumo de recursos computacionales. En este artículo se propone una métrica de complejidad cognitiva para evaluar el diseño y la granularidad de las aplicaciones basadas en microservicios, la cual define el esfuerzo requerido, o el grado de dificultad para comprender los microservicios que componen el sistema. Se analizaron casos típicos que pueden presentarse en el diseño de aplicaciones basadas en microservicios, en los cuales el cálculo de la complejidad cognitiva fue correcto y consistente con la dificultad de entender, mantener y desarrollar un sistema de microservicios, por lo tanto, la métrica propuesta es una opción viable para analizar la complejidad en sistemas basados en microservicios.

Keywords: Software metrics, software complexity, cognitive complexity, microservices granularity, microservices-based applications.

Palabras clave: Métricas de software, complejidad del software, complejidad cognitiva, granularidad de los micro – servicios, aplicaciones basadas en microservicios.

How to cite?

Vera–Rivera, F.H. Cognitive complexity points: a metric to evaluate the design of microservices-based applications. Ingeniería y Competitividad, 2024, 26(1) e-21013145. .

https://doi.org/10.25100/iyc.v26i1.13145

Recibido: 15-09-23
Aceptado: 26-02-24

Correspondencia:
fredyhumbertovera@ufps.edu.co

Conflict of interest: none declared

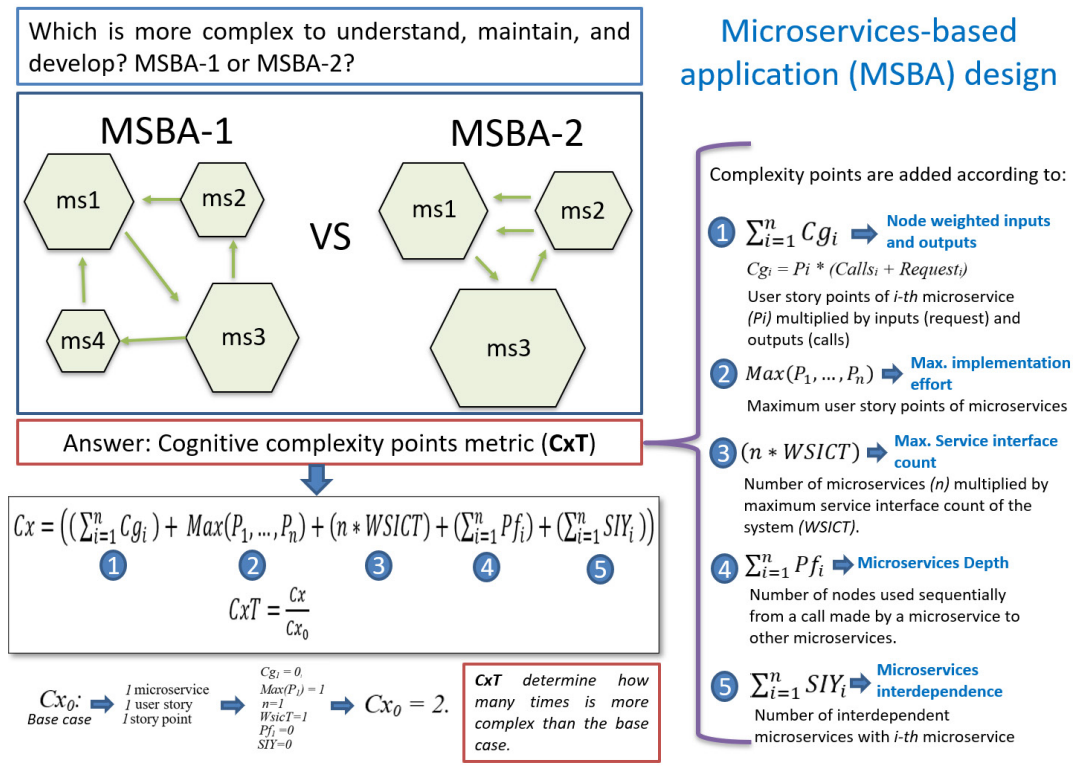
OPEN ACCESS

Why was it carried out?
When designing microservices-based applications, it is necessary to determine the number and services that each of them will implement, that is, to define their granularity. The reasoning and evaluation of the proposed design is essential for correct implementation and subsequent deployment. The proposed metric evaluates and estimates at design time the complexity of understanding and maintaining the microservices proposed in the design phase. It assigns complexity points according to the size of each microservice, its history points, its calls and requests, as well as the complexity of the graph they form. The metric was created as part of the Microservices Backlog, a model that uses intelligent algorithms to determine the granularity of microservices; this metric allows us to evaluate and compare the complexity of the solutions obtained by these algorithms.

What were the most relevant results?
The most important results consist in the definition of a way to estimate the complexity of understanding and implementing a microservices-based system at design time, making it possible to compare several proposals and to select the one with the least complexity. The metric was used in various case studies to assess the complexity of the proposed design and the granularity of the microservices. The metric allows us to choose the design with the least points of complexity.

What do these results provide?
The research proposes a novel metric to evaluate the complexity of microservices-based system and its granularity at design time.

Graphical Abstract



doi: 10.25100/iyc.v26i1.13145

Introduction

Software metrics allow us to measure and monitor different aspects and characteristics of the software product, there are metrics at the design, implementation, testing, maintenance, and deployment time. Therefore, these metrics allow us to understand, control, and improve what happens during the development, operation, and maintenance of the software and to take corrective and preventive actions.

One of the most important metrics is cyclomatic complexity, which can be used in the development or maintenance phases among others. Cyclomatic complexity is the metric that brings us how complex is the logic of a program, it is based on a graph that represents the flowchart that is determined by the representation of the control structures of a given program, it is a software metric that provides a quantitative measure of the logical complexity of a program (1). Complexity metrics have a lot of potential uses which include: the provision of feedback during a software project to help control the design activity, and the provision of detailed information about software modules to help pinpoint areas of potential instability during testing and maintenance (2).

The introduction of cognitive computing into the software engineering domain through the work of Wang (3) has led to the emergence of a new set of complexity metrics called cognitive complexity metrics. These metrics introduce cognitive weights, which define the required effort, relative time, or degree of difficulty to understand the software (2). Cognitive Complexity is a measure of the degree of difficulty involved in intuitively understanding a block of code; as opposed to cyclomatic complexity, which determines how difficult it is to test the code. To establish the value of cognitive complexity, points are established at which they must be fixed within an algorithm, as follows: a) It increases when there is a jump in the code flow (up-down, left-right); Some elements that increase cognitive complexity are: loops, conditionals, exceptions (try/ catch/throws), switch or case instructions, sequences of logical operators (a || b && c || d), recursion, jumping to labels (go to label), for loop. b) It is incremented when control structures are nested. c) The code is not more complex by using language structures that allow us to include several sentences in a single line. One of the purposes that cognitive complexity seeks is to encourage good practices when coding, so that in this way a more understandable and therefore maintainable product is obtained (4).

On the other hand, microservices are an architectural and organizational approach to software development in which applications are made up of small independent services that communicate through a well-defined Application Programming Interface (API) (5), many companies use microservices to structure their applications. Also, microservices architecture has been used in other areas such as the Internet of Things (IoT), edge computing, the development of autonomous vehicles, telecommunications, E-Health, and E-Learning systems, among others.

A great challenge when designing microservices-based applications is to find an appropriate partition or granularity of the microservices, it is performed and designed intuitively, according to the experience of the architect or the development team. The definition of the granularity of microservices is an open research topic. There are no standardized patterns, methods, or models that allow defining how small a microservice should be. The most used strategies to estimate the granularity of microservices are machine learning, semantic similarity, genetic programming, and domain engineering

doi: 10.25100/iyc.v26i1.13145

(6). During the design of a microservices-based application, the granularity of the microservices must be determined, how many microservices should make up the system, what relationship or dependencies exist between each one, and low coupling and high cohesion must be sought in the system.

Additionally, it is very important to determine the complexity of understanding, implementing, and maintaining the proposed design. The definition of the complexity of the microservices-based applications design is fundamental since it directly affects the performance of the application, development, testing, maintainability, storage (transactions and distributed queries), and the use and consumption of computational resources. The computing resources are used mainly in the cloud since the cloud is the most common platform where microservices are executed and deployed (7). Therefore, the following research question was proposed: How to measure the cognitive complexity of microservices-based applications design? To answer this question, the metric of cognitive complexity points (CxT) is proposed in this paper. This metric evaluates the difficulty of understanding and maintaining the design of microservices-based applications.

In previous work, we proposed the Microservices Backlog (*MB*) (8), (9), a semiautomatic model for defining and evaluating the granularity of microservice-based applications; MB decomposes the candidate microservices, allowing to analyze graphically the size of each microservice, as well as its complexity, dependencies, coupling, cohesion metrics, and the number of calls or requests between microservices. *CxT* is a fundamental part of *MB*, in previous works the approach of *CxT* is not fully formalized and explained, for this reason in this article its approach is detailed and its application in the design of microservices-based applications is validated.

A systematic literature review was carried out to identify the methods and metrics used to evaluate the granularity of microservices (10), within these metrics complexity metrics were identified: Function points. A method for measuring the size of the software. A function point count is a measurement of the amount of functionality that software will provide (11). COSMIC function points estimate the size in the planning phase, based on the user's functional requirements. The four main data group types are entry, exit, read, and write. The COSMIC function point calculation is aimed at measuring the system at the time of planning. This size calculation can be used for estimating efforts (12). Total response for service (TRS). The sum of all responses for operation (RFO) values for all operations of the interface of service (13). Number of singleton clusters (NSC) and Maximum cluster size (MCS) (14), these metrics are used to assess whether the size of the microservices is adequate.

Section 2 presents the methodology of this research, section 3 proposes de complexity metric to evaluate microservice-based application design, section 4 shows the results and discussions, and finally section 5 presents the conclusions.

Methodology

The research employed the approach of design science research as outlined by Hevner et al. (15). This research framework attempts to enhance the development of artefact creations through a consistent and iterative procedure, in which the artifact is evaluated and improved in each iteration. In this research, the artifact is the metric of cognitive complexity points to evaluate Microservices-based applications (MSBA) designs. Figure 1 shows the adaptation of design science research to this work.

doi: 10.25100/iyc.v26i1.13145

Based on the problem context and the knowledge base, the cognitive complexity points metric *(CxT)* is proposed, the mathematical formulation is presented in section 3, the methodology used to calculate the metric is: 1) Define the user stories and user stories dependences, 2) Define microservices granularity and MSBA design, 3) Define de microservices Backlog model, 3) Calculate the metrics calls, request, microservices story points, microservices interdependences, weighted service interface count, with this metrics *CxT* can be calculated. Then, the metric results are validated in simple cases and typical cases, which can appear in many MSBA designs. Finally, *CxT* is used in four study cases to evaluate microservice granularity.

Cognitive complexity points (CxT)

Measuring complexity is critical to developing microservices-based applications. If the complexity is high, the cost of change is higher, also the cost of implementation and the development time are increased. Measuring the complexity of the system at design time is important for decision-making for its implementation, testing, maintenance, and deployment.

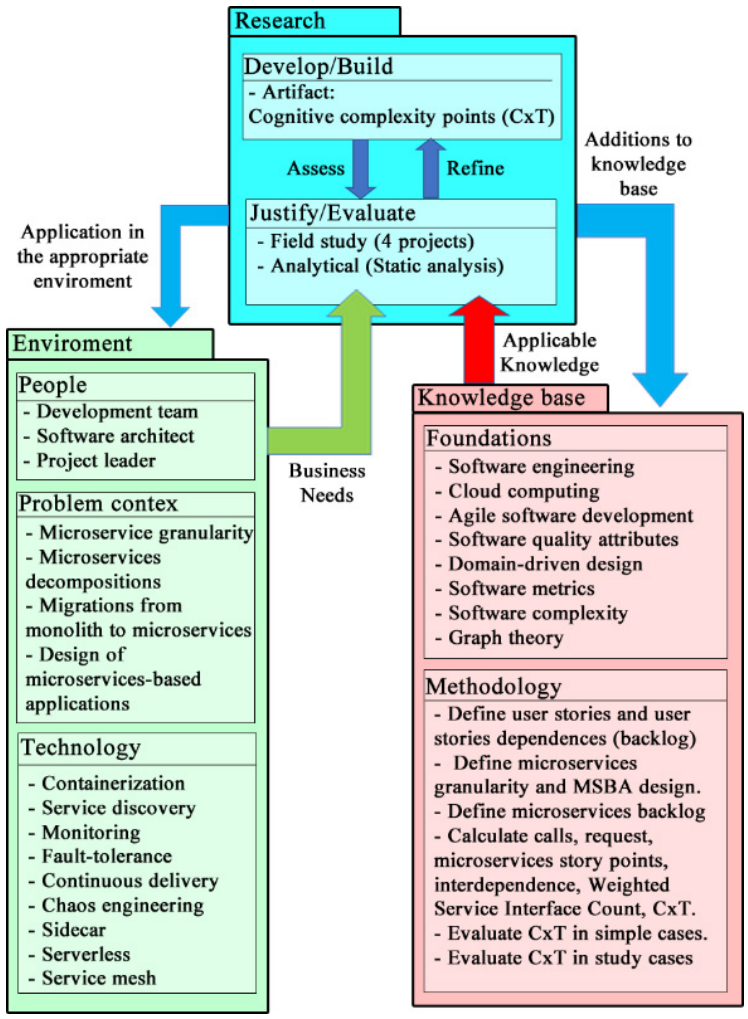


Figure 1. Research methodology

The mathematical formulation begins with a microservices-based application (MSBA), that is made up of a set of microservices (MS_i), as shown in Ec.1.

doi: 10.25100/iyc.v26i1.13145

$$MSBA = \{MS_1, MS_2, MS_3, \dots, MS_i\} \quad (1)$$

Each MS_i is composed of user stories or operations (US_j), Ec. 2. The user stories have an identifier, a name, a description, an estimated effort points (P), an estimated development time (T), a priority, and dependencies between each story that makes up the system.

$$MS_i = \{US_1, US_2, US_3, \dots, US_j\} \quad (2)$$

User story points are an estimate of the effort required to develop the user story. The points are an indicator of the development speed of the team; therefore, each microservice (MS_i) has a total of story points associated with it (equation 3).

$$P_i = \sum_{j=1}^{m} PU_j \quad (3)$$

Where:

P_i is the total story points associated with MS_i. m is the number of user stories associated with the MS_i. PU_j corresponds to the story points estimated by the development team for the j_{th} user story of MS_i.

CxT is calculated by adding complexity points according to the complexity of the microservice. It was raised in accordance with the following postulates:

The difficulty of developing, maintaining, and understanding an application based on microservices is estimated.

The starting point was the estimation of the user story points made by the development team.

Points are added according to the complexity of the microservice, its relationships, and its dependencies.

It is based on the complexity of a graph and its depth.

A base case is considered, which corresponds to the least complexity; this case would be an *MSBA* with only one microservice, with a value of one point of the estimated story point for its development, which would be the simplest case to develop. For this case $Cx_0 = 2$.

CxT corresponds to the number of times that the MSBA is more complex in relation to the base case.

Cognitive points are increased according to the next points:

The total estimated points for each microservice that is part of the application *(P_i)*.

The number of microservices that are part of the application *(n)*.

doi: 10.25100/iyc.v26i1.13145

The number of user stories or operations associated with each microservice.

The number of invocations - calls (out) and requests (in) of the microservice.

The depth of the number of calls that a microservice makes to other microservices. Corresponds to the number of consecutive nodes used in the call from one microservice to another.

Formally *CxT* was defined as follows:

$$Cx = \left(\left(\sum_{i=1}^{n} Cg_i \right) + Max(P_1, \ldots, P_n) + (n * WSICT) + \left(\sum_{i=1}^{n} Pf_i \right) + \left(\sum_{i=1}^{n} SIY_i \right) \right) \quad (4)$$

$$CxT = \frac{Cx}{Cx_0} \quad (5)$$

Where:

i = ith microservice.

$Cg_i = Pi * (Calls_i + Request_i)$

P_i = Total user stories points of the *i-th* microservice. See equation (3).

Max(P1, ..., Pn): Maximum P_i of *MSBA.*

n = number of microservices of *MSBA.*

WSICT: Highest *WSIC* of the application, defined in equation (6).

Pf_i*:* Number of nodes used sequentially from a call made by a microservice to other microservices, counted from the i_{th} microservice; Greater depth implies greater complexity of deploying and maintaining the application.

SIY: Microservice interdependence, number of interdependent microservices with MS_i.

Cx_0*:* The base case where the application has one microservice, with one user story and one point of estimated story point. So $Cg_1 = 0$, $Max(P_1) = 1$, $n=1$, $WsicT=1$, $Pf_1 = 0$, $SIY=0$, $Cx = 2$. Therefore, $Cx_0 = 2$.

Measuring or estimating the performance of an application at design time is difficult and imprecise. We use calls and requests between microservices to estimate performance. We assume that if there are more calls and requests between the microservices, then the communication, latency, and response time of the application increases, therefore the performance of the application is directly affected. Ideally, in a microservices-based application, you would have microservices that do not communicate with each other and that work independently. Therefore, we define two metrics:

doi: 10.25100/iyc.v26i1.13145

Calls from a microservice *(calls$_i$)*: The calls correspond to the number of *MS$_i$* invocations to other microservices.

Requests for a microservice *(requests$_i$)*: Requests correspond to the number of invocations of other microservices to *MS$_i$*.

Figure 2 shows the calculation of the calls, requests, and WSICT of the MSBA with three microservices and four user stories.

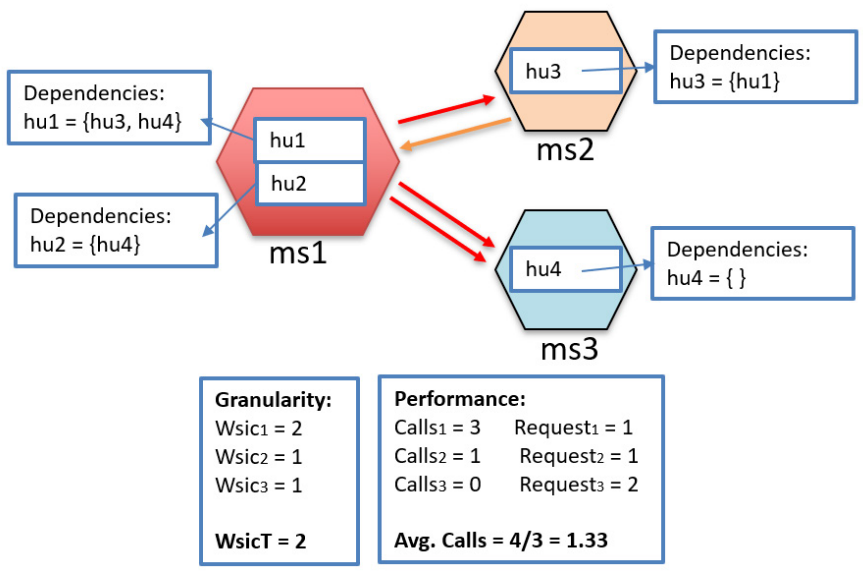


Figure 2. Example of the calculation of *WSICT*, *calls,* and *requests*.

Weighted Service Interface Count *(WSIC)*: The *WSIC* is the number of exposed interface operations of the *MS$_i$* (16). For our model, a user story is related to an operation (one-to-one); therefore, we adapt this metric as the number of user stories associated with the *MS$_i$*. Other authors called this metric the number of operations. We define *WSICT* as the maximum number of user stories associated with a microservice, so *WsicT* is the maximum *WSIC* of the *MSBA*, so:

$$WSICT = Max(WSIC_1, WSIC_2, \ldots, WSIC_n) \quad (6)$$

Microservice Interdependence *(SIY)*: *SIY* corresponds to the number of interdependent microservice pairs (12). In this case, *SIY$_i$* defines the number of pairs of microservices that are bidirectionally dependent on *MS$_i$* divided by the total number of microservices. In the example of Figure 1, it is possible to calculate *SIY* as follows: *SIY$_1$ = 1/3 =0.33*, *SIY$_2$ = 1/3 =0.33,* and *SIY$_3$ = 0*; because *ms$_1$* has one interdependent microservice just like *ms$_2$* while *ms$_3$* has no interdependency.

Additionally, for the example of Figure 1, we define one point of estimated story points for each user story, we calculated *CxT* as follows:

P$_1$ = 2, P$_2$ = 1, and *P$_3$ = 1.*

*Cg$_i$ = Pi * (Calls$_i$ + Request$_i$),*

Cg$_1$ = P$_1$ (calls$_1$ + request$_1$) = 2 (3+1) = 8

doi: 10.25100/iyc.v26i1.13145

$Cg_2 = P_2\ (calls_2 + request_2) = 1\ (1+1) = 2$

$Cg_3 = P_3\ (calls_3 + request_3) = 1\ (0+2) = 2$

$\sum_{i=1}^{n} Cg_i$

$= 12$

$Max(P_1, \dots, P_n)$

$= 2$

WSIC1=2, WSIC2=1, WSIC3=1, so WSICT = 2; n=3, therefore *n * WSICT = 6*

Pf_i*:* Number of nodes used sequentially from a call made by a microservice to other microservices.

$Pf_1 = 2$, A call is performed by ms_1 to ms_2, and another call is performed by ms_1 to ms_3.

$Pf_2 = 1$, A call is performed by ms_2 to ms_1.

$Pf_3 = 0$, No calls are performed by ms_3.

$$\sum_{i=1}^{n} Pf_i = 3$$

$$\sum_{i=1}^{n} SIY_i = 0.33 + 0.33 + 0 = 0.66$$

$$CxT = \frac{Cx}{Cx_0} = \frac{12+2+6+3+0.66}{2} = 11.83 \text{ points}$$

The cognitive complexity points (CxT) for this case imply that MSBA is 11.83 times more complex than the base case.

A critical point of the proposed metric is the dependencies between user stories. They must be identified and provided as input data to the method. A dependency is defined between US_i and US_j when US_i calls or executes US_j. For example, to create a fly trip (US_1) you must obtain get the city (US_2) of departure and destination, this implies that US_1 has a dependency on US_2. The dependencies can be calculated according to the business logic of the application.

Additionally, dependency is defined when a user story uses or calls another user story. In the migration from monolith to microservices, the user stories can be replaced by the operations/methods or services of the application; in this case, a dependency corresponds to an execution dependency, in which one operation calls another operation to fulfill its purpose. In the cases where the monolithic application source code is available; to define the dependencies between user stories, the source code can be analyzed to identify the invocation dependencies between user stories and/or operations.

doi: 10.25100/iyc.v26i1.13145

Implementation algorithm

The cognitive complexity metric was automated and implemented as part of the Microservice Backlog tool (8), within the metrics calculator component. The algorithm that was implemented for its automatic calculation is summarized below.

Cognitive Complexity algorithm

1. CgiT=0
2. SumSIY=0
3. Sumpf = 0
4. For i=1 to n
5. Get p= $P_i(i)$
6. Calculate c = $calls_i(i)$
7. Calculate r = $request_i(i)$
8. Calculate cgi = p * (c+r)
9. Calculate pf = $Pf_i(i)$
10. Calculate Sumpf = Sumpf + pf
11. Identify mpi = max(P_i)
12. Identify mwsic = max($Wsic_i$)
13. Calculate CgiT = CgiT + cgi
14. Calculate siy = $SIY_i(i)$
15. Calculate SumSIY = SumSIY + siy
16. End for
17. Calculate w = n * mwsic
18. Calculate Cx = CgiT + mpi + w + Sumpf + SumSIY
19. Calculate CxT = Cx / Cx_0 = Cx / 2

Calls and requests are calculated through a matrix of invocations where each MS_i is in the rows and columns and at their intersection the number of times that one microservice invokes another appears. The algorithm was implemented in Python and used to calculate cognitive complexity in the examples and case studies.

Results and discussion

To validate the proposed cognitive points metric (CxT), we are going to assume several cases where a microservices-based application is clearly more complex than another, carry out the calculations and verify that the given points correspond correctly to each application, with the least complex having fewer points and more points to the more complex.

Case 1: Two microservices without dependencies versus the same microservices with dependencies

If there are dependencies and invocations between the microservices that make up the MSBA, their complexity must be greater than an MSBA without a dependency, therefore the CxT of MSBA-a must be less than CxT of MSBA-b. Figure 3 shows this example and the calculation of CxT. The results show that indeed the CxT of MSBA-a is 4 times higher than the CxT of MSBA-b.

doi: 10.25100/iyc.v26i1.13145

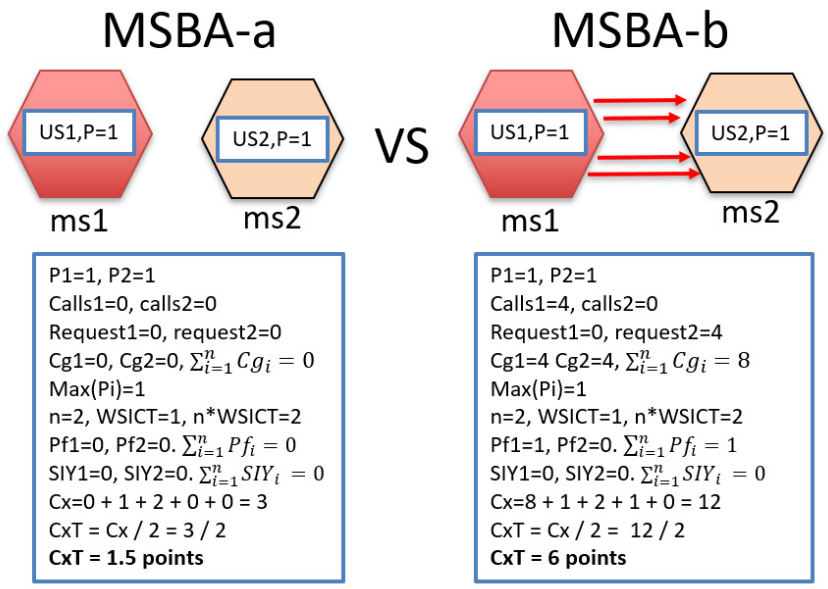


Figure 3. Example of the calculation of CXT case 1.

Case 2: Two microservices with interdependence versus the same microservices without interdependence

The interdependence between microservices implies a high coupling between them, causing greater complexity when implementing, maintaining, and deploying the application. Any change applied to a microservice may imply changing the other microservice as well. Figure 4 shows the calculation for this case, where the complexity CxT of MSBA-b without interdependence may be less than the CxT of MSBA-c with interdependence. The results confirm this approach.

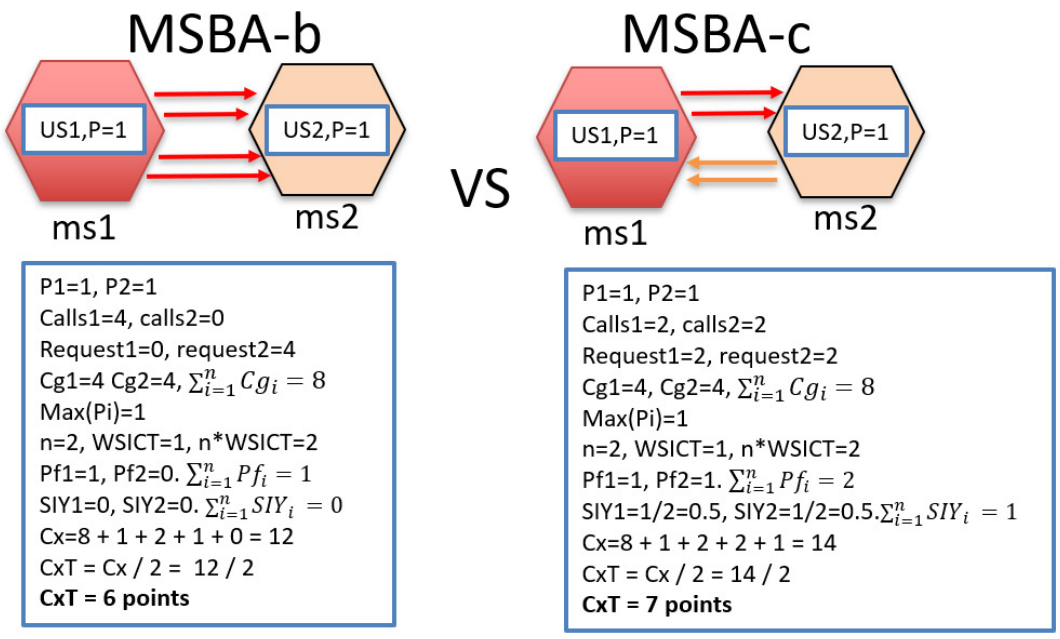


Figure 4. Example of the calculation of CXT case 2.

doi: 10.25100/iyc.v26i1.13145

Case 3: Two microservices with few calls versus the same microservices with high calls

Creating a microservices-based application where there are many calls or dependencies between the microservices implies more complexity than an application with fewer calls. Figure 5 shows this case, where CxT of MSBA-d is less than CxT of MSBA-b, and CxT of MSBA-a is also lower than CxT of MSBA-d, confirming that the lower number of invocations of other microservices, the cognitive complexity points should be lower.

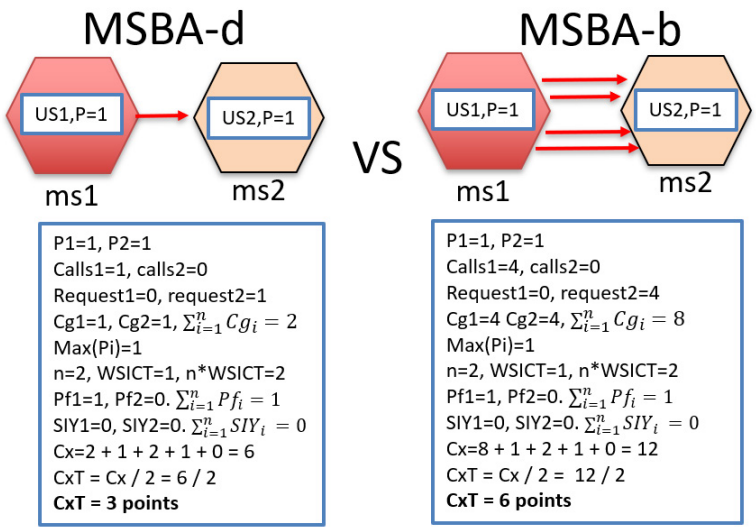


Figure 5. Example of the calculation of CXT case 3.

Case 4: calls with depth versus the same calls without depth

Successive invocations in a microservices-based application imply more complexity than having those same invocations with a smaller depth. Figure 6 shows this case and confirms this statement, CxT of MSBA-e is greater than CxT of MSBA-b.

Case 5: calls to a microservice with fewer story points versus calls to a microservice with higher story points

Developing and maintaining a microservice that has a higher number of story points is more complex, the effort is higher; if the story points increase the complexity must increase. This case is illustrated in Figure 7. We can see that the CxT of MSBA-b is less than the CxT of MSBA-f.

The analyzed cases correspond to typical cases that can appear in the MSBA design, the calculation of cognitive complexity (CxT) is correct and consistent with the difficulty of understanding, maintaining, and developing an MSBA, therefore it is a viable option for analyzing complexity in MSBA.

Afterward, we detail the calculation of the metric in case studies in which we obtained different designs and we calculated CxT to compare and measure their complexity in four study cases (Cargo Tracking, JPet Store, Foristom conferences, and Sinplafut). We used CxT to evaluate methods and algorithms for the microservice granularity definition (8), (9), (17). Table 1 shows the results.

doi: 10.25100/iyc.v26i1.13145

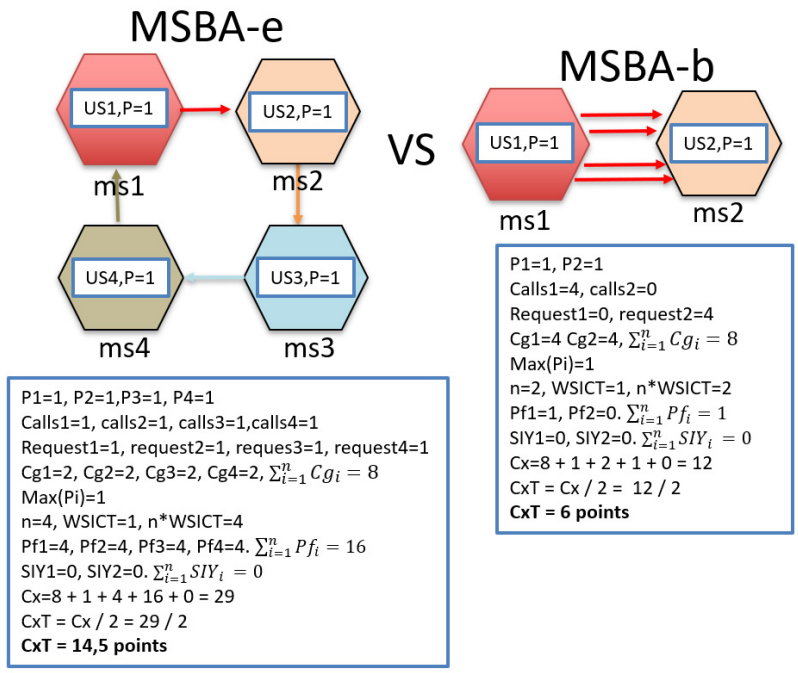


Figure 6. Example of the calculation of CXT case 4

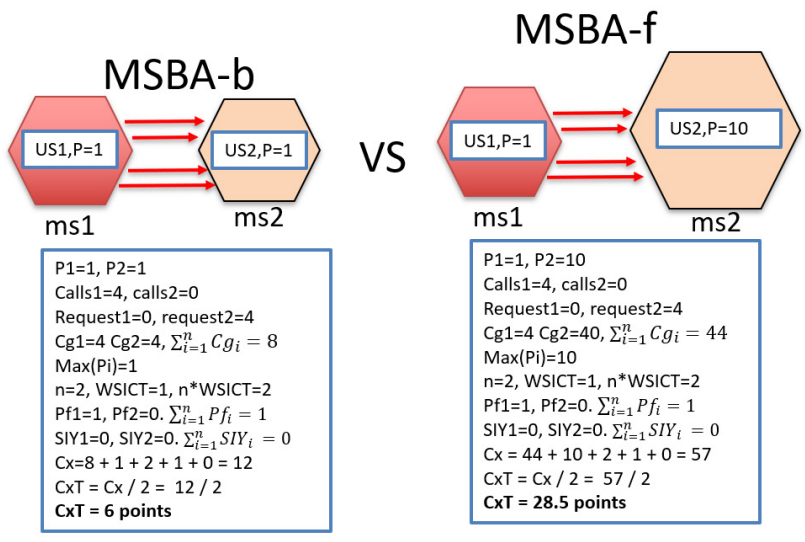


Figure 7. Example of the calculation of CXT case 5

The design method (second column) corresponds to the procedure or method used to obtain the microservices of MSBA, which were obtained from the state-of-the-art revision, they are used to define the microservices granularity.

Table 1 shows different CxT calculations for the study cases, with different microservices granularity, some with more or fewer microservices, with different numbers of user stories, with different estimations of story points, some with many invocations between microservices and others with very few, even zero. Cognitive complexity estimates the degree of difficulty to understand, create and maintain an MSBA, for the cases analyzed it helps to make design decisions, and allows the architect or development team to select the design with less complexity. Making these kinds of decisions at design time can help to reduce MSBA development, maintenance, and deployment costs.

doi: 10.25100/iyc.v26i1.13145

Table 1. Study cases results

Study case	Design Method	Metrics					
		n	WsicT	Max Pi	Calls	Avg. Calls	CxT
Cargo-Tracking (18) (19)	Genetic programming (8)	3	6	23	3	1.0	74.0
	Semgromi (9)	4	9	35	8	2.0	178.5
	Domain-driven design (DDD) (20)	4	6	27	9	2.3	145.0
	Service Cutter (21)	3	10	41	8	2.7	202.5
	MITIA (19)	4	5	19	12	3.0	190.0
Jpet-Store (22)	Genetic programming	5	9	35	3	0.6	102.5
	Semgromi	5	6	20	7	1.4	140.5
	Domain-driven design (DDD)	4	8	22	9	2.3	200.0
	Execution Traces (23)	4	7	19	8	2.0	175.5
Foristom Conferencias (8)	Genetic programming	4	8	67	0	0.0	49.5
	Semgromi	5	13	90	7	1.4	466.5
	Domain-driven design (DDD)	4	9	83	6	1.5	426.0
Sinplafut (24)	Genetic programming	13	13	49	24	1.8	788.5
	Semgromi	11	16	58	24	2.2	814.0
	Domain-driven design (DDD)	9	19	75	23	2.6	920.5
	A priori development team.	5	34	127	9	1.8	721.0

Most of the complexity metrics are calculated at execution time, or at development time, for their calculation the source code of the program is required to identify

doi: 10.25100/iyc.v26i1.13145

the flows and time it takes to execute that program. At design time, few metrics are proposed, therefore, CxT represents a contribution to the analysis of software complexity, specifically in microservices-based applications. For calculating CxT does not require the source code of the microservices or runtime information.

Conclusions

In this work, a cognitive complexity metric was proposed that allows estimating the effort and difficulty of understanding, developing, and maintaining a microservices-based application at design time. The estimation of the complexity is made from the proposed design, in which the granularity of the microservices, their relationships, and dependencies must be defined.

The mathematical formulation is based on graph theory, where the calls (outputs) and requests (inputs) between each microservice, their interdependence, and the sequential call (depth) that can occur between the microservices are considered. Its calculation was demonstrated in a set of typical cases that can occur in any microservices-based application, verifying that there are fewer points in cases of low complexity and more points in cases of high complexity.

Additionally, the metric was used to compare methods for defining the granularity of microservices in four study cases, allowing a comparative analysis of cognitive complexity, the development team can evaluate different ways of distributing user stories in microservices and make decisions at design time. Therefore, with this proposal we can reason about the complexity and granularity of microservices at design time, thus covering one of the research gaps proposed in the state-of-the-art.

References

1. Pressman RS. Ingeniería del Software un enfoque práctico. Septima. Mexico: McGraw Hill Interamericana editores; 2010.

2. Misra S, Adewumi A, Fernandez-Sanz L, Damasevicius R. A Suite of Object Oriented Cognitive Complexity Metrics. IEEE Access. 2018;6(c):8782–96.

3. Wang Y. On the cognitive informatics foundations of software engineering. In: Proceedings of the Third IEEE International Conference on Cognitive Informatics, 2004 [Internet]. IEEE; 2004 [cited 2019 Feb 6]. p. 22–31. Available from: http://ieeexplore.ieee.org/document/1327456/

4. López López S. Complejidad Cognitiva [Internet]. enmilocalfunciona.io. 2018 [cited 2019 Feb 5]. Available from: https://enmilocalfunciona.io/complejidad-cognitiva/

5. Newman S. Building Microservices [Internet]. First Edit. Building Microservices. Gravenstein Highway North, Sebastopol, CA 95472, United States of America: O'Reilly Media, Inc.; 2015. 102 p. Available from: http://oreilly.com/catalog/errata.csp?isbn=9781491950357 for

doi: 10.25100/iyc.v26i1.13145

6. Vera-Rivera FH, Gaona C, Astudillo H. Defining and measuring microservice granularity—a literature overview. PeerJ Comput Sci [Internet]. 2021 Sep 8 [cited 2022 Feb 7];7:e695. Available from: https://peerj.com/articles/cs-695

7. Hamzehloui MS, Sahibuddin S, Salah K. A Systematic Mapping Study on Microservices Mohammad. In: Saeed F, Gazem N, Mohammed F, Busalim A, editors. IRICT: International Conference of Reliable Information and Communication Technology 2018 [Internet]. Cham: Springer International Publishing; 2019. p. 1079–90. (Advances in Intelligent Systems and Computing; vol. 843). Available from: http://link.springer.com/10.1007/978-3-319-99007-1

8. Vera-Rivera FH, Puerto E, Astudillo H, Gaona CM. Microservices Backlog–A Genetic Programming Technique for Identification and Evaluation of Microservices From User Stories. IEEE Access [Internet]. 2021 [cited 2022 Feb 3];9:117178–203. Available from: https://ieeexplore.ieee.org/document/9519691/

9. Vera-rivera FH, Puerto Cuadros E, Perez B, Gaona Cuevas CM, Astudillo H. SEMGROMI — a semantic grouping algorithm to identifying microservices using semantic similarity of user stories. PeerJ Comput Sci. 2023 May;9(e1380).

10. Vera-Rivera FH, Astudillo H, Gaona-Cuevas CM. Defining and measuring microservice granularity – a literature overview. PeerJ Comput Sci. In review.

11. Totalmetrics.com. Total Metrics Approach - Function points [Internet]. www.totalmetrics.com. [cited 2020 Aug 5]. Available from: https://www.totalmetrics.com/our_approach

12. Vural H, Koyuncu M, Misra S. A Case Study on Measuring the Size of Microservices. In: Laganá A, Gavrilova ML, Kumar V, Mun Y, Tan CJK, Gervasi O, editors. International Conference on Computational Science and Its Applications - ICCSA 2018 [Internet]. Berlin, Heidelberg: Springer Berlin Heidelberg; 2018. p. 454–63. (Lecture Notes in Computer Science). Available from: http://link.springer.com/10.1007/b98048

13. Perepletchikov M, Ryan C, Frampton K, Tari Z. Coupling Metrics for Predicting Maintainability in Service-Oriented Designs. In: 2007 Australian Software Engineering Conference (ASWEC'07) [Internet]. IEEE; 2007 [cited 2019 Jun 18]. p. 329–40. Available from: http://ieeexplore.ieee.org/document/4159685/

14. Nunes L, Santos N, Rito Silva A. From a Monolith to a Microservices Architecture: An Approach Based on Transactional Contexts. In: 13th European Conference, ECSA 2019 Lectures Notes in Computer Science 11681 [Internet]. Springer; 2019. p. 37–52. Available from: http://link.springer.com/10.1007/978-3-030-29983-5_3

15. Hevner AR, March ST, Park J, Ram S. Design science in information systems research. MIS Q [Internet]. 2004 [cited 2018 May 16];28(1):75–105. Available from: https://pdfs.semanticscholar.org/fa72/91f2073cb6fdbdd7c2213bf6d776d0ab411c.pdf

doi: 10.25100/iyc.v26i1.13145

16. Hirzalla M, Cleland-Huang J, Arsanjani A. A Metrics Suite for Evaluating Flexibility and Complexity in Service Oriented Architectures. In Springer, Berlin, Heidelberg; 2009 [cited 2019 Jun 18]. p. 41–52. Available from: http://link.springer.com/10.1007/978-3-642-01247-1_5

17. Vera-Rivera FH, Puerto-Cuadros EG, Astudillo H, Gaona-Cuevas CM. Microservices Backlog - A Model of Granularity Specification and Microservice Identification. In: International conference on service computing SCC 2020 Lecture Notes in Computer Science [Internet]. Springer Science and Business Media Deutschland GmbH; 2020 [cited 2020 Nov 20]. p. 85–102. Available from: https://link.springer.com/chapter/10.1007/978-3-030-59592-0_6

18. Li S, Zhang H, Jia Z, Li Z, Zhang C, Li J, et al. A dataflow-driven approach to identifying microservices from monolithic applications. J Syst Softw. 2019;157.

19. Baresi L, Garriga M, De Renzis A. Microservices identification through interface analysis. In: European Conference on Service-Oriented and Cloud Computing - Lecture Notes in Computer Science [Internet]. Springer, Cham; 2017 [cited 2017 Nov 2]. p. 19–33. Available from: http://link.springer.com/10.1007/978-3-319-67262-5_2

20. Evans E. Domain-Driven Design [Internet]. Addison Wesley; 2004. 529 p. Available from: http://dddcommunity.org/book/evans_2003/

21. Gysel M, Kölbener L, Giersche W, Zimmermann O. Service Cutter: A Systematic Approach to Service Decomposition. In: IFIP International Federation for Information Processing 2016 [Internet]. Springer, Cham; 2016 [cited 2019 May 17]. p. 185–200. Available from: https://link.springer.com/chapter/10.1007%2F978-3-319-44482-6_12

22. mybatis.org. Mybatis Jpetstore-6: A web application built on top of MyBatis 3, Spring 3 and Stripes [Internet]. [cited 2020 Nov 22]. Available from: https://github.com/mybatis/jpetstore-6

23. Jin W, Liu T, Cai Y, Kazman R, Mo R, Zheng Q. Service Candidate Identification from Monolithic Systems based on Execution Traces. IEEE Trans Softw Eng [Internet]. 2019;X(X):1–1. Available from: https://ieeexplore.ieee.org/document/8686152/

24. Vera-Rivera FH, Vera-Rivera JL, Gaona-Cuevas CM. Sinplafut: A microservices – based application for soccer training. In: 5th International Week of Science, Technology & Innovation Journal of Physics: Conference Series [Internet]. 2019. p. 012026. Available from: https://iopscience.iop.org/article/10.1088/1742-6596/1388/1/012026

doi: 10.25100/iyc.v26i1.13145