Received: 15 March 2021 | Revised: 26 October 2021 | Accepted: 29 November 2021

DOI: 10.1002/smr.2418

SPECIAL ISSUE - EMPIRICAL

Journal of Software: Evolution and Process WILEY

The athletic heart syndrome in web service evolution

Apostolos V. Zarras | Ioannis Dinos | Panos Vassiliadis

Department of Computer Science & Engineering, University of Ioannina, Ioannina, Greece

Correspondence
Apostolos V. Zarras, Department of Computer Science & Engineering, University of Ioannina, Ioannina, Greece.
Email: zarras@cs.uoi.gr

Abstract

Despite the particular standards, technologies, and trends (W3C, RESTful, micro-services, etc.) that a team decides to follow for the development of a service-oriented system, most likely the team members will have to use one or more services that solve general-purpose problems like cloud computing, networking and content delivery, storage and database, management and governance, and application integration. Typically, general-purpose services are long-lived, they have several responsibilities, their interfaces are complex, and they grow over time. The way that these services evolve also affects the evolution of any system that will depend on them. Consequently, the selection of the particular services that will be used is a main concern for the team. In this paper, we report a pattern, called *the athletic heart syndrome*, which facilitates the selection of services that evolve properly. Patterns specify best practices that emerge from multiple real-world cases. In our context, *the athletic heart syndrome* comes out from a study that concerns the evolution of a set of popular, long-lived Amazon services that cover different domains. According to *the athletic heart syndrome*, the developers should select services whose heartbeat of changes looks like the heartbeat of an athlete when he is at rest. Specifically, the heartbeat of changes should consist mostly of calm periods, interrupted by few spikes of change. Similarly, the incremental growth of the services should involve mainly calm periods of maintenance, separated by spikes of growth. Selecting services that adhere to the pattern signifies high chances that the services evolve to deal with changing requirements. The pattern further guarantees that the service evolution involves both the expansion of the services with new functionalities and the maintenance of existing ones. The pattern also assures that the complexity increase in the service interfaces will be smooth and tolerable. Finally, conformance with the pattern implies that the growth of the services will be predictable.

KEYWORDS
complex systems, web service evolution, Lehman's laws

1 | INTRODUCTION

Web services constitute a particular class of software systems that expose their functionalities through the web. They emerged as a promising solution that facilitates the development of complex distributed systems, by promoting software reuse and reducing development cost. Over the years, different standards, technologies, and trends came up.[1] W3C services[1] expose a set of operations, specified in terms of the Web Services

[1]We use the term *W3C* services to refer to *document-passing* services (typically operating with respect to the SOAP and WSDL standards); they also come by the name *WS-** services.

Description Language (WSDL), while RESTful services expose a set of resources, which can be accessed using the standard HTTP primitives. More recently, the current trend in the development of service-oriented systems is microservices.[1] According to this trend, a service-oriented system is composed of small independent services, provided by small self-contained teams. Microservices are specialized services with few, or even a single, responsibility. Consequently, they can be easily, reused, maintained, or even replaced.

Regardless of the standards, technologies, and trends that a team chooses to follow for the development of a particular system, usually there is a need to use one of more services that solve general-purpose problems like cloud computing, networking and content delivery, storage and database, management and governance, and application integration. Typically, these services have been around for long and most likely they do not follow the most recent trends, concerning the way that they have been designed and implemented. As we have already shown in prior studies, general-purpose services have many responsibilities,[2] they have complex interfaces, and they grow bigger as they evolve.[3] Nevertheless, the developers of service-oriented systems still have to use them because, as we said before, they solve problems that are frequently encountered in many different kinds of systems.

Like all software, general-purpose services evolve over time. Their evolution may involve the expansion of the services with new functionalities to meet new requirements, or the maintenance of existing functionalities.[3] The evolution process is under the control of the service providers. This fact alone makes the evolution of general-purpose services a very important issue that can affect the viability of the system that depends on them and a major concern for the developers of the dependent system. Relying on services that do not evolve properly shall have a negative impact on the evolution of the dependent system, as well.

In general, service evolution is a challenging research issue with several interesting approaches that emerged so far, towards dealing with this issue. In particular, the state of the art on service evolution includes three central topics, specifically: (i) *change-detection tools*, that is, tools that enable the detection of changes in subsequent service releases, and empirical studies that employ such tools to investigate the evolution properties of real-world services;[4–8] (ii) *frameworks* that facilitate the management of service changes, both for the service providers, and the developers of the systems that depend on the evolving services;[4,9–14] and, (iii) *patterns* that specify best practices allowing to tackle the evolution of services in a more systematic and easy way.[15,16]

In our prior efforts,[3] we investigated the issue of service evolution from a theoretical perspective. Specifically, we studied a set of popular, long-lived, general-purpose Amazon services that cover different domains, to see if their evolution follows Lehman's laws of software evolution.[17] Our findings showed that the evolution of the examined services follows to a large extent Lehman's theory, which states that *a software system that solves a real-world problem (i.e., an E-type systems in Lehman's terminology) evolves in a controlled way, with respect to a feedback based evolution process*. The feedback is of two kinds, positive and negative. *Positive feedback* leads to the growth of the system to meet new requirements, while negative feedback triggers maintenance activities that limit the growth of the system. Table 1 summarizes the findings of our previous work concerning the validity of Lehman's laws in the case of the examined services. Specifically, we found evidence that supports the validity of Laws I, II, III, V, VI, and VIII. On the other hand, we disproved Law IV, while for Law VII we were not able to reach any conclusion.

In this paper, we revisit the issue of service evolution, this time from a more practical viewpoint. *We get in the shoes of the developer who wants to select a general-purpose service to use in his system. The developer looks for a service that meets his functional requirements. However, he also wants to be sure that the service evolves properly*. Specifically, the developer seeks a service that is enhanced and maintained by the service provider. As the service evolves, he wants to be sure that the complexity of the provided interface will not be a problem for service usage. He also wants the service to evolve in a way that allows him to comprehend and master the provided functionalities. Finally, the developer wants to make predictions about the evolution of the service.

Our main research objective is to provide the developer with a pattern that formally specifies the key criteria for selecting a general-purpose service that evolves according to his expectations. To identify these criteria and specify them in the form of a pattern, we perform a more detailed study of the services that we investigated in our previous work.[3] As we already know that these services live normal lives that respect Lehman's theory of software evolution, our goal is to identify the properties that indicate proper service evolution. These properties become the basic selection criteria of the pattern. As in our prior work,[3] we study changes performed in the operations that are provided by the services. In addition, in this paper, we study changes performed in the data-types that are used for sending/receiving data to/from the services. Moreover, we study the detailed service release notes that provide more insight concerning the purpose of the changes. More formally, we specify our research goal as follows.

> **Research Goal:** *Given a set of popular, long-lived, general-purpose services that cover different domains, identify a pattern that specifies common properties which indicate proper service evolution, to facilitate the selection of other services that also evolve with respect to these properties*.

To address our research goal, we focus on a more detailed list of research questions, given below. We begin by looking for a pattern in the heartbeat of service changes and in the growth of the service functionalities. Then, we investigate the purpose of the service evolution and particularly the various kinds of growth and maintenance (adaptive, perfective, corrective) activities it involves. Following, we focus on the impact

TABLE 1 Lehman's laws of software evolution in the case of Amazon services

Lehman's Laws	AWS Services
	EC2, ELB, AS, SQS, RDS, MTurk
Law I - Continuing Change: *An E-type system must be continually adapted, or else it becomes less satisfactory in use.*	*Confirmed*
Law II - Increasing Complexity: *As an E-type system is changed its complexity increases and becomes more difficult to evolve, unless work is done to maintain or reduce the complexity.*	*Confirmed*
Law III - Self Regulation: *Global E-type system evolution is feedback regulated.*	*Confirmed*
Law IV - Conservation of Organizational Stability: *The work rate of an organization evolving an E-type system tends to be constant over the operational lifetime of that system, or phases of that lifetime.*	*Disproved*
Law V - Conservation of Familiarity: *The incremental growth of E-type systems is constrained by the need to maintain familiarity.*	*Confirmed*
Law VI - Continuing Growth: *The functional capability of E-type systems must be continually enhanced to maintain user satisfaction over system lifetime.*	*Confirmed*
Law VII - Declining Quality: *The quality of an E-type system will appear to be declining, unless rigorously maintained and adapted to operational environment changes.*	*Inconclusive*
Law VIII - Feedback System: *E-type evolution processes are multi-level multi-loop, multi-agent feedback systems.*	*Confirmed*

of service evolution on the complexity of the service interfaces. Finally, we investigate the possibility of making predictions about the way services evolve.

- **RQ1:** *Is there a pattern in the evolution of the services?*
- **RQ2:** *What is the purpose of the service evolution?*
- **RQ3:** *What is the impact of the service evolution on the complexity of the service interfaces?*
- **RQ4:** *Is it possible to make predictions about the evolution of the services?*

The main research result of our study is *the athletic heart syndrome* pattern. *According to the pattern, the developer should select a service if the heartbeat of changes performed in the service interface, and the incremental growth of the service functionalities looks like the heartbeat of a healthy athlete when he is at rest. In particular, the heartbeat of changes performed in the service interface should involve mostly calm periods, in which the service interface does not change, interrupted by spikes of changes that include additions, updates, and rare deletions. The incremental growth of the functionalities offered by the service should also follow this form, consisting mainly of calm periods of maintenance, separated by spikes of growth.* The alteration of calm periods with spikes of changes shows that the service changes to meet evolving requirements. Moreover, the alteration of calm periods with spikes of changes shows that the service evolution combines enhancement and maintenance activities. Due to the spikes of growth, the complexity of the service interfaces increases. However, the calm periods make sure that the increase is smooth. Finally, the combination of calm periods with spikes of growth makes the cumulative growth of the service functionalities predictable and gives time to the developer to learn and master the evolving functionalities.

We structure the remainder of the paper as follows: In Section 2, we discuss related work; in Section 3, we details the setup of our study; in Section 4, we report our findings; in Section 5 we define *the athletic heart syndrome* pattern; in Section 6, we focus on threats to validity; finally, in Section 7, we summarize our findings and provide insights for future work.

2 | RELATED WORK

Service evolution is an active field of research for more than 15 years. For a recent survey of previous efforts in this field, the interested reader can refer to the work of Tran et al.[4] Following, we discuss our contribution with respect to the state of the art on service evolution. We organize the discussion in three parts, based on our research questions and the overall research goal of our study. Specifically, RQ1 and RQ2 concern

service changes and their purpose. Consequently, in Section 2.1, we focus on previous works that concern the detection of service changes and related empirical studies. RQ3 and RQ4 pertain to the impact of changes in the complexity of service interfaces and the possibility of making predictions about service evolution. Hence, in Section 2.2, we discuss approaches that deal with service change management, impact analysis and prediction. Finally, our ultimate research goal is to provide a pattern for the selection of services whose evolutionary behavior facilitates service change management, impact analysis and change prediction. Thus, in Section 2.3, we discuss previous efforts on service evolution patterns.

2.1 | Service change detection and related studies

Service changes can take place at different levels of abstraction. In particular, Tran et al[4] survey previous research efforts that focus on interface, semantic, protocol and process changes. Similar classifications of the different kinds of changes involved in service evolution are provided by Treiber et al[18] and Wang and Wang.[19] Our work is more closely related to previous works that focus on the detection of interface-level changes.

In particular, Fokaefs et al[5] proposed VTracker, a change detection tool for W3C services that detects changes between different releases of a service interface, specified in WSDL. The use of the tool has been validated in real-world services. In the validation of the tool, the authors observed mostly operation additions and updates, while the deletions were relatively few. Romano and Pinzger[6] proposed WSDLDiff that allows a more fine-grained analysis of differences between service interfaces; the prominent features of the tool include the full coverage of the service operations and data types, the filtering of irrelevant changes and the accurate tracking of changes. WSDLDiff has also been validated in real-world services. The validation of the tool provides a detailed view of the changes that occurred in the examined services at the level of operations and data types, along with a frequency analysis of the different types of changes. Romano and Pinzger also found mostly additions and updates and rarely deletions. The approach of Romano and Pinzger[6] is part of a broader ongoing research agenda that targets the change-proneness analysis of service oriented software.[20]

Taking a step further, Fokaefs and Stroulia introduced WSDarwin,[7,21] a change detection tool that extends VTracker. WSDarwin supports both W3C and RESTful services. The change detection process is based on automatically generated WADL specifications. In the case of a W3C service, the WADL specification is generated directly from the respective WSDL service specification. In the case of a RESTful service, the WADL specification is constructed from a given set of URLs of corresponding REST API requests. Hence, a basic requirement for the generation of a complete WADL specification is a complete set of URLs of corresponding REST API requests. Li et al[22] also focused on changes that occur in subsequent releases of RESTful services. In this work, the changes are detected by inspecting REST API reference and migration guides. In their study, the authors observed operation/parameter additions and renamings, parameter and return type changes, some operation deletions that were basically due to the merging of two operations in one, and few operation splits. Espinha et al[23] investigate the evolution of RESTful services from a different perspective. Specifically, the authors interviewed developers to identify problems, due to the evolution of the services that they used. The authors further found that different providers follow different practices and policies for changing services and essential features like versioning are sometimes neglected.

Our work also focuses on the detection of changes in subsequent releases of a service interface. In our study we considered W3C services, because the detection of changes in WSDL specifications is more convenient and accurate.[7] However, this choice does not affect the results that we report in the paper because at the time of the study AWS supported both W3C and RESTful releases of the services and the service releases have been evolving in sync. The results of our study concerning the different kinds of changes that occur during the service lifetimes are in accordance with those of the previous efforts. In particular, the changes in the services are mostly operation/type additions and updates and rarely deletions. However, the overall goal of our work is different from the aforementioned efforts. Specifically, a key contribution of our study is the *athletic heart syndrome* pattern that specifies properties of proper service evolution.

From a broader perspective, our study relates to previous works in the area of schema evolution. In this area we have investigated the evolution of database schemas with respect to Lehman's laws of software evolution.[24] Interestingly, the way that database schemas evolve is similar to that of the services that we considered in our study. Specifically, our results showed that the schemas grow over time. However the growth is usually small, with long calm periods. Concerning the evolution of individual tables,[25] we observed that most tables are subjects to few changes. A possible reason behind the observed similarities in the evolutionary behaviors of the examined services and database schemas could be that in both domains the impact of changes on the dependent systems is high. Nevertheless, a deeper investigation is needed to support this claim.

2.2 | Service change management and prediction

A significant part of the state of the art on service evolution concerns change management and versioning approaches.

One of the early works in this line of research is by Treiber et al.[18] The authors proposed a holistic approach for managing information that concerns different aspects of service evolution like requirements, interface, implementation, and QoS changes. The backbone of the proposed

approach is a unified extensible information model. Leitner et al[26] proposed a mechanism that facilitates the documentation and management of changes. Zou et al[27] generate customized service release notes, per dependent system that uses a service, containing information only about changes performed on the functionalities used by the dependent system.

Andrikopoulos et al[9] proposed a formal framework that allows to perform automatic compatibility checks between different service releases. Khebizi et al[10] focused on service protocol changes. Their framework enables migration from old to new service protocols. The framework relies on a declarative language that lets service providers to specify protocol migration rules for the different releases of the services that they provide. Banati et al[28] proposed a version management framework for maintaining multiple service versions. In a similar vein, the work of Campihnos et al[11] allow multiple service versions to be deployed simultaneously. Service requests coming from dependent systems are redirected to the right service versions that meet specific type safety criteria. As discussed in the vision paper of Baresi and Garriga,[1] the importance of managing multiple coexisting service versions is magnified, as we move from systems that rely on the service-oriented architectural style towards systems that employ the micro-services style. To this end, Sampaio et al[29] proposed a service evolution model for micro-services that combines structural, deployment and runtime information about evolving micro-services. Groh et al[12] also deal with the evolution of micro-services. Specifically, the authors proposed a framework that keeps track of interface, protocol and semantic changes in micro-serves and facilitates the validation of dependent systems against these changes.

Service testing is a key research issue towards making sure that service changes do not introduce new faults that would break the systems that depend on the evolving services. ServicePot[30] and ParTes[31] are two general purpose frameworks for service testing. Nguyen et al[32] proposed a change-driven service testing approach. The basic idea behind the proposed approach is to prioritize the test cases of a service orchestration/choreography, based on their relevance to changes, performed on the composed services.

The co-evolution of service orchestrations/choreographies with the orchestrated/choreographed services[4] is another interesting research issue that involves several different problems. For instance, Baresi et al[13] proposed an approach for the consistent dynamic evolution of service choreographies. The proposed approach allows a service choreography to safely adopt the most recent versions of services that would not break the choreography execution. Moreover, the proposed approach facilitates the retirement of service versions that are no longer needed. Calinescu et al[33] proposed an approach for dynamically evolving service orchestrations that meet reliability and performance requirements. In a similar vein, Lv et al[34] deal with dynamically evolving service orchestrations that meet QoS requirements. Wang et al[14] proposed a distributed knowledge based evolution model for services choreographies that allows to deal with failed or retired services. Autili et al[35] introduced an interesting approach for evolving choreographies with respect to context changes. The proposed approach builds upon a previous work,[36] which enables the dynamic synthesis of service choreographies that satisfy given choreography specifications.

Wang et al[8] make an interesting attempt to predict the evolution of service interfaces, via a machine learning approach. To this end, the authors employed a neural network to predict size metrics like the number of operations, types, bindings, etc. The neural network was trained with size metrics derived from previous service releases. The validation of the approach showed that the predicted values are quite close to the actual ones.

Our contribution is complementary with all the aforementioned efforts. In particular, dealing with service evolution would be easier for systems that depend on services that properly evolve. The *athletic heart syndrome* pattern facilitates the selection of such services.

2.3 | Service evolution patterns

The state of the art on service evolution includes several interesting patterns that document good practices, which aim at making the evolution of services more systematic and easy to handle.

In particular, Wang et al[15] introduced four patterns in their work. The *compatibility pattern* allows developers to determine whether their implementation is compatible with a particular service version that they intent to use. The *transition pattern* specifies a change strategy that minimizes the impact of service changes on the systems that depend on the service. The intent of the *split-map* pattern is to reduce the complexity of a service interface, by splitting it into smaller interfaces, while the goal of *merge-map* is to reduce duplication in service interfaces, by merging similar interfaces.

The work of Lübke et al[16] concerns service evolution patterns that allow to balance compatibility and extensibility, during the lifetime of evolving services. Specifically, the *API description* pattern defines the knowledge that should be shared between the provider of a particular service and the developers of the systems that depend on the service. The *version identifier* pattern allows indicating the current capabilities of a service and possible incompatibilities, while *semantic versioning* facilitates the comparison of different interface versions and the detection of incompatibilities. *Two in production*, is a useful strategy for updating a service interface without breaking the systems that use it. The intent of the *limited lifetime guarantee* pattern is to let the developers of systems that use a service know how long they can rely on a particular service version, while the goal of the *eternal lifetime pattern* is to support developers, who cannot migrate their systems to newer service versions. *Aggressive obsolescence* documents a strategy that reduces the effort and the resources required for the maintenance of prior service versions that are still used.

Finally, the *experimental preview* pattern lets the provider of a service to smoothly introduce a new, possibly immature, service version and get early feedback from developers of systems who are willing to use it.

The *athletic heart syndrome* pattern is different from the aforementioned patterns. The intent of the pattern is to let the developer of a system select services that properly evolve. To this end, the pattern documents service evolution properties that indicate a "healthy" service lifetime.

3 | SETUP

To organize our study, we performed three main steps. In the first step, we identified the set of services that we consider in our study. In the second step, we gathered evolution data and constructed the evolution histories of the examined services. Finally, in the last step, we calculated certain metrics that we consider in our study based on the service evolution histories. Following, we provide further details concerning each one of the aforementioned steps.

3.1 | Identification of the examined services

In our study, we focus on a set of services, provided by the Amazon Web Services (AWS) infrastructure. The basic reasons for this choice are listed below:

- *Popularity:* AWS is a very successful and widely accepted infrastructure that has millions of customers of various types, like NASA, NASDAQ, Netflix, Facebook, Adobe, and D-Link.[2]
- *Longevity:* AWS has been around since the early 00's, with its services being continuously enhanced.
- *Domain coverage:* AWS offers a wide spectrum of services that cover various domains like cloud computing, networking and content delivery, storage and database, management and governance, and application integration.

For the purpose of our study, we selected six services that belong to different domains. The list of services that we consider is given in Table 2. When we started this study, AWS provided both W3C and RESTful releases of the services, which were evolving in sync. Therefore, without loss of generality in our study we decided to focus on the W3C releases of the selected services because their parsable WSDL specifications make the gathering of data concerning the evolution of the services easier.[21] The year of the first release of the examined services varies from 2005 to 2009. The time spans of the services' lives vary from 3 to 8 years, while the number of service releases within these time spans range from 12 to 73.

Table 3 provides descriptive statistics for the distribution of the service releases per year. In particular, the table provides the min, max, average and standard deviation for the number of service releases per year. We observe that for the majority of the services there is a continuous flow of releases per year with an average number of releases per year that ranges from 2.22 to 9.13. The only exception to the rule is MTurk. For this service, the average number of service releases per year is 2.22. However, the minimum number of service releases per year is 0 because there is at least a 2 years pause in the continuous flow of releases. Next, we discuss in more detail the key functionalities, offered by the services.

Elastic Compute Cloud (EC2) allows allocating and managing virtual servers hosted at the Amazon infrastructure. The service provides operations that enable the configuration of several properties of the allocated virtual servers, including their CPU, memory, and storage. Moreover, it provides operations for the deployment of applications that execute on the allocated virtual servers. Elastic Load Balancing (ELB) complements

TABLE 2 List of examined services

Services	1st release	Time span	Releases	Domain
EC2 aws.amazon.com/ec2	2006	8 years	73	Cloud computing
ELB aws.amazon.com/elasticloadbalancing/	2009	5 years	14	Networking and content delivery
AS aws.amazon.com/autoscaling/	2009	3 years	12	Management and governance
SQS aws.amazon.com/sqs/	2006	7 years	16	Application integration
RDS aws.amazon.com/rds/	2009	5 years	41	Storage and databases
MTurk aws.amazon.com/mturk/	2005	8 years	20	Crowd-sourcing marketplace

[2] aws.amazon.com/solutions/case-studies/all/

TABLE 3 Descriptive statistics for the service releases per year

Case study	Min	Max	Avg	Stdev
EC2	2.00	17.00	9.13	5.54
ELB	1.00	6.00	2.80	2.05
AS	1.00	5.00	2.40	1.67
SQS	1.00	5.00	2.67	1.51
RDS	1.00	16.00	8.20	6.61
MTurk	0.00	8.00	2.22	2.82

EC2 with functionalities for balancing the load of virtual servers that have been allocated, using EC2. ELB offers operations that enable the creation of a load balancer, which distributes requests over the virtual servers. Moreover, it allows monitoring the state of the virtual servers and redistributing requests from failed servers to operational servers. Auto Scaling (AS) provides means for increasing/decreasing the number of virtual servers that have been allocated via EC2. The scaling is done automatically, with respect to a given set of conditions. Simple Queue Service (SQS) facilitates application integration via message-based communication. SQS provides operations for creating message queues, sending/receiving messages via queues, managing queue attributes, and configuring queue access rights. Relational Database Service (RDS) provides operations for allocating DB instances. Each DB instance can comprise multiple relational databases, accessed via the DB engine that executes on the DB instance. The RDS service further provides operations for configuring the computation, memory and storage capacity of the allocated DB instances, and for tuning the properties of the DB engines that execute on top of them. Mechanical Turk (MTurk) is a crowdsourcing service that makes it easier for individuals and businesses to outsource their tasks (e.g., simple data validation, survey participation, and content moderation) to a distributed workforce who perform these tasks virtually. The service offers functionalities for creating tasks, allocating the tasks to workers, collecting the results that they produce, approving the results, and paying the workers.

3.2 | Construction of service evolution histories

For each service we manually collected from the AWS site[3] a set of release notes, corresponding to the different releases of the service. The release notes describe the purpose of the changes that have been performed and provide a URL that refers to the WSDL specification of the provided functionalities. Several subsequent service releases may refer to the same WSDL specification, indicating that in these releases the service interface did not change. A service *interface* defines a set of operations. An *operation* is defined in terms of input/output *messages*. A message definition refers to the *data types* that are used to exchange data with the service. All the WSDL specifications that we collected for the examined services are available in our github directory.[37]

At this point, it is worth mentioning that AWS no longer provides access to WSDL specifications. Instead, AWS provides access to programming-language-specific SDKs that offer service-specific APIs. Hence, further service evolution studies would require parsing and analyzing the aforementioned APIs.

To process the data that we gathered from the AWS site we developed a tool that relies on a well-known open source API, called Membrane SOA Model,[4] which allows to parse and compare WSDL specifications. The tool takes as input a set of WSDL specifications that correspond to the different releases of a particular service and produces as output the service evolution history. In brief, the service evolution history provides basic size and change metrics for the operations and the data types, specified in the WSDL specifications. More formally, the concept of service evolution history is defined below.

Definition 1. Service evolution history: For a service s, we define the notion of service evolution history as a list, $H_s = \{r_1^s, r_2^s, \ldots, r_N^s\}$, consisting of the different releases of s, which are totally ordered, based on the release dates.

Definition 2. Service release: We define a service release r_i^s that belongs to the evolution history H_s of a service, s, as a tuple, $r_i^s = (ID, date, Size, Change)$, consisting of the following elements:

- *ID* is a unique release identifier for r_i^s that denotes the order of r_i^s in H_s.
- *date* denotes the release date of r_i^s.

[3] aws.amazon.com.
[4] https://www.membrane-soa.org/soa-model/.

- *Size* is a tuple that consists of the following size metrics:
  - $Size[I]$ gives the number of interfaces defined in the specification of r_i^s.
  - $Size[O]$ gives the number of operations provided by the interfaces of r_i^s.
  - $Size[T]$ gives the number of data types defined in the specification of r_i^s.
- *Change* is a tuple that comprises the following change metrics:
  - $Change[O_A]$, $Change[O_D]$, $Change[O_U]$ measure the number of operations that have been added, removed, and updated, respectively.
  - $Change[T_A]$, $Change[T_D]$, $Change[T_U]$ measure the number of data types that have been added, removed, and updated, respectively.

3.3 | Service evolution metrics

In our study, we employ service evolution metrics for the *incremental growth* and the *cumulative growth* of service operations and data types. Moreover, we assume a metric for the *complexity* of service interfaces.

Definition 3. Incremental growth: We define the incremental growth of service operations (resp. data types) $IG_{op}(r_i^s, r_{i+1}^s)$ (resp. $IG_t(r_i^s, r_{i+1}^s)$) for a pair of subsequent service releases r_i^s, r_{i+1}^s that belong to the evolution history, H_s, of a service, s, as the difference in the number of operations (resp. data types) defined in the specification of r_{i+1}^s and r_i^s, that is, $IG_{op}(r_i^s, r_{i+1}^s) = r_{i+1}^s.Size[O] - r_i^s.Size[O]$ (resp. $IG_t(r_i^s, r_{i+1}^s) = r_{i+1}^s.Size[T] - r_i^s.Size[T]$).

The cumulative growth metrics that we assume for the operations and data types are defined below.

Definition 4. Cumulative growth: We measure the cumulative growth of the operations (resp. data types) $G_{op}(r_i^s)$ (resp. $G_t(r_i^s)$) for a service release r_i^s that belongs to the evolution history, H_s, of a service, s as the number of operations (resp. data types) defined in the specification of r_i^s, i.e., $G_{op}(r_i^s) = r_i^s.Size[O]$ (resp. $G_t(r_i^s) = r_i^s.Size[T]$).

In prior studies,[17,24,38–40] Lehman and his colleagues showed that system growth can be estimated, via a feedback-based growth prediction formula, known as the inverse square (IS) model. In our study, we use this model to calculate predictions for the cumulative growth of service operations. To this end, we adapt the IS growth prediction formula, with respect to our definition of cumulative growth.

Definition 5. IS model for services—According to the IS model, the predicted cumulative growth of the operations, $\widehat{G_{op}}(r_i^s)$, provided by a service release r_i^s that belongs to the evolution history, H_s, of a service, s, is: $\widehat{G_{op}}(r_i^s) = \widehat{G_{op}}(r_{i-1}^s) + \frac{\bar{E}}{\widehat{G_{op}}(r_{i-1}^s)^2}$, where $\widehat{G_{op}}(r_{i-1}^s)$ is the estimated cumulative growth of the operations, provided by the previous service release, r_{i-1}^s, and $\bar{E}$ estimates evolution effort. More specifically, $\bar{E}$ is the average of individual E_j, calculated for the service release history H_s, as follows: $E_j = (G_{op}(r_j^s) - G_{op}(r_{j-1}^s)) * G_{op}(r_{j-1}^s)^2$, where $G_{op}(r_j^s)$ is the actual cumulative growth of the operations, provided by a service release r_j^s, and $G_{op}(r_{j-1}^s)$ is the actual cumulative growth of the operations, provided by the previous service release r_{j-1}^s.

Concerning the complexity of service interfaces we assume the metric that is defined next.

Definition 6. Complexity: For a service release r_i^s that belongs to the evolution history, H_s, of a service, s, we measure complexity as the complement of the number of interfaces, defined in the specification of r_i^s, divided by the number of operations, provided by the interfaces of r_i^s, i.e., $C(r_i^s) = 1 - \frac{r_i^s.Size[I]}{r_i^s.Size[O]}$.

The metric is inspired by the generic approach that has been proposed by Sneed,[41] for measuring the structural complexity of service interfaces, with respect to the entities and relations defined in their specification. Moreover, the metric is based on our prior work on the systematic decomposition of service interfaces,[2] where we empirically found that developers prefer smaller fine-grained interfaces that consist of few operations which focus in specific functionalities, to larger coarse-grained interfaces that comprise lots of operations. Essentially, the value of the metric is high if a service provides lots of operations, grouped in few interfaces. The metric takes its maximum value if all the operations are grouped in a single interface; the larger the number of operations, the larger the value of the metric.

4 | FINDINGS

In this section, we describe the research findings of our the study over the collected service histories.

4.1 | Contextualization and theoretical foundations

Before proceeding with the research questions and the produced answers, we believe it is important to structure the theoretical framework of our research. This subsection is devoted to establishing the rationale behind our research questions, and how they fit together.

Our theoretical framework over which we base this study is the one provided by Easterbrook et al.[42] The framework of Easterbrook et al. is basically structured along three fundamental categories of research questions that an empirical study can pose.

1. The first category of research questions are *exploratory* questions, trying to uncover the components, the properties, and the base rates and normal behavior of the studied phenomena.
2. The second category of research questions are *relationship & causality* questions, trying to find out (a) what causes phenomena to occur, and (b) whether different events frequently co-occur, relate or cause each other.
3. The third category of research questions are *design* questions, trying to move from knowledge extraction to normative guidelines on how to best design well behaved systems, or, how to achieve their good properties.

Placing our work in the context of the above framework, we try to cover all the parts of the framework via different questions that we pose, along with the methods that we use in order to answer them. In a nutshell, our research strategy evolves as follows.

Exploratory questions. Our starting point is, naturally, exploratory questions. As Easterbrook et al. mention,[42] these questions are along the lines of asking "does a phenomenon X occur?", "what are the components and properties of phenomenon X?", "how do we measure X?", "what are the frequency and base rates of X?". In our case, of course, the phenomenon X is service evolution. Our research question RQ1, mainly addresses the family of questions "how does service evolution look like?" and "are there commonalities in the evolution of different services?". In more concrete terms, we discuss the main components of service evolution (namely operation and data type evolution), the main measurements of interest (change breakdown in additions, deletions and updates, both overall and also by component type, as well as the heartbeat, incremental and cumulative growth, and, interface complexity). Mean, minimum, and maximum values are also of interest to establish base rates. RQ2 is mainly concerned with how exactly the service evolution took place and the specific types of changes that took place.

Relationship and causality questions. To move one step further than simple exploratory questions, we proceed to examine the impact of change. RQ3 studies the correlation of change to the complexity of the service interfaces. RQ4 is a bridge between the knowledge extraction and normative families of questions, as it is concerned with the prediction of how a service will evolve. Based on the feedback characteristics of Lehman's laws for evolving systems, the motivation is to investigate whether we can correlate the amount of past change to the amount of future change in a system. To a large extent, this is an (auto) correlation question; at the same time, it opens the door for the next category of inquiries.

Normative, design questions. This family of questions is addressed in the following Section, when the Athletic Heart Syndrome pattern is established. "How will I know a healthy service when I see it?" is the question of interest, as we are mostly interested with the client developer of services, rather than the designer, tester or implementer of them. This normative question was not originally intended when the research study of the collected services was performed. However, once the similarities of the service evolution in different histories was evident, a normative research question was the obvious step. Our results for this family of questions are laid out in the following section.

So, having established our theoretical framework, we are now ready to move on to examine each research question, and its findings, one by one.

4.2 | RQ1: **Is there a pattern in the evolution of the services?**

To address this research question, we study the heartbeat of changes performed in the interfaces of the examined services and the incremental/cumulative growth of the functionalities that they provide.

Figure 1 gives a detailed view of the changes that have been performed in the operations of the examined services. Each bar chart corresponds to a particular service; each bar gives the cumulative number of changes introduced in a service release, along with the statistical breakdown of the changes, with respect to their type (operation additions, deletions, updates). In the figure, we observe that in many service releases the service interface does not change. Calm periods consisting of subsequent service releases that preserve the service interface are interrupted by releases with spikes of changes. In particular, AS and SQS are *very quiet*, with the percentage of service releases that involve spikes of changes being 8% and 9%, respectively. The rest of the services, namely EC2, ELB, RDS, and MTurk, are *more active*. In these services, the percentage of

releases with spikes of changes ranges from 22% to 67%. The overwhelming majority of the changes are additions and updates. This observation is inline with previous studies that report similar results.[5,6] The deletions of operations are rare. The percentage of deletions, over the total number of changes, varies for the examined services from 0% to 6.8%.

Concerning the changes in the data types of the examined services, the situation is much similar. Operation additions, deletions and updates are strongly correlated with data type, additions, deletions, and updates, respectively. Specifically, in Table 4, we observe that the values of Spearman's correlation coefficient between these variables range in [0.97, 1], [0.62, 1], and [0.80, 1].

Figure 2 gives the incremental growth of the operations provided by the examined services. In this figure, we also observe calm periods of zero growth, interrupted by scarce spikes of positive growth. For the quiet services (AS, SQS) we observe only a couple of spikes separated by long calm periods. On the other hand, for the active services (EC2, ELB, RDS, MTurk), we have more spikes, separated by shorter calm periods. The behavior of MTurk is slightly different from the rest of the active services, as we have subsequent spikes, and only one short calm period.

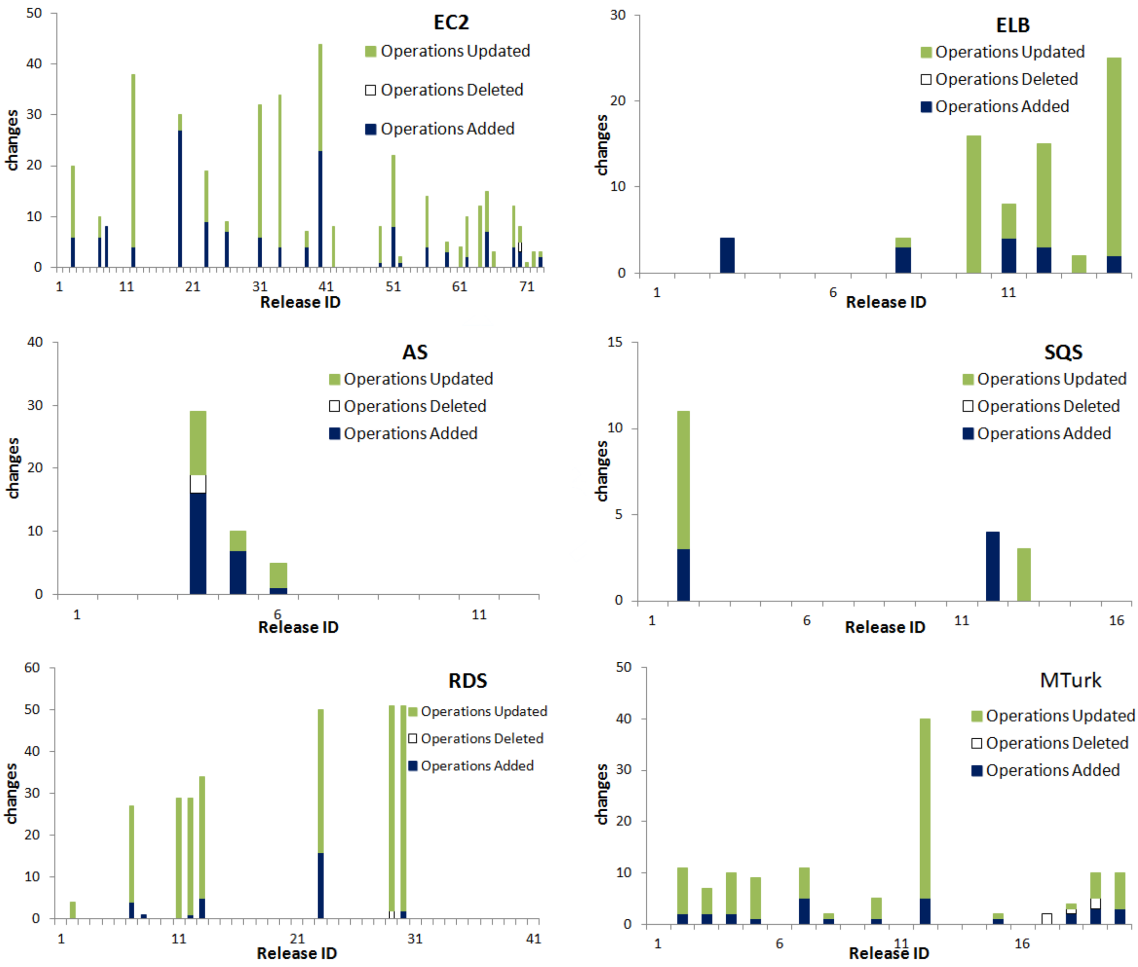

FIGURE 1 Operation additions, deletions and updates per service release

TABLE 4 Spearman's ϱ between operation and data type changes

Services	Additions	Deletions	Updates
EC2	0.98	0.62	0.94
ELB	0.97	-	0.96
AS	1.00	1.00	1.00
SQS	1.00	-	1.00
RDS	0.99	1.00	0.80
MTurk	0.98	1.00	0.99

Figure 3 gives the cumulative growth of the operations provided by the examined services. In all cases, we observe a increasing trend. However, we also observe the alteration between calm periods and periods of growth. In the case of the quiet services (AS, SQS), the cumulative growth of the operations occurs in a few abrupt steps, while in the active services the cumulative growth of the operations is more smooth, happening in multiple smaller steps.

The incremental growth and the cumulative growth of data types evolve similarly with the incremental growth and the cumulative growth of operations, respectively. In particular, Table 5 gives the values of Spearman's correlation coefficient for IG_{op} and IG_t. As we see, for most services the correlation is very strong, ranging from 0.96 to 1.0. The only exception is EC2, where IG_{op} and IG_t are also positively correlated, but the strength of the correlation is moderate. Table 5 further provides the values of Spearman's correlation coefficient for G_{op} and G_t. Again, the correlation is very strong, ranging from 0.80 to 1.0.

Hence, our results show that all the services follow the same evolution pattern. According to this pattern, *the heartbeat of changes consists mostly of calm periods, in which the interface does not change, separated by spikes of changes that involve additions, updates and, rarely, deletions*. Respectively, *the incremental growth of the service functionalities involves an alteration of calm periods, in which the service functionalities do not expand, with spikes of growth*.

Hereafter, we call the aforementioned pattern, *the athletic heart syndrome*, to reflect the analogy between the calm periods and the spikes of changes/growth that we observe in the evolution of the examined services and the heartbeat of a healthy athlete when he is at rest. Typically, an ordinary person's heartbeat should be strong and regular. A slower heartbeat, called *bradycardia* in medical terms, may indicate a medical problem. However, for an athlete, bradycardia is an indication of a good physical condition. In the context of service evolution, the spikes of changes show that the examined services continually adapt to satisfy evolving requirements. The spikes of growth show that the services are continually enhanced with new functionalities. Calm periods allow the developers of dependent service-oriented systems to absorb changes that occur during the adaption of the services and familiarize with new functionalities added during the growth of the services.

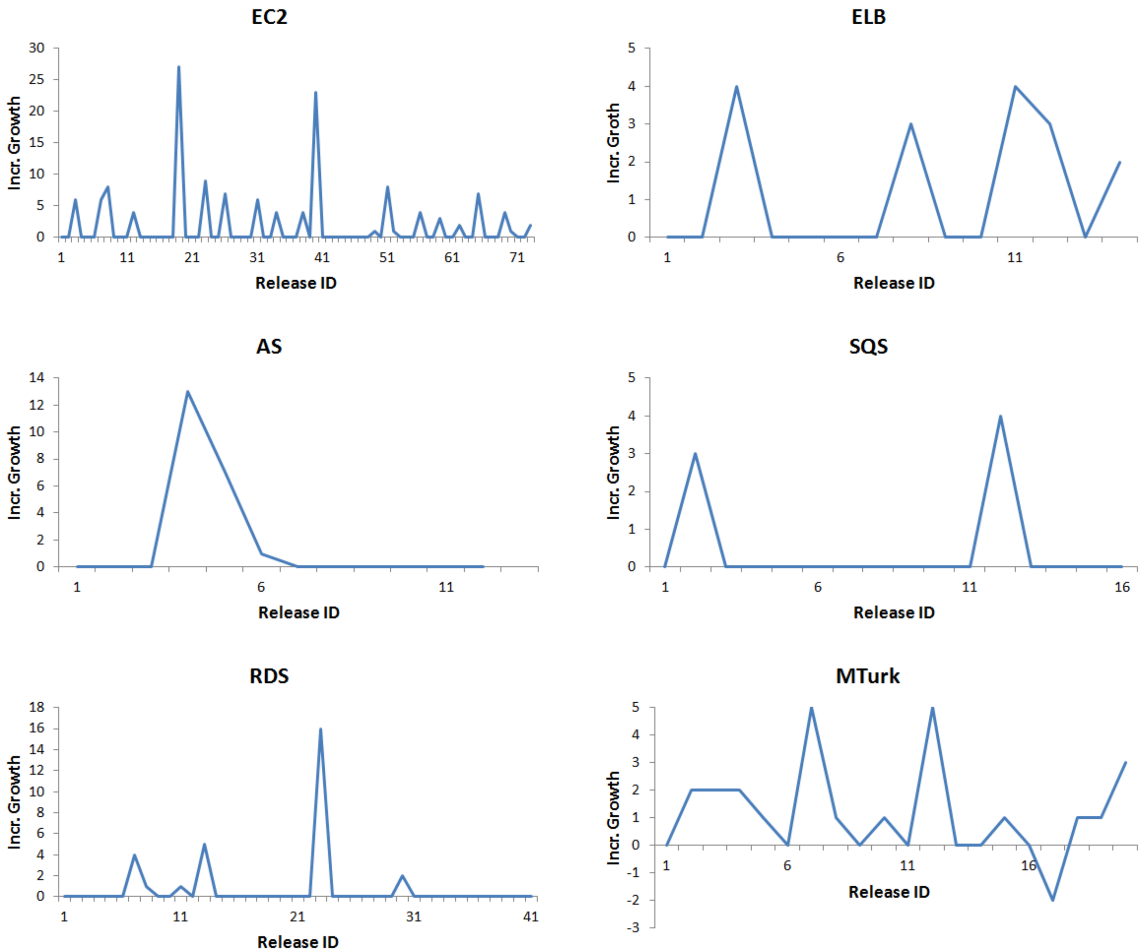

FIGURE 2 Incremental growth of service functionalities, measured as the difference in the number of operations offered by subsequent service releases

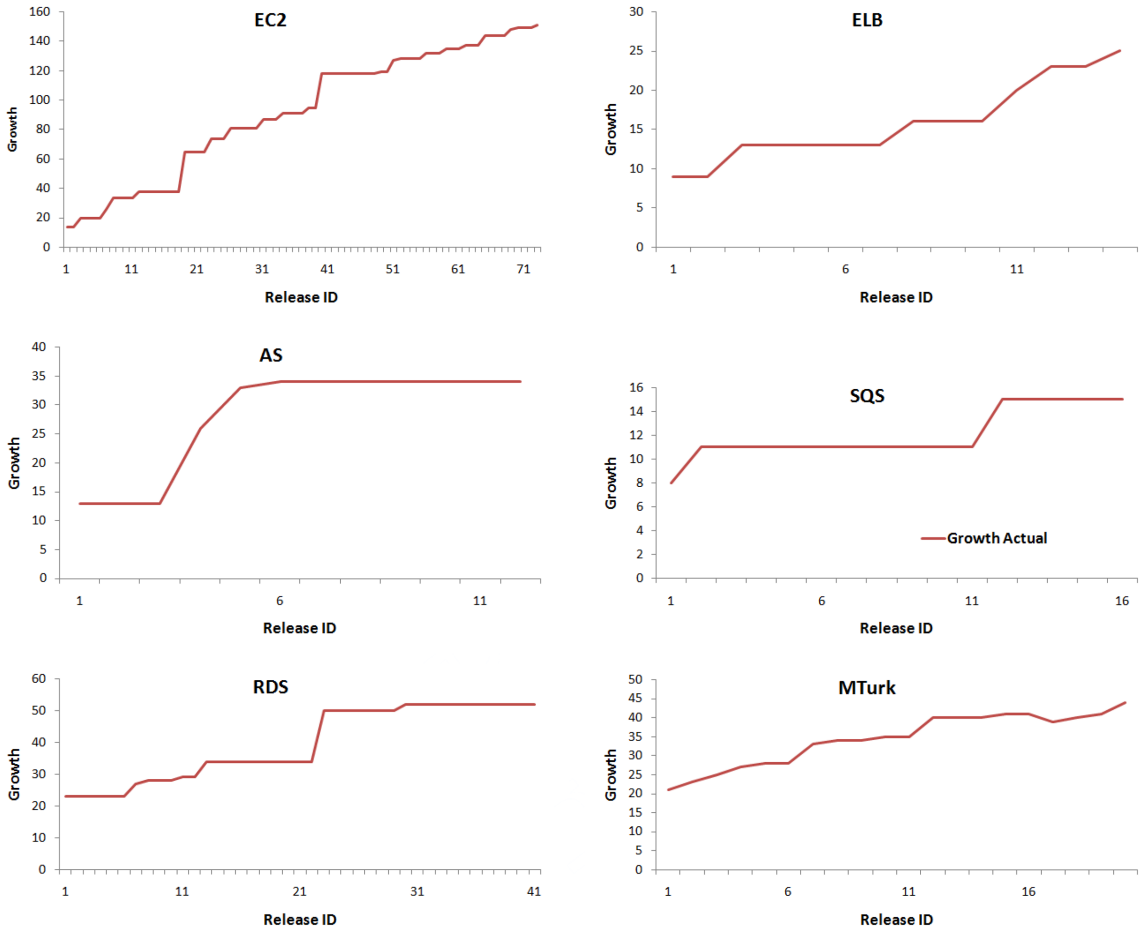

FIGURE 3 Cumulative growth of service functionalities, measured in terms of the number of operations, provided by subsequent service releases

TABLE 5 Spearman's ϱ between (a) the incremental growth of operations and data types (IG_{op}, IG_t) and (b) the growth of operations and data types (G_{op}, G_t)

Services	ϱ for IG_{op}, IG_t	ϱ for G_{op}, G_t
EC2	0.50	0.99
ELB	0.97	0.99
AS	1.00	1.00
SQS	1.00	1.00
RDS	0.99	0.99
MTurk	0.96	0.99

4.3 | RQ2: **What is the purpose of the service evolution?**

To find out more about the purpose of the changes that have been performed in the examined services, we study in more detail the service release notes that we gathered from the AWS site. To begin with, we classify the service releases in four different categories, based on the purpose of the changes that have been performed:

- **Functionality Growth/Update (FGU)**: The first category includes releases that focus on the growth and/or the update of the provided functionalities.
- **Service Platform Enhancements (SPE)**: The second category consists of releases that concern the provisioning of SDKs for various programming languages and service management tools.

- **Service Infrastructure Enhancements (SIE)**: The third category comprises releases that improve the underlying infrastructure with new or upgraded OSs, DBMSs, hardware and so on.
- **Service Regions and zones (SRZ)**: The fourth category includes releases that deal with service availability in various geographical regions and zones.

At this point, we must note that FGU releases involve spikes of interface-level changes, while SPE, SIE and SRZ releases concern maintenance activities that leave the service interface unchanged. Tables 6-11 provide summaries of the release notes. The white rows provide information about FGU releases, while the light red rows provide information about calm periods that consist of SPE, SIE, and SRZ releases. The last column of the table refers to the categories in which the service releases belong to. Following, we discuss in more detail the purpose of the changes that have been performed in the case of each service.

Regarding EC2, the top five releases, with regard to the number of changes performed in the service interface, are R40, R12, R34, R31, and R19. R40 is the release with the highest number of changes. The purpose of these changes is to add support for the EC2 Virtual Private Network (VPN) internet gateway. The main reason for the changes that took place in R12 is to add support for the specification and management of EC2 bundle tasks, while R34 introduces changes that allow tagging and filtering of EC2 resources. In R32, the purpose of the changes is to enable EC2 placement groups, while in R19, the changes support the Amazon Virtual Private Cloud (VPC). In several calm periods (e.g., R4–6, R13–18, R32–33, R57–58, and R63), the goal is to provide different kinds of EC2 instance types. Other calm periods (R20–22, R24–25, R39, R41, R73), include releases that aim at supporting various OSs and technologies (e.g., IBM software, Windows, Oracle software, and Linux). Moreover, some calm periods (e.g., R27–30 and R35–37) include releases that provide service programming support for programming languages like Java and PhP), as part of corresponding AWS SDKs. Finally, there are calm periods (e.g., R20–22 and R27–30) with releases that deal with the service availability in various regions and zones.

In the case of ELB, there are seven releases that introduce changes in the service interface. In particular, R14 comes with the largest amount of changes to enable cross-zone load balancing. Then, we have R10 that enables new features, related to IPv6 and application instance lock down, and R12 which adds support for using ELB in Amazon VPC applications. R11 provides new security features. Finally, R3 and R8 involve fewer changes to support ELB sticky sesions and the creation/deletion of load balancer listeners, respectively. The calm periods (R1-2, R4-5, R9) include releases that focus on the .NET SDK and the availability of ELB in various regions and zones.

In AS, R4 is the release with the highest number of changes in the service interface. The purpose of the changes is to enable various features like auto scaling tag groups and instances, management of scaling triggers and so on. R5 and R6 involve fewer changes to support Amazon SNS notifications and instance management policies. The calm periods comprise releases concerning the .NET SDK, support for VPCs, monitoring, console management, etc.

In the case of SQS, R2, R12, and R13 involve changes in the service interface. R2 comes with the highest number of changes to support shared queues, anonymous access to queues, visibility and permission management. R12 and R13 introduce changes for new endpoints and timers. The calm periods (R1, R3-11, R13) include releases concerning SDKs for different programming languages like Java and PhP, releases dealing with the availability of SQS in several regions and zones, monitoring, console management, and so forth.

Concerning RDS, R29, R30, and R23 involve relatively high numbers of changes in the service interface, to support, respectively, file log access, DB instance management and DB options groups. R13, R12, R11, and R7 introduce fewer changes, focusing on DB instance management, while R2 and R4 have very few changes, concentrating on DB instance deployment and replication issues. Several calm periods (e.g., R14–22, R24–28, and R30) comprise releases that aim at supporting various DBMSs (e.g., SQL server, Oracle, and MySQL). Other calm periods (e.g., R1, R3-6) include releases concerning the Java SDK, the availability of RDS in various regions and zones, and console management.

In the case of MTurk, R12 introduces the highest number of changes in the service interface, to add new features for the management of workers and the design of Human Intelligence Tasks (HITs). Then, there are several releases (e.g., R2, R7, R4, R19, and R20) with moderate changes focusing on the management of HITs. Finally, there are releases with very few changes (e.g., R10, R15, and R16) which are also related to the management of workers and HITs. The calm periods mainly consist of releases that deal with the MTurk Web site (e.g., R6, R13–14, and R20) and the payment of workers (e.g., R13–14 and R16).

To summarize, the information provided in the release notes of the examined services shows that *the athletic heart syndrome involves both functional growth and maintenance activities*. The calm periods of the pattern *concern mainly adaptive*[5] *and perfective*[6] *maintenance activities that improve the service programming platform, the service infrastructure and the service availability in various regions and zones. In the service release notes there is no evidence of perfective maintenance activities aiming at re-structuring and maintainability improvements. Moreover, in the service release notes there is not much evidence regarding corrective*[7] *maintenance activities*. However, these observations do not really mean that the evolution of the services does not include such kind of maintenance activities. In general, internal service implementation corrections and improvements are

[5]Adaptive maintenance concerns modifications performed to deal with changes in the environment (e.g., upgrading OSs, libraries, and porting to new platforms).[43]
[6]Perfective maintenance focuses on modifications that improve documentation, performance, maintainability and other qualities.[43]
[7]Corrective maintenance aims at correcting discovered faults.[43]

TABLE 6 EC2 summary of release notes

EC2			
RID	**Operation changes**	**Description**	**Class**
R1-2		First release of the service, support for reserved instances, IBM software, AWS .NET SDK, ...	SPE & SIE
R3	20	Support for rebooting instances and configuring instance attributes	FGU
R4-6		Support for various instance types	SIE
R7	10	Description of availability zones and address management	FGU
R8	8	Volume and snapshot management	FGU
R9-11		Support for SLAs, Microsoft windows and SQL server	SIE
R12	38	Support for specifying and managing task bundles	FGU
R13-18		Support for new instances types, AWS SDKs,	SPE & SIE
R19	30	Support Amazon Virtual Private Cloud	FGU
R20-22		Porting to IBM software and availability of reserved instances in Europe	SIE & SRZ
R23	19	Introduces Amazon Machine Instances (AMIs) backed by Amazon Elastic Block Storage	FGU
R24-25		Support for Windows 2008	SIE
R26	9	Introduces spot instances	FGU
R27-30		New AWS SDK for Java, support for Asia-Pasific region, monitoring capabilities, high memory instance types	SPE, SIE & SRZ
R31	32	Placement groups and licences	FGU
R32-33		Introduces new cluster instance types	SIE
R34	34	Tags, filters and idempotent run instance calls	FGU
R35-37		New AWS SDK for PhP, support for basic monitoring, new cluster GPU instance type	SPE & SIE
R38	7	Support for VM import	FGU
R39		Support for Oracle applications	SIE
R40	44	Support for VPC internet gateway	FGU
R41		Windows server 2008 R2	SIE
R42	8	Support for instances launched in VPC	FGU
R43-48		IPv6, pricing changes, VM import CLI updates	SIE & SRZ
R49	8	Describe Instance Status API	FGU
R50		New choices of reserved instance offerings	SIE
R51	22	Support for New Elastic Network Interfaces (ENIs) for EC2 Instances in a VPC	FGU
R52	2	Support for Amazon instance status checks	FGU
R53-55		Introduce 64-bit option for all Amazon Machine Images	SIE
R56	14	Support for AWS Marketplace	FGU
R57-58		Introduce support for large instances in Amazon Virtual Private Cloud (Amazon VPC)	SIE
R59	5	Support for exporting Windows Server instances	FGU
R60		Introduces several new Spot Instance, new features include integration with Auto Scaling and AWS CloudFormation.	SPE & SIE
R61	4	AWS Identity and Access Management (IAM) roles on EC2 instances.	FGU
R62	10	Support multiple IP addresses for Amazon EC2 instances in Amazon.	FGU
R63		Support for high I/O instances	SIE
R64	12	Introduces a new Amazon Elastic Block Store (EBS) . This release also introduces the ability to launch selected Amazon EC2 instance types as EBS-Optimized instances.	FGU
R65	15	Introduces a new Reserved Instance Marketplace.	FGU
R66	3	Introduces the new M3 extra-large and M3 double-extra-large instance types. These instance types are only available in the US East (N. Virginia) Region.	FGU
R67-68		Increased maximum number of IOPS (input/output operations per second) that you can provided for an Amazon EBS volume, introduced the ability to copy an Amazon Elastic Block Store (Amazon EBS) snapshot	SIE
R69	12	Introduces the ability to copy an Amazon Elastic Block Store (Amazon EBS) snapshot.	FGU
R70	8	Copying an AMI from one AWS region to another and support for default virtual private clouds (VPC).	FGU
R71	1	Creating IAM policy statements that apply to particular Amazon EC2 resources.	FGU
R72	3	Controlling the assignment of public IP addresses to instances launched in a VPC.	FGU
R73	3	Support for Reserved Instance type modifications.	FGU
> R73		AMI Linux, new launch manager, support for new regions and new kinds of instances, usage reports, automatic instance recovery, console enhancements ...	SPE, SIE & SRZ

immediately available to the dependent systems that rely on the services, without having to modify them or explicitly notify the developers of these systems via a new service release.

4.4 | RQ3: **What is the impact of the service evolution on the complexity of the service interfaces?**

Figure 4 gives the complexity of the interfaces, provided by the examined services. In general, we observe that *the complexity is high in all cases*. The complexity remains high for the entire evolution history of all services. There are no indications of any complexity control activities. In particular, in RQ1 we saw that the deleted operations are very few. Moreover, the possibility of interface decomposition is not exploited, as the number of service interfaces usually remains unchanged for the entire evolution history of the examined services.

TABLE 7 ELB summary of release notes

ELB			
RID	**Operation changes**	**Description**	**Class**
R1-2		First release of the service, new SDK for .NET, updates for US West region	SPE & SRZ
R3	4	Support for sticky sessions	FGU
R4-7		Support for the Asia Pacific (Singapore) region, AWS Identity and Access Management (IAM)	SRZ
R8	4	Creation/deletion of load balancer listeners and https	FGU
R9		Update for X-Forwarded-Proto and X-Forwarded-Port headers.	SIE
R10	16	Support for IPv6, zone apex domain names, application instance lockdown	FGU
R11	8	Secure communication from the corresponding Elastic Load Balancer to application servers, health checks for back-end server using HTTPS, back-end server authentication	FGU
R12	15	Support for using Elastic Load Balancing in Amazon VPC applications	FGU
R13	2	Introduce Internal Load Balancing in Amazon Virtual Private Cloud (Amazon VPC).	FGU
R14	25	Provides means to enable Cross-Zone Load Balancing	FGU
> R14		Support for access logs, connection draining, AWS certificate manager, tagging, disync mitigation mode, ...	SPE, SIE & SRZ

TABLE 8 AS summary of release notes

AS			
RID	**Operation changes**	**Description**	**Class**
R1-4		Initial release of the service, AWS SDK for .NET, updates for US West region	SPE & SRZ
R4	29	Auto scaling tag groups and instances, removal of scaling triggers creation or update	FGU
R5	10	Support for Amazon SNS notifications	FGU
R6	5	Description of instance termination policy types	FGU
> R6		Support for public Ips for VPCs, console support, account monitoring	SIE & SRZ

TABLE 9 SQS summary of release notes

SQS			
RID	**Operation changes**	**Description**	**Class**
R1		Initial release of the service, bug fixes, support for large messages	SIE
R2	11	Added shared queues and anonymous access, new valid values for parameters in two operations, support for visibility and permission management.	FGU
R3-11		New regions, AWS SDKs for PHP and Java, AWS console support	SPE
R12	4	New endpoints	FGU
R13	3	Support for new timers	FGU
> R13		New regions in US, Asia, Europe, new console design, support for new AWS SDKs	SPE & SRZ

TABLE 10 RDS summary of release notes

RDS			
RID	**Operation changes**	**Description**	**Class**
R1		Initial release of the service, new AWS SDK for Java, support for new regions in Asia, US	SPE & SRZ
R2	4	Support for multi-AZ deployments	FGU
R3-6		AWS console support	SPE
R7	27	Reserved instances, DB version management, identity access management (AMI)	FGU
R8	1	Support for read replicas	FGU
R9-10		AWS console support for reserved instances and read replicas	SPE
R11	29	Updates in the API and support for Oracle database engine	FGU
R12	29	Introduce new types of reserved DB instance offerings	FGU
R13	34	Support for running DB instances in Amazon VPC	FGU
R14-22		Support for MySQL server, Microsoft SQL server	SIE
R23	50	Provisioning of IOPS, options groups, tags.	FGU
R24-28		Support for Oracle micro DB, SSL for SQL server DB instance, new instance types	SIE
R29	51	DB instance notifications subscriptions, instance renaming and security group migration.	FGU
R30	51	Support for log file access	FGU
> R30		Upgrades of SQL server, MySQL, PostgreSQL, advanced Oracle security features, Oracle statspack, ...	SIE

TABLE 11 MTurk summary of release notes

MTurk			
RID	**Operation changes**	**Description**	**Class**
R1		Initial release of the service	SPE
R2	11	HIT types, reviewing HITs, updates to HIT qualifications	FGU
R3	7	Forced HIT expirations and HIT search	FGU
R4	10	HIT notifications multimedia content in HITs, locale based qualification requirements	FGU
R5	9	Support for bonus payments and private HITs	FGU
R6		XHTML formatted content and external Web sites question hosting	SPE
R7	11	Support for getting HITs per qualification type, getting qualifications per qualification type, revoking qualifications, rejecting qualifications	FGU
R8	2	Support for querying bonus payments	FGU
R9		Adding the ability to assign qualification to worker without a qualification request	SPE
R10	5	Support for configuring the number of assignments a worker can accept for HITs	FGU
R11		New developer sandbox for testin Mturk applications in a controlled environment	SPE
R12	40	New features for blocking/unblocking workers and HITs design	FGU
R13-14		New Web site functionalies for payment options and transaction history	SPE
R15	2	Improvements to the worker notification operations	FGU
R16		Payment improvements for US workers with deposits to Amazon payment personal accounts	SPE
R17	2	Support for removing qualification types	FGU
R18	4	Removal of operations for getting and setting worker accept limits and addition of new qualification feature regarding worker number of approved HITs	FGU
R19	10	Support for new features concerning Mturk review policies	FGU
R20	10	New features for getting assignments, approving rejected assignments and getting list of blocked workers	FGU
> R20		Enhancements to Web site functionalities	SPE

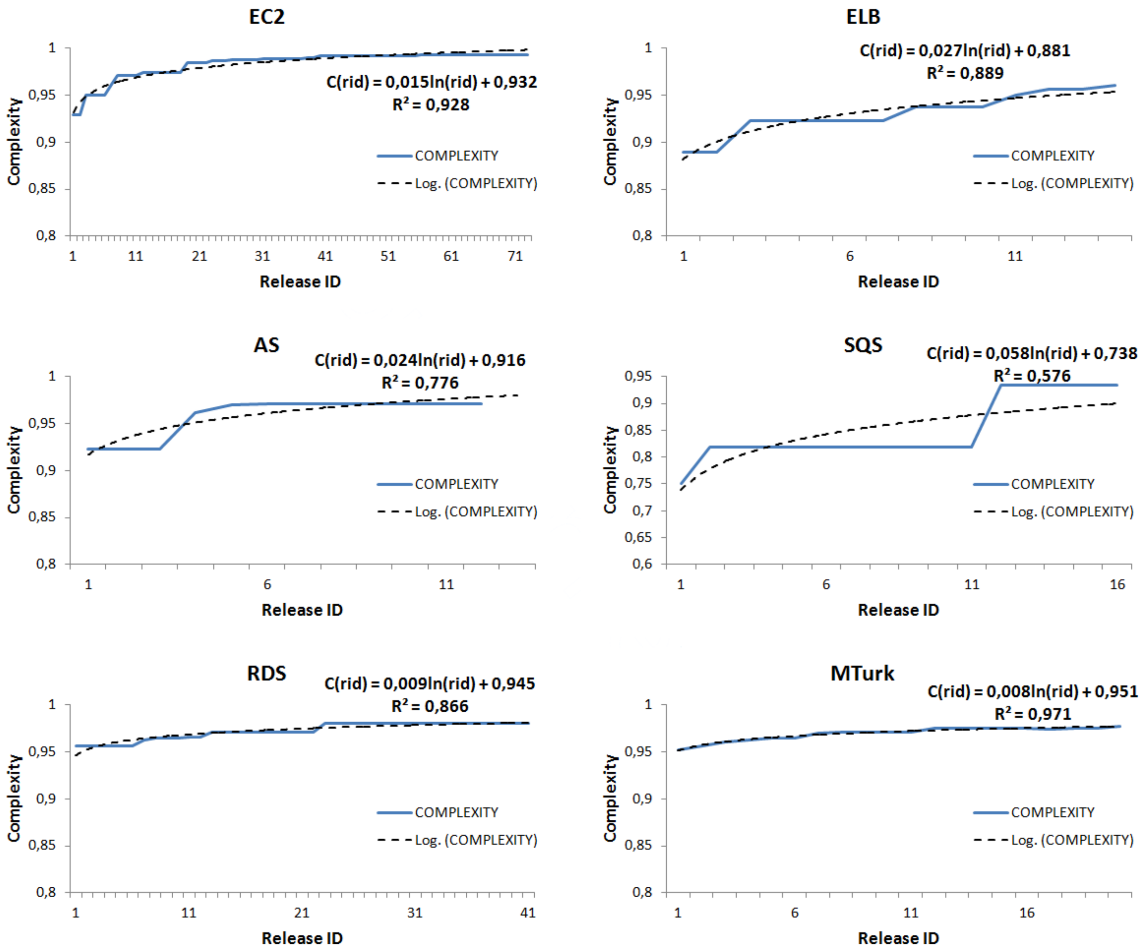

FIGURE 4 Complexity of service interfaces per service release

In fact, during the life of the services we observe a complexity increase. The complexity increases slowly with a logarithmic trend. To provide more insight on this observation we performed a logarithmic regression analysis. The regression formula that we obtained for each service is given at the upper right corner of the corresponding line chart, along with the value of the R^2 statistic. In general, the values of the R^2 statistic range from 0 to 1; high R^2 values indicate that a regression equation explains well the relationship between the variables involved in the equation. For the examined services, the R^2 values that we obtained are medium-high, confirming the observed logarithmic trend. More specifically, for the active services (EC2, ELB. RDS, MTurk) in which we have a relatively large percentage of releases with changes in the service interfaces (RQ1), we observe very large R^2 values ranging between 0.86 and 0.97. For the quiet services (AS, SQS), the R^2 values are medium high (0.77 and 0.57, respectively). The main reason that the complexity increase is smooth is that the incremental growth of the services follows *the athletic heart syndrome* pattern and specifically the frequent calm periods that separate the spikes which introduce the expansion of the service functionalities.

In summary, *the complexity of the interfaces that are provided by the examined services starts high and smoothly increases during the evolution of the services*. Practically, this means that learning to use a service for the first time shall be the most difficult step for the developers of service-oriented systems. After this step, dealing with the smooth complexity increase will most likely not be a problem for the developers. The fact that the incremental growth of the services follows *the athletic heart syndrome* pattern contributes to this direction, as the alteration of spikes with calm periods can make the comprehension and the adoption of the performed changes easier.[3,17]

4.5 | RQ4: **Is it possible to make predictions about the evolution of the services?**

To begin, we focus on the heartbeat of changes, performed in the interfaces of the examined services. In the detailed analysis of the changes performed in the service operations that is given in Figure 1 we see that there is *a large fluctuation in the number and the types of changes performed in the different releases of each service*. This observation holds both for the quiet (AS, SQS) and the active services (EC2, ELB, RDS, MTurk). As expected, for the changes performed in the data types the situation is similar, as derived by the high values of the Spearman's correlation coefficient, between them and the changes performed in the operations, given in Table 4. Hence, it is not really possible to forecast the amount and the types of changes that occur in the interfaces of the examined services,

Taking a different direction, we investigate the possibility of making predictions for the cumulative growth of the functionalities that are provided by the services. Specifically, we employ the IS model, as adapted to our context in Section 3.3, to calculate the estimated cumulative growth of the operations, provided by the examined services. Then, we compare the estimated cumulative growth with the actual cumulative growth. To compare the estimated with the actual values we rely on the R^2 statistic. For the calculation of the average evolution effort $\bar{E}$ that is required by the IS feedback formula, we assume 4 different variants:

- In the first variant, denoted by $\tau = all$, we compute the average evolution effort $\bar{E}$, by taking into account all the releases that are included in the service evolution history. The first variant is inspired by the original IS model introduced by Lehman.[39]
- In the remaining three variants, we calculate the average evolution effort $\bar{E}$, based on a certain number of recent service releases. In particular, in $\tau = 2, \tau = 3$ and $\tau = 4$, we calculate $\bar{E}$, based on the previous 2, 3, and 4 service releases, respectively.

Figure 5 illustrates the comparison between the predicted and the actual cumulative growth. In general, the accuracy of the predictions depends on two different factors, the evolutionary behavior of the services and the number of service releases that are used for calculating the average evolution effort $\bar{E}$. In detail, we observe that typically the model predictions for the active services (EC2, ELB, RDS, MTurk) are better than the model predictions for the quiet services (AS, SQS). As discussed in RQ1, the reason for this is that the growth of the active services is more smooth compared to the growth of the quiet services that happens in a few abrupt steps. Moreover, for most services the accuracy of the predictions increases as we decrease the number of previous service releases that we consider for the calculation of the average evolution effort $\bar{E}$. In fact, $\tau = 2$ gives the best estimations in all cases, but MTurk where $\tau = all$ outperforms the other variants.

To summarize, *forecasting the heartbeat of changes is not likely*. Consequently, the accurate planning of the resources that should be spend for dealing with the changes is also not possible. However, *the IS model gives quite good predictions for the cumulative growth of the operations, provided by the examined services*. In particular, we can safely conclude that *the growth of the service operations can be predicted with accuracy based on a feedback-based formula that considers changes in previous service releases*. The developers of service-oriented systems can benefit from such predictions towards planning the resources that they will need for learning new functionalities and dealing with the increasing complexity of the services.

4.6 | Brief summary of findings

As we have already mentioned in the beginning of this Section, the main idea of our study was to address knowledge questions around how does the evolution of services takes place, what its impact can be, and, whether we can predict it. In a nutshell, we can say that (i) the evolution of

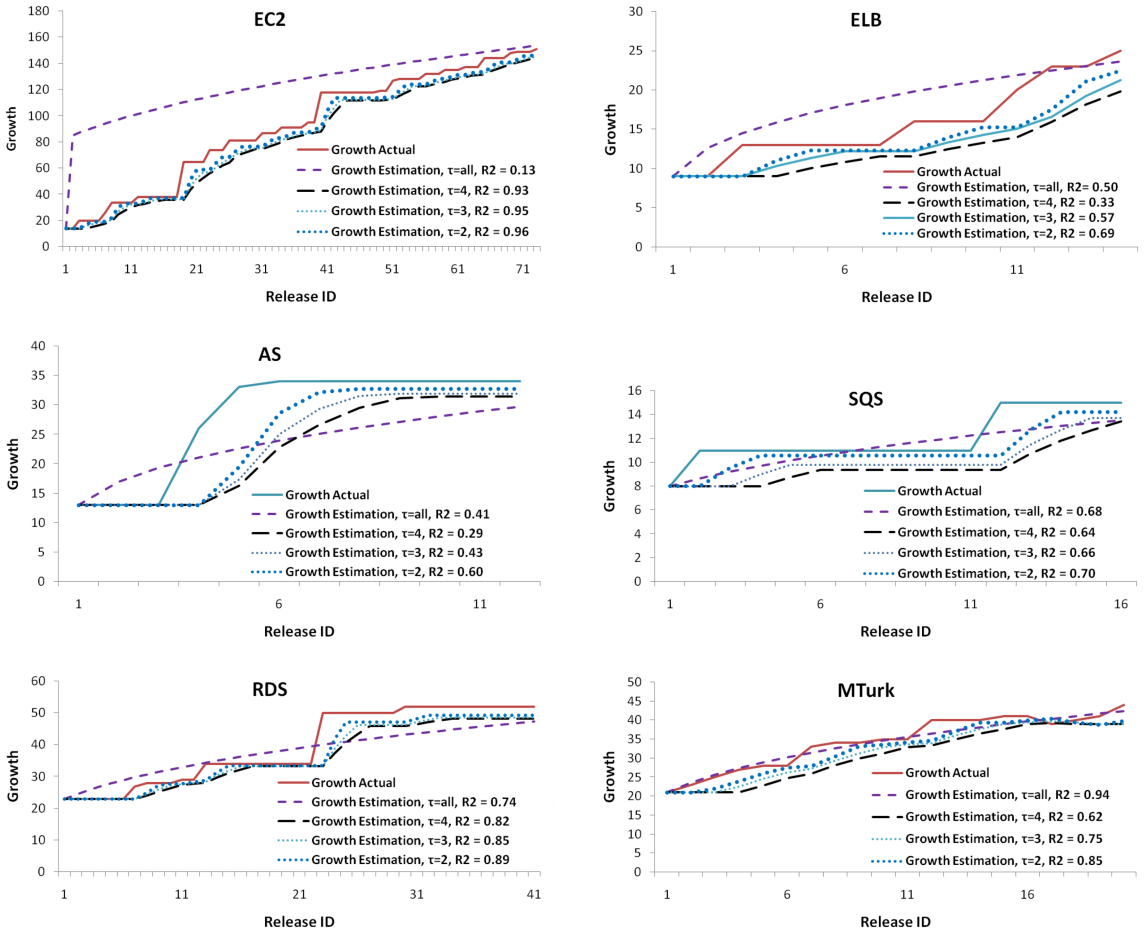

FIGURE 5 Inverse square model predictions of cumulative growth

services takes place with spikes of concentrated change amidst calm periods (which is reflected in all the major measurements of service evolution, namely heartbeat, incremental and cumulative change); (ii) both additions and maintenance actions take place; (iii) interface complexity starts high and increases over time; and, finally, (iv) whereas heartbeat forecasting does not seem plausible, the forecasting of the cumulative growth can be well approximated for the short term by a feedback-based model.

Having discussed all these knowledge extraction research questions, we are now ready to proceed to the research normative question, on how does a healthy Web service looks like, from the viewpoint of the developer of a service-oriented system. This will be the core of the next section.

5 | THE ATHLETIC HEART SYNDROME PATTERN AND ITS PRACTICAL IMPLICATIONS

A key lesson after performing this study on the evolution of Amazon services is that the assessment of the evolutionary behavior of web services is quite demanding and challenging, especially since the typical developer of a service-oriented system does not have access to the internals of the services that he uses or he intends to use. The minimum information that one needs to perform this task is service release notes that allow the developer to extract information, concerning the changes that have been performed in subsequent service releases. Still, the task is not easy because there are many issues to consider. So, *our experience shows that although assessing the evolution of services is an interesting research exercise, the successful completion of such a task requires time and effort that an ordinary developer working under deadlines and backlog requirements may not have.*

To help the developer of a service-oriented system in selecting general-purpose services that evolve in an appropriate way, we formally specify the athletic heart syndrome pattern that emerged during our study, along with its consequences and practical implications for the developer. Figure 6 gives an overview of the pattern. The pattern does not deal with criteria that concern the functionalities of the services, but rather, it focuses on evolution-driven selection criteria.

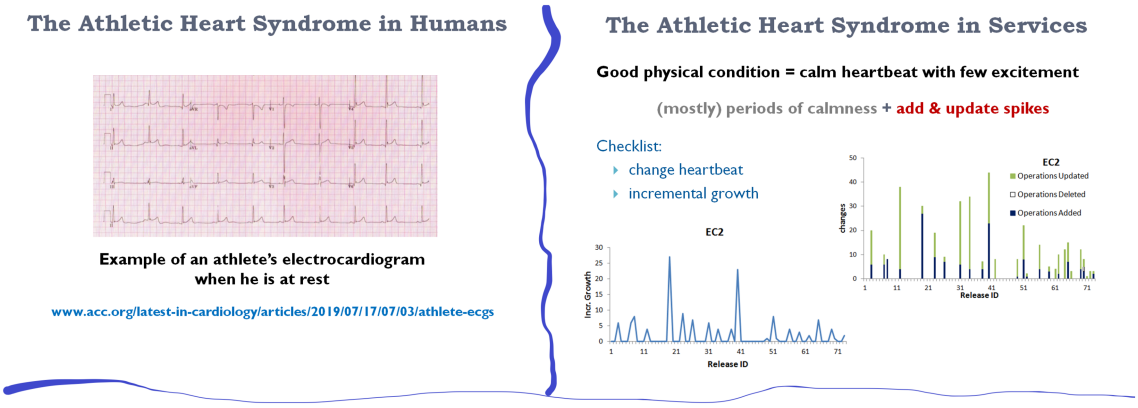

FIGURE 6 *The athletic heart syndrome* in service evolution

Patterns are a valuable means of conveying knowledge and experience. In the patterns community,[8] there are several different pattern writing forms/templates that allow to systematically document patterns.[44,45] Here, we shall rely on the Complien (a.k.a. canonical) form to specify our evolution-driven service selection pattern. The Complien form is very popular and widely accepted in the patterns community. Moreover, it has been used for the specification of several other service evolution patterns, reported in previous related works.[15,16]

According to the Complien form, a pattern specification consists of five parts: The **context** part describes the situations in which the pattern can be applied; the **problem** part describes what the problem actually is; the **forces** part discusses in more detail the different issues that should be resolved by the solution and possible trade-offs between them; the **solution** part gives in detail the solution to the problem; the **consequences** part reports benefits and liabilities of the solution, forces that have been resolved, unresolved forces, etc. A pattern specification may comprise some additional parts.[15,16] In our case, we provide a comparison between our pattern with other **related approaches and patterns**.

5.1 | Context

A development team implements a complex service-oriented system. To realize the functionalities that are specific to the system they develop a number of collaborating micro-services. To deal with general purpose problems like storage, networking, resource allocation and management they want to reuse general-purpose services, offered be an external service provider (e.g., a cloud infrastructure).

5.2 | Problem

A member of the team wants to select a general-purpose service to be used for the implementation of the system. To address the risk of using an external service in the implementation of the system, the team member wants to select a service that evolves properly. The methodology under which the selection takes place (i.e., whether there is a single service being evaluated, or alternative services of different providers being compared) is orthogonal to the problem.

5.3 | Forces

The problem is difficult; the developer should investigate several issues, without having access to the internals of the service implementation.

F1 The developer wants to select a service that changes to meet evolving requirements.
F2 The selected service should evolve in a way that lets the developer understand the evolving functionalities and deal with changes.
F3 The evolution of the selected service should involve both growth and maintenance activities.
F4 As the service evolves, the developer wants to be sure that the complexity of the service interface will not be a problem for service usage.
F5 The developer wants to make predictions about the way that the service evolves.

[8] hillside.net; europlop.net.

5.4 | Solution

To select a service that evolves according to his/her expectations, the developer should proceed as follows:

- Pick a service that provides access to prior release notes.
- Check the heartbeat of the changes, performed in the service interface. The heartbeat of a healthy service should look like the heartbeat of a healthy athlete when he is at rest. Specifically, the heartbeat of changes should consist mostly of calm periods, in which the service interface does not change, interrupted by spikes of change that include additions, updates, and rare deletions.
- Check the incremental growth of the service. In particular, the incremental growth of the service should comprise mainly calm periods of maintenance, interrupted by spikes of growth.
- If the previous properties hold, the developer can select the service. Mostly likely, the evolution of the service will be appropriate.

5.5 | Consequences and practical implications

C1 The alteration of calm periods with spikes of change is a clear sign that the service changes to meet evolving requirements.
C2 Releases with spikes of changes that include updates and deletions in the service interface require attention, because the developer will likely have to modify the system to keep up with the changes that have been performed in the service interface.
C3 The developer can take advantage of the calm periods during which the service interface does not change, to assimilate and to master the functionalities of the service, as they evolve in the spikes of changes that take place in between the calm periods.
C4 Spikes of growth show that the service is enhanced with new functionalities.
C5 Calm periods in the incremental growth of the service denote maintenance activities. Most likely, the maintenance activities will be improvements of the service programming platform, the service infrastructure and the availability of the service to different geographical regions and zones.
C6 The interface of the service may be complex. Probably, the complexity will increase as the service evolves. However, the increase of the interface complexity will be smooth due to the frequent calm periods. Thus, the developer should account for a reasonable amount of time and learning effort to start using the service. Likely, the effort that will be required for using subsequent releases of the service will be smaller.
C7 The developer will not be able to forecast the exact amount and the type of changes that will be performed over time. Hence, he/she will not be able to plan accurately the necessary resources for coping with these changes. Instead, the developer should allocate resources for dealing with the worst case.
C8 The developer will be able to coarsely predict the amount of new functionalities that will be added, as the service evolves. He/she can benefit from such predictions towards planning the resources that he/she will need for learning new functionalities that contribute to the complexity increase of the service interface.

5.6 | Related approaches and patterns

In Section 2, we discussed in detail the state of the art on service evolution. Here, we specifically focus on a comparison between these approaches and *the athletic heart syndrome* pattern (Table 12). Before getting into the details of this comparison, we have to clarify that the diversity of the state of the art approaches is very large. Therefore, a comparison that relies on a particular scenario or case study cannot be useful and fair. For this reason, we compare the related approaches with our pattern at a logical level. The objective of the comparison is to discuss fitness for purpose, on the basis of the *problem* that is solved by the pattern and the *forces* that define this problem.

- **Service change detection:** All of the service change detection tools discussed in Section 2.1 can be useful for calculating the evolution history of services and facilitate the resolution of F1. However, the proposed tools cannot, on their own, resolve the remaining forces (F2–F5) of the problem solved by the pattern. In comparison with these approaches, *the athletic heart syndrome* pattern provides a solution that dictates the necessary methodological steps that resolve all the forces (F1–F5) of the problem, as reflected by the the resulted consequences (C1–C8) of the pattern.
- **Service change management and prediction:** The service change management approaches discussed in Section 2.2 can contribute to the resolution of F3 and, specifically, to the part of this force that refers to letting the developer deal with service changes easier. Moreover, the service evolution prediction approach[8] discussed in Section 2.2 can be employed to resolve F5. At this point, it is important to state that *the athletic heart syndrome* pattern and the related *change management and prediction approaches complement each other towards the resolution of F3 and F5*. Ideally, it would be useful to select services that evolve properly, according to the pattern, and further rely on advanced change

TABLE 12 Comparison with related approaches and patterns

	Pattern forces				
Related approaches	**F1**	**F2**	**F3**	**F4**	**F5**
Fokaefs et al.[5]	✓				
Romano and Pinzger[6]	✓				
Fokaefs and Stroulia[7,21]	✓				
Li et al.[22]	✓				
Espinha et al.[23]	✓				
Treiber et al.[18]		✓			
Leitner et al.[26]		✓			
Zou et al.[27]		✓			
Andrikopoulos et al.[9]		✓			
Khebizi et al.[10]		✓			
Banati et al.[28]		✓			
Campihnos et al.[11]		✓			
Sampaio et al.[29]		✓			
Groh et al.[12]		✓			
Tran et al.[4]		✓			
Nguyen et al.[32]		✓			
Baresi et al.[13]		✓			
Calinescu et al.[33]		✓			
Wang et al.[14]		✓			
Autili et al.[35]		✓			
Wang et al.[8]					✓
Wang et al.[15]		✓		✓	
Lübke et al.[16]		✓			
Athletic Heart Syndrome pattern	✓	✓	✓	✓	✓

management approaches, like the ones discussed in Section 2.2. Nevertheless, the aforementioned efforts do not cover the resolution of F1 and F3, concerning the selection of services which are regularly enhanced and maintained to meet evolving requirements. In addition, the related approaches do not resolve F4 that focuses on the evolving complexity of service interfaces. The way to deal with these forces is to study the evolution of the services as described by *the athletic heart syndrome* pattern and select services that fulfill the criteria specified in the solution of the pattern.

- **Service evolution patterns:** The service evolution patterns discussed in Section 2.2 can be helpful for the resolution of F3 and F4. Specifically, some of the patterns can help the developer towards dealing with changes performed during the services' lifetime. Other patterns can be applied to reduce the evolving complexity of the service interfaces. Nevertheless, the related patterns do not concern the study of the service evolution, which is required for selecting services that evolve properly. Consequently, there is a difficulty in using other patterns towards resolving the rest of the forces (F1, F2, F3) that are handled by *the athletic heart syndrome* pattern.

To summarize, in comparison with the state of the art approaches, *the athletic heart syndrome* pattern provides an *effective* solution that resolves all the forces (F1–F5) of the problem, as reflected by the resulted consequences (C1–C8) of the pattern. When it comes to *efficiency*, the effort for applying the pattern is proportional to the size of the evolution history of the services under study and the evolutionary behavior of the services. Assessing the evolution of active (respectively: long-lived) services requires more effort compared to quiet (respectively: short-lived) services.

6 | PATTERN VALIDITY

To increase our confidence on the validity of the pattern that we report in this paper, we had to deal with certain threats, discussed in this section.

6.1 | Construct validity

To address the research questions of our study, we have measured operation and data types additions, deletions and updates at the service-interface level. This is pretty much the standard procedure in all of the previous related studies on service evolution (Section 2.1). Some of these studies also consider operation renaming, merges and splits. In our study, we do not trace such changes. However, when it comes to the detailed heartbeat of changes and the service growth, operation renamings, merges, and splits end up to operation additions, deletions and updates. In addition to what is typically done in previous related studies on service evolution, in our work we also consider the detailed release logs of the services to produce a more detailed view about the purpose of the changes that have been performed. Therefore, we consider that the treatment of our research questions is sufficient.

Concerning the accuracy of the service evolution histories, we used Membrane SOA, a well-known open-source API, as the basis for their construction. To strengthen our confidence on the accuracy of the detected changes, we performed tests of the API, based on synthetic WSDL specifications that covered the different types of changes that we consider in our study. Moreover, we manually inspected random samples of the collected data. Finally, we checked if the distributions of the detected changes are inline with those detected in other similar works.[5,6] Notably, as in these studies, we also found frequent additions and updates and rare deletions.

The metrics that we employed in our study are inline with similar metrics that have been employed in the literature. Specifically, the incremental growth of a software system is typically measured as the difference in the number of modules, included in subsequent releases of the system.[17,38,40] The growth of the software system is usually measured with respect to the size of the system, using metrics like the number of modules that constitute the system, lines of code, and so forth.[17,38,40] The complexity metric that we used is based on Sneed's[41] structural complexity of service interfaces and our previous work on the systematic decomposition of service interfaces.[2]

A possible limitation of our study that is also present in previous related studies is that we did not consider changes in the implementations of the services. This was not possible because we did not have access to the service implementations. Nevertheless, assuming such information was ever made publicly available, extending the study with information about the internal workings of the services would possibly allow to refine *the athletic heart syndrome* pattern.

6.2 | Internal validity

Internal validity concerns possible cause and effect relationships established in a study. In our study, we validated the positive correlation between the heartbeat of changes performed in service operations and data types using Spearman's correlation coefficient. Moreover, we used Spearman's correlation coefficient to validate the positive correlation between the growth of operations and data types. To validate the smooth logarithmic trend in the complexity increase of the service interfaces and the accuracy of the cumulative growth predictions we relied on the R^2 statistic.

The comparison of our approach with other related approaches and patterns was driven by the context, the problem and the forces of *the athletic heart syndrome* pattern. Consequently, the results of the comparison are related to the aforementioned framework. Under a different framework, the comparison of *the athletic heart syndrome* pattern with the related approaches and patterns can result into different conclusions.

6.3 | External validity

When it comes to *external validity*, obviously, our results cannot be generalized to the overall population of Web services. However, this is not at all the purpose of our study. Instead, our goal is to provide a pattern that formally specifies the criteria for selecting services that evolve properly. In general, patterns report best practices that can be used to solve common problems. However, not all best practices can be considered as valid patterns. Typically, a best practice should verified to be a recurring phenomenon,[46] preferably in at least three real world cases. In the patterns community this is known as "The Rule of Three".[47] In our study, we observed the athletic syndrome pattern in 6 different services. We selected the services based on their popularity, their longevity and domain coverage. The population of the examined services is comparable with the number of services that have been studied in other related studies of Web service evolution.[5,6] According to Lehman,[17] 8 to 10 is a sufficient number of releases for a software evolution study. In our study, the number of service releases that we considered for the examined service varied from 12 to 73, released in time spans that varied from 3 to 8 years. Based on the above, we are confident that *the athletic heart syndrome* qualifies as a valid service selection pattern. Two possible issues that would allow to further enrich the pattern are to consider (a) more services from different providers, and, (b) services that offer similar functionalities.

7 | CONCLUSION

In this paper, we introduced *the athletic heart syndrome* pattern which formally specifies the criteria for selecting services that evolve properly. To come up with the pattern, we studied the evolution of a set of popular, long-lived, general purpose Amazon services. In particular, we studied the heartbeat of changes performed in the service interfaces, the incremental growth and the growth of the provided functionalities. Moreover, we investigated the detailed service release notes to find out more about the purpose of the changes that have been performed during the services' lives. The evolution of a particular service adheres to the pattern if its heartbeat of changes looks like the heartbeat of a healthy athlete when he is at rest. In particular, the heartbeat of changes performed in the service interface should consist mainly of calm periods in which the interface does not change, interrupted by spikes of change that include additions, updates and rarely deletions. The incremental growth of the functionalities provided by the service should also involve mostly calm periods, separated by spikes of growth. Conformance to the pattern guarantees that the service evolves over time. Adherence to the pattern further assures that the service evolution is a combination of growth and maintenance activities. The pattern guarantees that the complexity increase in the service interface will be smooth and that the expansion of the provided functionalities will be predictable. Finally, conformance with the pattern facilitates the comprehension and the adoption of the evolving functionalities.

Extending our study with services offered from different service providers and services that provide similar functionalities is one possible research direction of this work. Automating the selection of general-purpose services that conform with *the athletic heart syndrome* pattern is another possible extension of this work. Another issue for future research is to study the co-evolution of general-purpose services and systems that depend on them, towards assessing the impact of service evolution on these systems. The identification of patterns that allow to isolate dependent systems from service changes would be a useful tool for the developers of these systems. Finally, studying the evolution of microservices is also a challenging open issue for future research, as the main characteristics of these services are fundamentally different from the characteristics of the services that we have studied so far in the related literature.

DATA AVAILABILITY STATEMENT

The data that support the findings of this study are openly available in AWS-Data at github.com/zarras/AWS-Data/, reference number 53db8e8.

ORCID


Apostolos V. Zarras https://orcid.org/0000-0001-9521-5853

REFERENCES

1. Baresi L, Garriga M. Microservices: The evolution and extinction of web services? *Microservices, Science and Engineering*: Springer; 2020:3-28.
2. Athanasopoulos D, Zarras AV, Miskos G, Issarny V, Vassiliadis P. Cohesion-driven decomposition of service interfaces without access to source code. *IEEE Trans Serv Comput*. 2015;8:550-562.
3. Zarras AV, Vassiliadis P, Dinos I. Keep calm and wait for the spike! Insights on the evolution of amazon services. In: Proceedings of the 28th International Conference Advanced Information Systems Engineering (CAISE), Lecture Notes in Computer Science, vol. 9694. Springer; 2016:444-458.
4. Tran HT, Nguyen VT, Phan CV. Towards service co-evolution in SOA environments: a survey. In: Context-aware systems and applications, and nature of computation and communication. Springer; 2021:233-254.
5. Fokaefs M, Mikhaiel R, Tsantalis N, Stroulia E, Lau A. An empirical study on web service evolution. In: Proceedings of the 18th IEEE International Conference on Web Services (ICWS); 2011:49-56.
6. Romano D, Pinzger M. Analyzing the evolution of web services using fine-grained changes. In: Proceedings of the 19th IEEE International Conference on Web Services (ICWS); 2012:392-399.
7. Fokaefs M, Stroulia E. Using WADL specifications to develop and maintain REST client applications. In: Proceedings of the 22nd IEEE International Conference on Web Services (ICWS); 2015:81-88.
8. Wang H, Kessentini M, Ouni A. Prediction of web services evolution. *Proceedings of the 14th International Conference on Service-Oriented Computing (ICSOC)*, Lecture Notes in Computer Science, vol. 9936: Springer; 2016:282-297.
9. Andrikopoulos V, Benbernou S, Papazoglou MP. On the evolution of services. *IEEE Trans Softw Eng*. 2012;38(3):609-628.
10. Khebizi A, Seridi-Bouchelaghem H, Benatallah B, Toumani F. A declarative language to support dynamic evolution of web service business protocols. *SOCA*. 2017;11(2):163-81.
11. Campinhos J, Seco JC, Cunha J. Type-safe evolution of Web services. In: Proceedings of the IEEE/ACM 2nd International Workshop on Variability and Complexity in Software Design (VACE); 2017:20-26.
12. Groh O, Baraki H, Jahl A, Geihs K. COOP - automatic validation of evolving microservice compositions. In: Proceedings of the Seminar Series on Advanced Techniques & Tools for Software Evolution (SATTOSE), CEUR Workshop Proceedings, vol. 2510. CEUR-WS.org; 2019.
13. Baresi L, Guinea S, Manna VPL. Consistent runtime evolution of service-based business processes. In: Proceedings of the 11th IEEE/IFIP Conference on Software Architecture; 2014:77-86.
14. Wang X, Feng Z, Chen S, Huang K. Dkem: A distributed knowledge based evolution model for service ecosystem. In: Proceedings of the 25th IEEE International Conference on Web Services (ICWS); 2018:1-8.
15. Wang S, Higashino WA, Hayes M, Capretz MAM. Service evolution patterns. In: Proceedings of the 21st IEEE International Conference on Web Services (ICWS); 2014:201-208.

16. Lübke D, Zimmermann O, Pautasso C, Zdun U, Stocker M. Interface evolution patterns: balancing compatibility and extensibility across service lifecycles. In: Proceedings of the 24th ACM European Conference on Pattern Languages of Programs (EuroPLoP); 2019:15:1-15:24.
17. Lehman MM, Ramil JF. *Software Evolution and Feedback: Theory and Practice*: Wiley; 2006.
18. Treiber M, Truong HL, Dustdar S. On analyzing evolutionary changes of web services. In: Feuerlicht G, Lamersdorf W, eds. *Service-Oriented Computing*, LNCS, vol. 5472; 2009:284-297.
19. Wang Y, Wang Y. A survey of change management in service-based environments. *SOCA*. 2013;7(4):259-273.
20. Romano D. Analyzing the change-proneness of service-oriented systems from an industrial perspective. In: Proceedings of the 35th International Conference on Software Engineering (ICSE); 2013:1365-1368.
21. Fokaefs M, Stroulia E. Wsdarwin: A decision-support tool for Web-service evolution. In: Proceedings of the 29th IEEE International Conference on Software Maintenance (ICSM); 2013:444-447.
22. Li J, Xiong Y, Liu X, Zhang L. How does Web service API evolution affect clients? In: Proceedings of the 20th IEEE International Conference on Web Services (ICWS); 2013:300-307.
23. Espinha T, Zaidman A, Gross H-G. Web API growing pains: Loosely coupled yet strongly tied. *J Syst Softw*. 2015;100:27-43.
24. Skoulis I, Vassiliadis P, Zarras AV. Open-source databases: Within, outside, or beyond Lehman's laws of software evolution? In: Proceedings of the 26th International Conference Advanced Information Systems Engineering (CAISE), Lecture Notes in Computer Science, vol. 8484. Springer; 2014:379-393.
25. Vassiliadis P, Zarras AV, Skoulis I. Gravitating to rigidity: Patterns of schema evolution - and its absence - in the lives of tables. *Inf Syst*. 2017;63:24-46.
26. Leitner P, Michlmayr A, Rosenberg F, Dustdar S. End-to-end versioning support for web services. In: Proceedings of the 5th IEEE International Conference on Services Computing (SCC); 2008:59-66.
27. Zou ZL, Fang R, Liu L, Wang QB, Wang H. On synchronizing with web service evolution. In: Proceedings of the 15th IEEE International Conference on Web Services (ICWS); 2008:329-336.
28. Banati H, Bedi P, Marwaha P. Wsdl-temporal: An approach for change management in web services. In: Proceedings of the 2nd International Conference on Uncertainty Reasoning and Knowledge Engineering (URKE); 2012:44-49.
29. Sampaio AR, Kadiyala H, Hu B, Steinbacher J, Erwin T, Rosa N, Beschastnikh I, Rubin J. Supporting microservice evolution. In: Proceedings of the 33rd IEEE International Conference on Software maintenance and evolution (icsme); 2017:539-543.
30. Ali M, Angelis GD, Polini A. Servicepot - an extensible registry for choreography governance. In: Proceedigns of the 7th IEEE international symposium on service-oriented system engineering (sose); 2013:113-124.
31. Angelis FD, Fanì D, Polini A. Partes: A test generation strategy for choreography participants. In: Proceedings of the 8th IEEE International Workshop on Automation of Software Test (AST); 2013:26-32.
32. Nguyen CD, Marchetto A, Tonella P. Test case prioritization for audit testing of evolving Web services using information retrieval techniques. In: Proceedings of the 9th IEEE International Conference on Web Services (ICWS); 2011:636-643.
33. Calinescu R, Grunske L, Kwiatkowska M, Mirandola R, Tamburrelli G. Dynamic QoS management and optimization in service-based systems. *IEEE Trans Softw Eng*. 2011;37(3):387-409.
34. Lv C, Jiang W, Hu S, Wang J, Lu G, Liu Z. Efficient dynamic evolution of service composition. *IEEE Trans Serv Comput*. 2018;11(4):630-643.
35. Autili M, Salle AD, Gallo F, Pompilio C, Tivoli M. On the model-driven synthesis of evolvable service choreographies. In: Proceedings companion of the 12th European Conference on Software Architecture: Companion Proceedings (ECSSA). ACM; 2018:20:1-20:6.
36. Autili M, Inverardi P, Tivoli M. Automated synthesis of service choreographies. *IEEE Software*. 2015;32(1):50-57.
37. Zarras AV, Dinos I, Vassiliadis P. AWS-data: WSDL descriptions for EC2, ELB, AS, SQS, RDS and MTurk. github.com/zarras/AWS-Data/; 2021.
38. Lehman MM, Ramil JF, Wernick P, Perry DE, Turski WM. Metrics and laws of software evolution—the nineties view. In: Proceedings of the 4th IEEE International Software Metrics Symposium (Metrics); 1997:20-34.
39. Lehman MM, Ramil JF, Perry DE. On evidence supporting the feast hypothesis and the laws of software evolution. In: Proceedings of the 5th IEEE International Software Metrics Symposium (Metrics); 1998:84-88.
40. Xie G, Chen J, Neamtiu I. Towards a better understanding of software evolution: an empirical study on open source software. In: Proceedings of the 25th IEEE International Conference on Software Maintenance (ICSM); 2009:51-60.
41. Sneed HM. Measuring Web service interfaces. In: Proceedings of the 12th IEEE International Symposium on Web Systems Evolution (WSE); 2010:111-115.
42. Easterbrook S, Singer J, Storey M-AD, Damian DE. Selecting empirical methods for software engineering research. In: Shull F, Singer J, Sjøberg DIK, eds. *Guide to Advanced Empirical Software Engineering*: Springer; 2008:285-311. https://doi.org/10.1007/978-1-84800-044-5_11
43. Bourque P, Fairley RE, (eds.). *Guide to the Software Engineering Body of Knowledge, version 3.0, IEEE Computer Society (Swebok)*: IEEE; 2014.
44. Meszaros G, Doble J. A pattern language for pattern writing. *Pattern languages of program design 3*: Addison-Wesley Longman; 1997:529-574.
45. Wellhausen T, Fiesser A. How to write a pattern? a rough guide for first-time pattern authors. In: Proceedings of the 16th ACM European Conference on Pattern Languages of Programs (Europlop); 2011:1-9.
46. Gamma E, Helm R, Johnson RE, Vlissides JM. *Design Patterns—Elements of Reusable Object-Oriented Software*: Addison-Wesley; 1994.
47. Kuchana P. *Software Architecture Design Patterns in java*: Auerbach; 2004.

How to cite this article: Zarras AV, Dinos I, Vassiliadis P. The athletic heart syndrome in web service evolution. *J Softw Evol Proc*. 2022;34(10):e2418. doi:10.1002/smr.2418