SOCA (2017) 11:1–31
DOI 10.1007/s11761-016-0203-8

ORIGINAL RESEARCH PAPER

A survey of autonomic computing methods in digital service ecosystems

Dhaminda B. Abeywickrama[1] · **Eila Ovaska**[1]

Received: 28 January 2016 / Revised: 12 September 2016 / Accepted: 16 November 2016 / Published online: 28 November 2016
© The Author(s) 2016. This article is published with open access at Springerlink.com

Abstract Service engineering of digital service ecosystems can be associated with several challenges, such as change and evolution of requirements; gathering of quality requirements and assessment; and uncertainty caused by dynamic nature and unknown deployment environment, composition and users. Therefore, the complexity and dynamics in which these digital services are deployed call for solutions to make them *autonomic*. Until now there has been no up-to-date review of the scientific literature on the application of the autonomic computing initiative in the digital service ecosystems domain. This article presents a review and comparison of autonomic computing methods in digital service ecosystems from the perspective of service engineering, i.e., requirements engineering and architecting of services. The review is based on systematic queries in four leading scientific databases and Google Scholar, and it is organized in four thematic research areas. A comparison framework has been defined which can be used as a guide for comparing the different methods selected. The goal is to discover which methods are suitable for the service engineering of digital service ecosystems with autonomic computing capabilities, highlight what the shortcomings of the methods are, and identify which research activities need to be conducted in order to overcome these shortcomings. The comparison reveals that none of the existing methods entirely fulfills the requirements that are defined in the comparison framework.

✉ Dhaminda B. Abeywickrama
dhaminda.abeywickrama@gmail.com

Eila Ovaska
eila.ovaska@vtt.fi

[1] Service and Information Architectures, VTT Technical Research Centre of Finland Ltd, Kaitoväylä 1, 90570 Oulu, Finland

Keywords Autonomous systems · Digital ecosystems · Service engineering · Self-* features · Quality attributes

Abbreviations

AC	Autonomic computing
ACE	Autonomic communication elements
ANS	Autonomic nervous system
BASE	Behavior, asynchrony, state and execution
BIONETS	BIOlogically inspired autonomic NETworks and Services
CASCADAS	Component-ware for Autonomic Situation-aware Communications, and Dynamically Adaptable Services
DAS	Dynamically adaptive system
DSE	Digital service ecosystem
DSL	Domain-specific language
EOA	Ecosystem-oriented architecture
ICARE	Innovative Cloud Architecture for Real Entertainment
IVS	Intelligent vehicle system
KAOS	Keep All Objectives Satisfied
MAPE	Monitoring, analyzing, planning and execution
MDE	Model-driven engineering
NIMSAD	Normative information model-based systems analysis and design
QoS	Quality of service
REST	Representational state transfer
SAPERE	Self-aware pervasive service ecosystems
SCC	Self-controlled components
SOA	Service-oriented architecture
UDDI	Universal Description, Discovery and Integration

URDAD Use-Case, Responsibility-Driven Analysis and Design

1 Introduction

A *digital service ecosystem* (DSE) has been defined as an open, loosely coupled, domain-clustered, demand-driven, self-organizing agents' environment where each entity is proactive and responsive for its own benefit [1,2]. A DSE can be seen as a new kind of self-organized environment that addresses openness and dynamicity, enabling collaborative innovation and co-creation among the members of the ecosystem [3]. In this context, a *digital service* can be any added value/benefit that is delivered digitally [3]. It is automated entirely and ideally controlled by the customer of the service.

DSEs are complex and dynamic due to several reasons, such as increasing number of components, devices and services; changes in the technology used; and applications becoming more difficult to manage. As a result, DSEs are evolving rapidly without much control. In this context, service engineering of DSEs has new challenges, such as change and evolution of requirements; gathering of quality requirements and assessment; and uncertainty caused by dynamic nature and unknown deployment environment, composition and users [3]. Another important challenge in digital ecosystems is co-evolution among ecosystem members and in customer participation. The complexity and dynamics in which these digital services are deployed, therefore, call for solutions to make such services *autonomic* [4,5], i.e., capable of dynamically self-adapting their behavior in response to changing situations. Autonomic computing (AC) initiative can provide strong elements in overcoming the main challenges and obstacles to the exploitation of DSEs.

The AC initiative's influence has been present in many computing domains, e.g., grid computing [6,7], artificial intelligence [8], robotics [9], control systems [10], service-oriented architecture (SOA) [11,12], cloud computing [13] and complex adaptive systems [14]. In recent years, several methods and techniques have been proposed to exploit the benefits of the AC initiative in service-oriented ecosystems, for example, SAPERE [15–20], CASCADAS [21–23] and BIONETS [24–27]. However, very little work has applied the AC initiative in the DSEs domain [28–30]. Looking at the state of the art, none of the methods seems to address in a generic and adaptive way the service engineering of DSEs, especially an ecosystem-based method on applying the AC initiative is missing in the DSE domain. Furthermore, there is no good systematic review of scientific literature when it comes to the DSE domain. There are several literature reviews on the general research area of AC [31–34] and a few narrow literature reviews focusing on its application in domains such as grid computing [35] and self-adaptive systems [36,37]. None of these reviews covers the DSE domain. A survey article that addresses the following is clearly missing in the literature: (1) the main requirements of a service engineering methodology for autonomic DSEs; (2) the shortcomings or gaps in existing AC methods in DSEs; and (3) the research activities required to overcome the shortcomings. This article aims to set this straight.

This survey article presents a review and comparison of the AC methods in DSEs from the viewpoint of service engineering, i.e., requirements engineering and architecting of services. The review is based on systematic queries in four leading scientific databases and Google Scholar, and it is organized in four thematic research areas. After the literature searches and analysis, 12 primary methods have been selected to be most relevant to our study and a review has been conducted by the authors to identify the most relevant aspects of the research. In this regard, 13 research questions have been used which have been incorporated in a comparison framework. This framework can be used as a guide for comparing the different scientific methods selected from the research areas.

This article unfolds as follows. In Sect. 2, we provide background information and definitions of the terminology that are frequently used in the context of the methods reviewed in this survey. Section 3 outlines the research method used in the literature review. Section 4 introduces our comparison framework for comparing the different primary methods selected from the research areas. In Sect. 5, we present an overview of each primary method and a comparison of these methods using the framework defined. Section 6 discusses the results of our survey, and Sect. 7 concludes the survey.

2 Background and definitions of the main technology

In this section, we provide background information and definitions of the terminology that are often used in the context of the methods analyzed. To this end, we define terms for AC, DSEs, digital services and quality attributes for the purposes of this article and place them in context.

2.1 Autonomic computing initiative

The terms *autonomic*, *autonomy*, *autonomous* and *autonomicity* have been presented in various domains such as language, biology and philosophy. In general, the term *autonomic* implies occurring involuntarily, unconsciously or automatically, or resulting spontaneously, from internal causes such as autonomic reflexes. Meanwhile, the term *autonomous* originates from ancient Greek in early nineteenth century, and in Greek it means having its own laws.

According to Oxford English Dictionary [38], autonomous signifies one's capability of self-governance or having the freedom to act independently, also implying self-containment and self-direction. *Autonomicity* signifies the state of being autonomic. Meanwhile, the term *autonomic computing* has been named after the human body's autonomic nervous system (ANS). ANS is responsible for the human body to perceive, adapt to and interact with the world in order to manage dynamically changing and unpredictable circumstances.

The evolution of AC from its inception can be described as follows. Several initiatives were undertaken by both industry and academia since early 1990s to develop self-managing and autonomous systems, thus contributing to the AC initiative. In this regard, Small Unit Operation Situational Awareness System is a notable preliminary self-managing project initiated in 1997 by the Defense Advanced Research Projects Agency (DARPA) [39]. This project developed technological aids that help the army with operational superiority, for example, providing the soldiers with richer information about the battle space or environment through improved communication and electronic sensing capabilities. Later, another project on self-management was initiated by DARPA called Dynamic Assembly for Systems Adaptability, Dependability, and Assurance. Its objective was to enable mission critical systems to meet high-assurance, dependability and adaptation requirements. In the late 1990s, NASA made use of the AC initiative in its space projects, such as the Mars Pathfinder and Deep Space 1. NASA's main aim was to make deep space probes more autonomous so that the probes can speedily adapt to extraordinary situations and space crafts are able to carry out autonomous operations for longer periods of time with no human intervention [35].

On March 8, 2011, Dr. Horn, research director at IBM, presented the importance of AC and its direction during a keynote speech at Harvard University [40,41]. Soon afterward, IBM server group introduced the eLiza project, which was later known as the AC project, thus beginning the AC journey at IBM. The AC initiative is a vision introduced by IBM for creating self-managed systems [4]. It seeks to render a computing system as self-managed, that is, to enable computer systems to manage themselves so as to minimize the need for human intervention [42]. The main goal of AC is to address the increasing complexity of modern computing systems by removing demand for skilled administrative interventions and automating system management [43]. AC benefits the IT domain in the short term by reducing the dependence on human involvement and the system total cost of ownership. Near short-term benefits more specifically are: improved user experience because of better system quality of service (QoS); reduced requirements for human intervention; better user access to services due to more natural human–machine interaction facilities; lower maintenance costs due to reduced requirements for human intervention; and lower usage costs due to better resource management [43].

In 2003, an architectural blueprint to build AC systems was introduced by IBM in which five building blocks for an autonomic system have been presented [42]. The blueprint also identified four self-* characteristics considered as fundamental for any autonomic system, and as a consequence, the most cited in the AC domain are: *self-configuration*, *self-healing*, *self-optimization* and *self-protection* [42]. These features are referred in short as self-chop [42]. Since AC domain's inception, the list of self-* features has been continuously growing. However, many of the latter features can be incorporated in the original self-chop list. Examples of the other self-* features are: *self-anticipating, self-adapting, self-adjusting, self-aware, self-critical, self-defining, self-destructing, self-diagnosis, self-governing, self-installing, self-managing, self-monitoring, self-organized, self-recovery, self-reflecting, self-simulation, self-stabilizing*. Other than these self-* properties, *context-awareness* specifically represents an additional key capability of an autonomic system. It means an autonomic system must be able to detect and adapt to changes in its execution environment, which can be user behavior, available resources or interactions with neighboring systems [43].

IBM has proposed five incremental levels of maturity in autonomy in [40,42] where self-management and autonomicity have been progressively integrated into the continuously evolving software system. They are *basic*, *managed*, *predictive*, *adaptive* and *autonomic* [43]. In another complementary classification scheme presented in [32], the autonomy of systems has been adapted to four classes: *support*, *core*, *autonomous* and *autonomic*. Today the AC initiative's influence has been present in many computing domains, such as grid computing, artificial intelligence and multi-agent systems, robotics, control systems, SOA, cloud computing and complex adaptive systems. However, very little scientific literature exists on the application of the AC initiative in the DSEs domain.

2.2 Digital service ecosystems

A *service ecosystem* is a socio-technical complex system where service providers can reach shared goals and utilize the services of other members in the ecosystem to gain added value [44,45]. A DSE is part of a service ecosystem, but it only covers the digital part, leaving out the social part. An example of a DSE is an interactive multi-screen TV services ecosystem in the Innovative Cloud Architecture for Real Entertainment (ICARE) project [3,46]. This DSE includes 25 service ecosystem members from Europe providing and using digital cloud-based services on operating end-to-end interactive multi-screen TV services. There are two dimensions in a DSE: species and underlying infrastructure and

services support [1]. According to [1], several factors characterize a DSE, for example a strong information infrastructure, a domain-oriented cluster and rich resources offering cost-effective digital services.

A DSE contains several elements: ecosystem *members*, ecosystem *infrastructure*, *capabilities*, and *digital services* [3]. The main *members* of a DSE are service providers, service brokers, service consumers and infrastructure providers. The ecosystem *capabilities* describe the capability model that defines the properties of the ecosystem. It also describes how the properties have been implemented using the ecosystem services provided by the ecosystem infrastructure. The ecosystem *capabilities* are implemented by the *infrastructure*, which supports the utilization of core competencies and core assets, flexible business networking and efficient business decision-making. Independent ecosystem members provide *digital services* in a DSE where the members provide additional value for both service consumers and other service providers [3].

In this context, a *digital service* can be any added value that is delivered digitally [1–3]. It is automated entirely and ideally controlled by the customer of the service [3]. Users can use digital services to enrich their everyday life, for example, exploiting services that can aid a person to monitor and guide in his or her health and well-being issues. An example for a digital service can be found in [47], which is a situation-aware safety service for children. In it, sensor and social web technologies have been exploited in the development of a safety service to enable proactive and instantaneous assistance and guidance for children in their daily lives.

2.2.1 Quality attributes

In DSEs, achieving the expected quality of a digital service is very challenging as the quality goals of all the supporting services need to be satisfied as well. Therefore, addressing quality attributes in the earliest possible phases of the software lifecycle like requirements engineering and architecture design is central in DSEs.

Service requirements for DSEs can be categorized as *functional*, *non-functional*, *business requirements* and *constraints* [3]. *Functional requirements* describe the behavior of a service that fulfills the tasks of the user. On the other hand, *non-functional requirements* describe the *qualities* of the service system, which can be defined as internally and externally observable properties. Meanwhile, *business requirements* help service providers to achieve business goals, and *constraints* are characteristics that limit the development and use of the service [3].

Quality is a term with multi-dimensional meaning, which depends on the context it is used. Software quality has been defined in IEEE 1061 [48] as the degree to which software possesses a desired combination of attributes. ISO/IEC 25010 [49] presents a software quality model with six categories of characteristics (i.e., functionality, reliability, usability, efficiency, maintainability and portability), which are then divided into sub-characteristics. These non-functional characteristics of a component or system are commonly known as *quality attributes*. Quality attributes can be categorized as execution and evolution quality attributes [50]. Execution qualities (e.g., performance, security, availability, usability, scalability, reliability, interoperability, adaptability) are observable at runtime. In comparison, evolution qualities (e.g., maintainability, flexibility, modifiability, extensibility, portability, reusability, integrability and testability) are not distinguished at runtime, and as a result, solutions for evolution qualities are in the static structures of the software system [50].

Several challenges and limitations can be identified for ecosystem-based service requirements engineering process, such as service co-innovation, service value co-creation, enabling infrastructure and utilization of ecosystem's assets [3]. Also, the definition of quality requirements for DSEs needs further exploration, and special skills are required in the innovation and requirements analysis, negotiation and specification phases. However, quality ontologies, quality-driven methods and tool support for attaching quality properties for architectural elements as discussed in [51–55] can aid the quality requirements engineering process.

3 Research method

This section outlines the research method used in this survey. Our method was motivated by the *normative information model-based systems analysis and design (NIMSAD)* framework [56]. NIMSAD focuses on classification and thematic analysis of scientific literature. It is a general framework for evaluating any methodology, and it uses the entire problem-solving process as the basis of evaluation. A main goal of our survey is to describe and compare each primary method against the comparison framework (see Sect. 4) defined in the study. Typically, surveys based on the systematic literature review (SLR) method [57] focus more on the guidelines followed, and thematic analysis and detailed comparisons of the primary methods are not given much emphasis. As in the SLR method, the current study follows three different stages: *planning*, *conducting* and *reporting* (see Fig. 1 for process steps and outcomes). In this section, we provide an overview of the procedure followed and describe in detail the research questions and the search strategy followed; the primary method selection procedure and criteria applied; the quality of the selected papers; the data elements extracted from the papers; and the data analysis and synthesis methods used. The review was conducted by a research Fellow

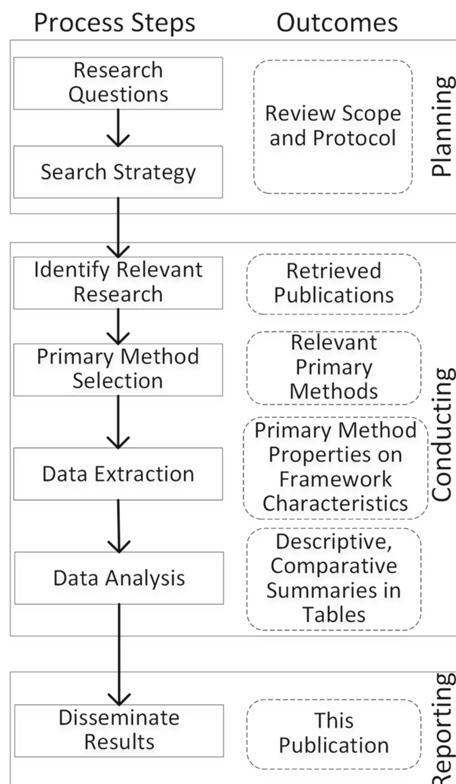

Fig. 1 Overall procedure of the survey

in AC systems, and the results were reviewed by a research professor in digital systems and services.

3.1 Planning stage

Research questions, search strategy and databases: The most important activity during planning (Fig. 1) is formulating the *research questions*. To this end, we have expressed our objectives in the form of 13 research questions (see Table 1), which have been defined from a broad perspective. Our objective was to capture a comprehensively full range of the literature on AC methods in DSEs.

A search strategy was defined to detect as much of the relevant literature as possible. That is, it needs to identify all relevant primary methods that address the research questions. To this end, literature searches were conducted from March–May 2015 (updated in January 2016) using four scientific databases—Scopus, IEEE Xplore, ACM digital library, Springer link—as well as Google Scholar. The scientific databases used are the most relevant in the software engineering area [57], and with the inclusion of Google Scholar, an exhaustive list of databases is not necessary. Our review is based on automatic search process which depends on the search engines of the scientific databases used. However, as the general search string (Boolean ANDs and ORs) has been adapted to each database according to its internal requirements, we contend that relevant studies have not been excluded. In each case, the search string "autonomic computing" AND "service ecosystem" was entered, with no temporal limitation. The initial results were as follows:

Scopus returned 58 results. Scopus is the largest abstract and citation database of peer-reviewed literature, indexing about 20,000 peer-reviewed journals, books and conference proceedings.

IEEE Xplore returned 2 results. This database covers electrical engineering, computer science and electronics, and indexes more than 160 journals and 1200 conference proceedings.

ACM digital library returned 1 result. ACM is the world's largest scientific educational computing society.

Springer link returned 26 results. This database contains journals and conference proceedings published by the Springer publishing house, and indexes over 8.3 million scientific documents.

Google Scholar returned 88 results. Google Scholar provides a simple way to broadly search for scholarly literature, allowing search across many disciplines and sources. This is beneficial in gaining an overall understanding of the results as it is based on various disciplines and sources.

3.2 Conducting stage

Once the planning stage is completed, the review proper (conducting stage) starts.

Primary method selection procedure and criteria: As an initial screening, titles and abstracts were read and the following three main research areas were manually identified:

- AC methods in DSEs
- AC methods in service ecosystems
- quality-driven software engineering methods.

The papers were considered from the perspective or viewpoint of service engineering, i.e., requirements engineering and architecting of services. A *research area* here represents an important study area considered for analysis and comparison. The use of solid quality-driven software engineering methods is essential in the service engineering of DSEs, as handling and managing quality in an ecosystem is a more complex and challenging process. In DSEs, service systems are integrated solutions from several service providers, and therefore, in order to achieve the intended quality of a digital service, quality goals of all the supporting services need to be satisfied too.

As the initial result set and number of research areas identified were very limited, the scope of the search was broadened. As stated in Sect. 1, DSEs are characterized by uncertainty caused by environmental disturbances or evolv-

Table 1 Research questions

ID	Research question
RQ1	What is the goal of the method?
RQ2	What are the benefits of using the method by the users (e.g., requirements engineers, service architects)?
RQ3	Does the method apply top-down approach or bottom-up approach or a combination of both to engineer services in the ecosystem?
RQ4	Does the method support both collective adaptation and adaptation by subparts, or does it operate with the guidance of a central controller only?
RQ5	What self-* features have been expressed in the method?
RQ6	Has the method supported expressing a comprehensive level of context-awareness?
RQ7	Has reflexivity been considered in the engineering process?
RQ8	What quality attributes have been expressed in the method?
RQ9	Does the method support evolution of the service ecosystem?
RQ10	Does the service ecosystem infrastructure support service interoperability?
RQ11	Does the component model of the method promote scalability of design (i.e., software engineering scalability) and execution complexity (i.e., performance scalability)?
RQ12	Has the method matured in several research papers?
RQ13	Has the method been applied at the conceptual level, as a proof of concept in the lab, or in the development of a large-scale industrial product using a case study?

ing requirements. Although the number of methods that address uncertainty using self-* features of the AC initiative in the DSE domain is scarce, as evident by the very limited results returned in the initial search process, valuable lessons can be learnt and applied through methods in other related domains like *dynamically adaptive systems* (DASs). Therefore, literature search was performed and the following search string "autonomic computing" AND "dynamically adaptive system" was entered with no temporal limitation. The result of this search is as follows:

"autonomic computing" AND "dynamically adaptive system"—Scopus: 20, IEEE Xplore: 2, ACM digital library: 35, Springer link: 48, Google Scholar: 69.

The titles and abstracts of the research articles returned were read and the following research area was manually identified:

- DASs-based methods that support self-* properties (in requirements engineering or architecting phases of the software lifecycle)

Figure 2 shows the research areas identified during the analysis. As shown in Fig. 2, the four research areas are represented by:

1. intersection of DSEs and AC
2. intersection of service ecosystems and AC
3. intersection of DASs and AC
4. quality-driven software engineering.

Note that, although an overlapping of the quality-driven software engineering research area can be identified with other domains (e.g., AC, service ecosystems, DASs), we consider quality-driven approaches independently from their application domain. Thus, it has been represented independently in Fig. 2.

After this analysis, as the resulting 349 articles were overlapping, articles indexed by two or more databases were eliminated. In order to handle the inconsistency between the meta-data format stored in different databases, we used the *RefWorks* reference management system. The benefit is it automates the task of aggregating research papers into a consistent list in a unified format.

The *selection criteria* are generally used to determine which studies are included in or excluded from a review. In this review, both theoretical and empirical studies, and studies conducted in both industry and in academia were considered for inclusion. The *inclusion* and *exclusion* criteria need to be based on the research questions, and for this purpose, the following criteria were used:

Inclusion criteria:

- The primary method is in one of the four main research areas identified during initial screening.
- The primary method provides evidence of service engineering, i.e., requirements engineering and architecting of services, which is the perspective considered in this study.

Exclusion criteria:

- The primary method provides no abstract or full text of the approach.

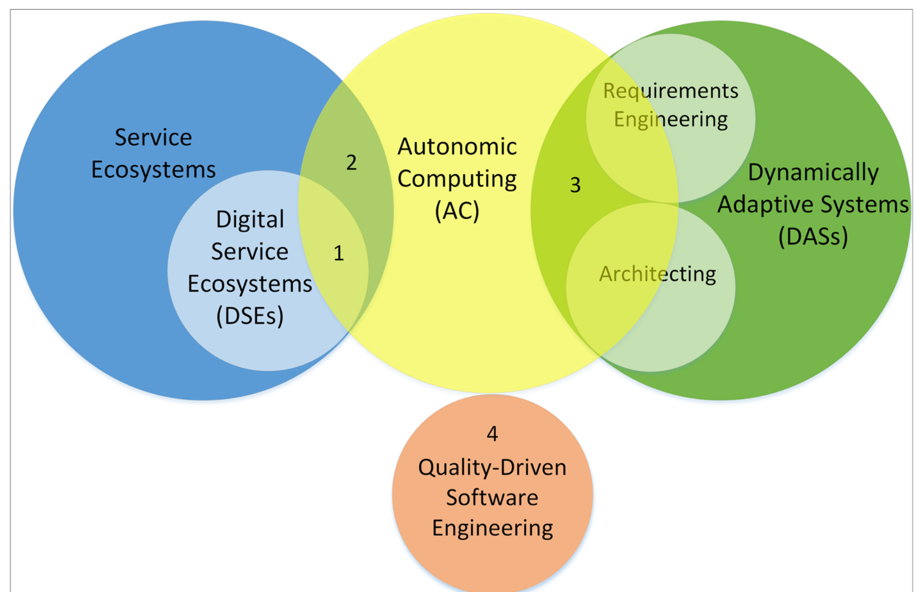

Fig. 2 Thematic research areas identified

- The primary method is written in a language other than English.

Finally, 12 primary methods were selected to be most relevant to our study and a review was conducted by the authors to identify the most relevant aspects of the research.

Quality of the selected papers: Quality criteria are important to assess the quality of the primary methods, which are aimed at minimizing bias and maximizing internal and external validity. To this end, first, quality instruments [57] can be formed which are checklists of factors that need to be evaluated for each primary method. Second, how quality data are to be used can be specified. However, in this review, no detailed quality assessment was performed as the goal of our survey was to identify all the AC methods in DSEs as much as possible. Existing scientific literature applying the AC initiative in DSEs is very little, which can be because it is still a very new research topic. Yet, as mentioned earlier, we used several general inclusion and exclusion criteria when selecting the primary methods for analysis.

Data elements extracted from papers: During the *data extraction* step (Fig. 1), data extraction forms were used to extract primary method properties from the primary methods. These primary method properties correspond and relate to the different characteristics (see Sect. 4) defined in the comparison framework. The intention was to help address all 13 research questions in each primary method. Some interpretation of data was necessary as not all information available was sufficient to answer all the 13 research questions. In addition, the following items were used during data collection: (1) the author(s) with their affiliations, the source (e.g., Journal article, conference paper, technical report) and year; (2) research area and scope; (3) the most relevant papers on the primary method; (4) a summary of the method; and (5) additional notes.

Data analysis and synthesis methods used: The *data analysis* step (Fig. 1) is used to synthesize the data so that the research questions can be answered. This step involved collating and summarizing the results of the primary methods in tables. Tables were used to organize the data with basic information about each study. The synthesis here is descriptive, exploratory and comparative. It is descriptive as the analysis is made by defining the research questions and elements of the comparison framework. Exploratory analysis is performed by finding out the thematic research areas and mapping the identified data/methods to them. Comparison analysis is done by studying and presenting characteristics of each primary method in the thematic research areas, and summarizing and analyzing the main findings.

In addition to answering the research questions, we used the data to identify interesting trends or limitations, such as how long and who has led the research in the respective research areas identified, i.e., any specific organization of researchers, and limitations of the current research approaches.

3.3 Reporting stage

We will disseminate the results of the review using a Journal article (this publication, see Fig. 1). The results of this review are provided in Sect. 5, while a discussion of the results is provided in Sect. 6.

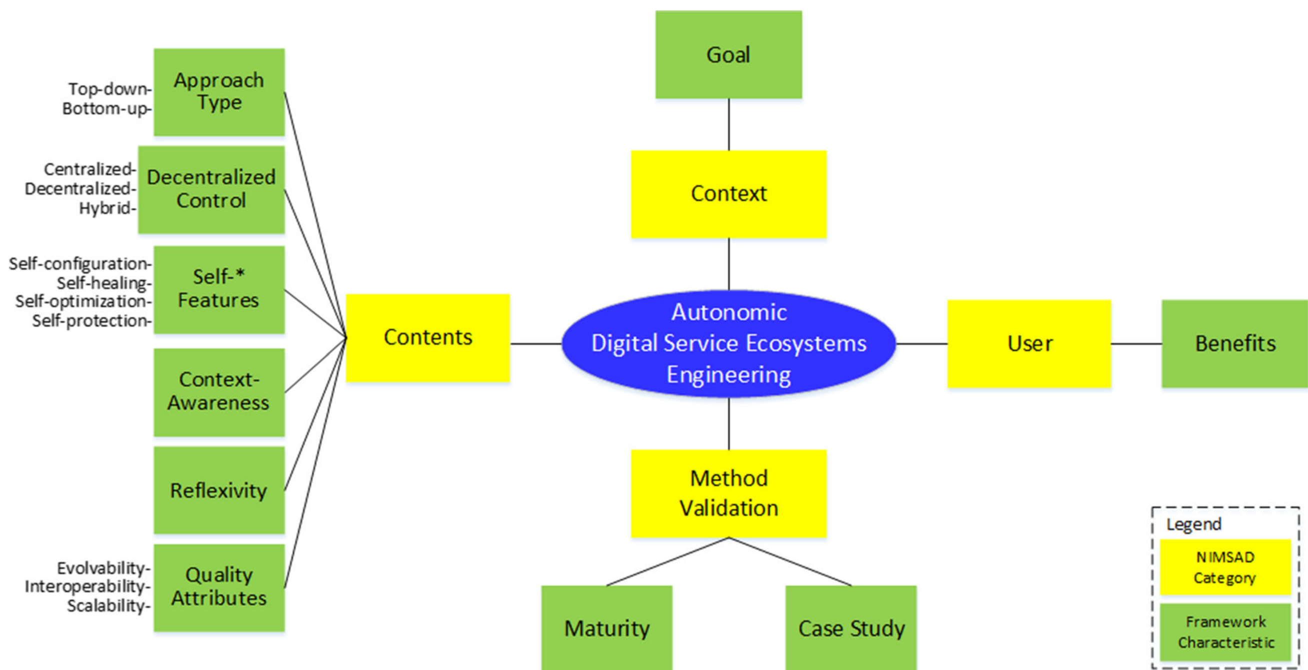

Fig. 3 Comparison framework taxonomy

4 A comparison framework for autonomic computing methods in digital service ecosystems

In this section, we introduce our comparison framework that we use for comparing the different scientific methods from the four thematic research areas. It incorporates the 13 research questions identified in Table 1, Sect. 3.1. We explain the different characteristics of the framework and provide justifications for their inclusion.

As mentioned in Sect. 1, none of the surveyed methods appears to address in a generic and adaptive way the service engineering of DSEs. That is, specifically, an ecosystem-based method on applying the AC initiative is missing in the DSE domain. Therefore, there is a need for a coherent, systematic ecosystem-based method and framework to support the requirements engineering and architecting of digital services with AC capabilities. This needs to be performed by adopting a generic and adaptive way to tackle the complex needs of adaptation behavior of these systems. To this end, several characteristics are significant, such as *top-down vs. bottom-up approach*, *decentralized control*, *self-* features*, *context-awareness*, *reflexivity*, *quality attributes (e.g., evolvability*, *interoperability*, *scalability)* and *method validation* (see Fig. 3). These characteristics are intended to make the framework both theoretical and practical, in which *method validation* focuses on the practical side of a method while the other characteristics focus on its theoretical side.

The categories of the comparison framework are based on the NIMSAD framework [56]. NIMSAD has been used in the development of a number of comparison frameworks in software engineering (e.g., [58,59]). NIMSAD defines four essential elements for evaluating a methodology: *method context*, *method user*, *method content* and *evaluation of method*. A distinctive feature of NIMSAD is its fourth element, evaluation, which is missing in many other similar frameworks [56]. For these reasons, NIMSAD has been selected in the present survey. The 13 research questions (RQs) established in Table 1 can be broadly categorized under these four categories (i.e., context: RQ1, user: RQ2, method: RQ3–RQ11, evaluation: RQ12–RQ13). First, in the context category, the analyzed method is examined from the problem situation point of view (see RQ1, Table 1). Second, in the user category, the method is examined from the viewpoint of the intended users of the method (RQ2). Third, the method contents category focuses on the content of the method itself (RQ3–RQ11). Finally, in the evaluation category, the validation details of the method are focused (RQ12–RQ13). Descriptions of each characteristic of the comparison framework are provided next.

Goal and expected benefits: First the goal of the analyzed method must be clearly defined. Also, the expected benefits of using the method need to be described.

Top-down versus bottom-up approaches: Autonomic systems can be characterized by their operating conditions and by multiple dimensional properties such as *top-down* and *bottom-up approaches*, and *centralization* and *decentralization* [36].

On the one hand, traditional *top-down approaches* can be adopted to engineer systems where specific functionalities or behavior is achieved by explicit design. On the other hand, *bottom-up approaches* (e.g., nature-inspired or bio-inspired approaches [21]) are used to achieve functionalities via spontaneous self-organization [17]. Both these approaches are beneficial where a top-down approach can be used to engineer specific local functionalities while the latter can be adopted to engineer large-scale behaviors. The line between these two approaches is often not clear, and a method can incorporate techniques from both alternatives.

Decentralized control: Adaptation logic can be decentralized, centralized or applied in a hybrid manner [37]. A method needs to define models and tools to support *decentralized control* so that both collective adaptation and adaptation by subparts can be provided. *Decentralization* (e.g., see [60]) is a feature of cooperative self-adaptive or self-organizing systems, which function without a central authority [36]. Decentralized systems are usually bottom-up and the large numbers of components contained in these systems interact locally according to simple rules, thus emerging the global behavior of the overall system. In a centralized system approach, a central unit controls the system, but this approach is not suitable for large systems due to its size and real-time constraints. Meanwhile, a hybrid approach has both centralized and decentralized elements [37,61]; thus, both collective adaptation and adaptation by subparts can be provided.

Self- features:* As described in Sect. 2.1, the four self-* characteristics (self-chop) considered as fundamental for any autonomic system, and as a result, most cited in the AC domain are *self-configuration*, *self-healing*, *self-optimization* and *self-protection* [42]. *Self-configuration* describes the adjustment of system components in a user independent manner to achieve overall system behavior according to higher-level goals. *Self-optimizing* is achieved when the system provides operational efficiency by tuning resources and balancing workload. Meanwhile, *self-healing* means that the system provides resiliency by discovering and preventing disruptions and recovers from malfunctions. *Self-protecting* means that the system secures critical assets and resources by anticipating, detecting and protecting against any security risks. Other than these self-chop properties, *self-adaptation* [36] is a key characteristic of an autonomic system. It is realized as a situation-based behavior that takes into consideration the functional and quality properties of the environment and system itself, and the needs of the users.

Context-awareness: The need for context-awareness (e.g., see [62,63]) is a recognized issue in complex adaptive systems such as DSEs [3]. Although acquiring data in order to support context-awareness is not an issue, handling significant amount of data is very challenging [17]. Also, awareness can encompass situations occurring not only at the locality of individual components but also at many different levels of the system. Therefore, in order to perform autonomous adaptation activities in a collective and coordinated way, they need to be driven by more comprehensive levels of awareness than the traditional context-aware computing models.

Reflexivity: Reflexivity is an important characteristic of a self-managed autonomic system, which means that the system must have knowledge of its components, current status, capabilities, limits, boundaries and interdependencies with other systems and available resources [64]. It is the capability of making intelligent decisions based on self-awareness. Also, the system must be aware of its possible configurations and how they affect specific non-functional, quality requirements. The knowledge processing is based on rules, machine learning algorithms and software agents. In the current study, we consider reflexivity as a technique that can be exploited to support evolution (evolvability) of the ecosystem.

Although *reflexivity* is a relatively new term in service engineering, *reflection* is a widely known mechanism that can be used to support reactive or proactive adaptation of software systems. *Reflection* is defined as the ability of software to examine and modify its structure or behavior at runtime [65,66]. Reflection can be of two types: introspection and intercession. Introspection is the observation of an application's own behavior, while intercession is the reaction on introspection's results, which can be structural, parameter or context adaptation [67]. Reflection techniques have been investigated with self-adaptive systems as an underlying principle for self-awareness on different levels of software, e.g., architectural reflection [68], behavioral reflection. However, these methods apply reflection on the software itself, while we consider reflexive behavior with respect to unanticipated changes at the larger ecosystem level to support evolution of the ecosystem, and not at the system level.

Quality attributes: Non-functional requirements describe the qualities of the system. From service development point of view, QoS defines a set of quality attributes that a particular service has to fulfill. As a consequence, quality attributes defined in the QoS specification of a service system has to be dealt in each software engineering phase: in requirements specification, architecture design and implementation.

As discussed in Sect. 2.2.1, quality attributes can be categorized as execution and evolution quality attributes. While all these attributes are important, however, in this survey, we only focus on quality attributes that are significant from the ecosystem viewpoint of service engineering of digital services (e.g., evolvability, interoperability, scalability).

Evolvability: By evolvability we refer to the ability of the ecosystem to evolve in dynamic situations (for example, see

[29]). An ecosystem is dynamic, evolving all the time as new members, services and value networks emerge [3]. Therefore, in order to adapt to the needs of the ecosystem, the ecosystem's knowledge management model should evolve too. Additionally, new support services need to emerge as and when required. As new requirements emerge, requirements innovation is a continuous process inside the ecosystem.

Interoperability: Interoperability is the ability of software to exchange information and to provide something new, which originates from exchanged information [69]. The main goal of interoperability models and rules is to enable the loosely coupled services to collaborate. In [53], six interoperability levels have been defined for smart environments, i.e., conceptual, behavioral, dynamic, semantic, communication and connection. In order to support ecosystem interoperability, four interrelated metamodels have been proposed in [70], which are domain ontology, methodology, domain reference model and knowledge management metamodels.

In DSEs, proper service engineering techniques are required to develop digital services that are interoperable, available and easily consumed by taking into consideration the specific capabilities of the ecosystem [3]. In order to support *service interoperability*, two main elements are required by the ecosystem to engineer services in an ecosystem: ecosystem infrastructure and knowledge repositories [71]. Ecosystem infrastructure makes services interoperable, available and easily consumed and therefore manages all service ecosystem operations. Meanwhile, storage of the collaboration models, service descriptions and ontologies of service types to support interoperability are provided by knowledge repositories. Other than *service interoperability*, *pragmatic interoperability* is achieved between ecosystem members when their intentions, business rules and organizational policies are compatible [3]. *Pragmatic interoperability* deals with context data, which is specified as internal state of the system [71]. It also deals with the specification of the system process that employs the data. For examples and usage of service and pragmatic interoperability, refer to [71].

Scalability: In general, scalability in software engineering has been commonly known as the ability of a system, network or process to handle growing amounts of work in a graceful manner or its ability to be enlarged to accommodate that growth. A formal definition of scalability for digital ecosystems has been provided in [72] as: "to a certain degree, a digital ecosystem is scalable if its performance stays effective and efficient while large amount of input data or large quantities of heterogeneous participating entities are added."

The component model for a DSE can potentially include a very large scale of target scenarios; thus, it must promote scalability of both design (i.e., software engineering scalability [73,74]) and execution complexity (i.e., performance scalability [73]). In other words, the component model for a DSE should be based on sound design principles that can be practically applied to small systems and to very large systems, and the component model of the DSE needs to exhibit scalable performances and QoS.

Method validation: There should be some level of evidence regarding the maturity of the method, such as the evidence of its use and applicability. It is important to ascertain whether the method has concretized in several research papers. Also, the method should provide a way to validate its results. In this regard, a method can be applied at the conceptual level, as a proof of concept in the lab, or in the development of large-scale industrial product using a case study.

5 Overview and comparison of autonomic computing methods in digital service ecosystems

This section presents each of the 12 primary methods organized in the four research areas (see Sect. 3.2, Fig. 2) in greater detail. To this end, an overview of each primary method is provided followed by a comparison of the primary methods against the comparison framework (Tables 2, 3, 4). The four research areas are:

- AC methods in DSEs
- AC methods in service ecosystems
- quality-driven software engineering methods
- DASs-based methods that support self-* properties (in requirements engineering or architecting).

5.1 AC methods in DSEs

This research area includes the articles found explicitly using the AC initiative in the DSE domain. Digital ecosystems are not characterized by only one reference model as they crosscut different business domains and value chains [72]. As a consequence, architectures need mechanisms to allow the participants to publish any model and investigate on models that are most suitable to their needs. In order to handle these challenges, the AC initiative has been exploited in three main studies in the DSE domain, which are:

- *self-controlled components* [28,75]
- *evolving SOAs* [29]
- *autonomic SOA for DSEs* [30,76–78].

See Fig. 4 and Table 2 for a comparison of these three primary methods against the framework.

Table 2 Comparison summaries of the AC methods in DSEs

Method	Self-controlled components	Evolving SOAs	Autonomic SOA for DSEs
Comparison characteristic			
Goal	Unify concepts of components and services in cloud applications	Allow services recombine and evolve over time, increasing its effectiveness for the users	Achieve a more adaptive and a robust architecture for DSEs
Benefits	Providing strong QoS guarantees of composed applications	Maintaining self-organizing evolution of digital ecosystems in a scalable architecture	Improving usability, adaptability and robustness of SOA
Top-down versus bottom-up approach	Top-down	Bottom-up	Top-down
Decentralized control	MAPE loops at the top global level, and interactions that occur through the hierarchical component model provide a high level of decentralized control	Architecture contains two optimization levels and a high level of decentralized control supported	MAPE-K loops support some degree of decentralized control
Self-* features	Self-reconfiguration, self-adaptation, self-management	Self-organization, self-management	Self-organization
Context-awareness	Not supported	Not supported	Basic UML metamodel to model the architecture with domain-independent stereotypes for adaptation handling
Reflexivity	Not supported	Not supported	Not supported
Quality attributes	Availability, integrity, time and capacity	Evolvability, scalability (robustness mentioned as a benefit but not supported)	None
Evolvability	Not supported	Self-organizing evolution of digital ecosystems	Not supported
Interoperability	Not supported	Not supported	Not supported
Scalability	Not supported	Push-oriented approach to support scalable architectures	Not supported
Method validation	Model applied to a Springoo application providing online merchant applications	Simulating an EOA-based digital ecosystem against a traditional SOA system	Work-in-progress prototype of the proposed SOA in the computation engineering domain

Self-controlled components

Cloud computing and the future Internet create a new ecosystem where everything is a service with custom composition and dynamic management of resources at runtime. In this context, in the OpenCloudware project [28,75], the authors introduce a compositional framework to compose components as services which can be self-controlled. In self-controlled components (SCC), self-control mechanisms are attached to them to enable autonomic application management during execution. The objective is to provide strong QoS guarantees of composed applications.

The authors adopt a *top-down approach* from architectural modeling to service implementation, and runtime support including autonomic contract management. The SCCs are based on grid component model which the authors have extended with service-oriented features. Autonomy has been introduced in the SCCs using feedback control loops with elements of monitoring, analyzing, planning and execution of adaptation activities (MAPE loops). A high level of *decentralized control* is provided using MAPE loops defined at the top global level of the composition, and interactions that occur through the hierarchical component model. This method supports several *self-* properties* such as *self-reconfiguration*, *self-adaptation* and *self-management*. MAPE loops in an SCC provide self-reconfiguration with actions to change the component structure or dependencies between the involved components at runtime. Self-management of resources is supported for each QoS criterion in the quality model. A significant feature and contribution in their work is the support for QoS control and management. For this, each SCC has a QoS control component to ensure compliance with the service contract. It defines four QoS criteria: availability, integrity, time and capacity.

Table 3 Comparison summaries of the AC methods in service ecosystems

Method	SAPERE	CASCADAS	BIONETS	Self-reconfiguration for service ecosystems
Comparison characteristic				
Goal	Propose a nature-inspired, self-organizing framework for engineering distributed pervasive services	Introduce a model of an autonomic component for ecosystem's evolution through self-awareness and self-organization	Provision a service ecosystem for autonomic services fulfilling user demands in a transparent, efficient manner	Enable a service-based system to continuingly adjust its configuration using an autonomic loop
Benefits	Defining and implementing general-purpose nature-inspired pervasive service ecosystems (benefit of reference architecture)	Offering services seamlessly for human-to-human, human-to-machine and machine-to-machine interactions	Overcoming device heterogeneity and achieving scalability via an autonomic and localized peer-to-peer communication paradigm	Satisfying service-level agreements with minimal resource consumption
Top-down versus bottom-up approach	Bottom-up	Both top-down and bottom-up	Bottom-up	Top-down
Decentralized control	Exhibited by the bottom-up emergence of self-organized patterns of coordinated behaviors	High-level of decentralized control with self-organization capabilities defined in the self-model	Supported to allow services to adapt and evolve at the component and global ecosystem levels	MAPE loops aimed at providing some degree of decentralized control
Self-* features	Self-adaptation, self-organization, self-management	Self-awareness, self-organization, self-management	self- configuration, self-healing, self-optimization, self-protection	Self-reconfiguration
Context-awareness	Variety of adaptive, self-organizing patterns used to provision distributed pervasive services	Extensively defined as a set of extended finite state machines (self-model)	T-Nodes, U-Nodes, and access points form the architecture providing context-aware services	Heuristics used to formalize a basic model of configuration and reconfiguration definitions
Reflexivity	Not supported	Not supported	Not supported	Not supported
Quality attributes				
Evolvability	Diversity and evolution of the ecosystem supported by four metaphors	Evolvability of the ecosystem by programming the self-model of the ACEs	Evolution built on self-organization at single components (micro) and global ecosystem (macro) levels	Not supported
Interoperability	Mechanism on how to explicitly externalize knowledge out of services and carry out interactions	Not supported	Not supported	Not supported
Scalability	Highlighted as one of the main challenges of data storage and analysis in pervasive, mobile computing	Architecture is scalable in several dimensions, such as memory, threads and communication delay	Scalability through an autonomic and localized peer-to-peer service-driven communication paradigm	Not supported
Method validation	Middleware validated using use cases on information and guidance services in a smart museum	Simulations using a use case on a decentralized server farm	Simulation and proof-of-concept implementation for a service mobility framework	Preliminary experiments (a sample system with three services); Method has not matured in several papers

Table 4 Comparison summaries of methods from DASs with self-* properties and quality-driven software engineering research areas

Research area	DASs-based methods that support self-* properties				Quality-driven software engineering methods
Method	Requirements reflection	Architectural styles for runtime adaptation	Digital evolution of behavioral models for autonomic systems	Evolutionary computation for DASs	Quality-driven analysis process based on URDAD
Comparison characteristic					
Goal	Propose a method to allow software systems to adapt continuously to changing environmental conditions	Present a method on architectural styles for runtime software evolution (dynamic adaptation)	Apply digital evolution to the design of software exhibiting self-* properties	Describe a process and a suite of tools to support development of DASs	Propose quality criteria, and quality drivers for a service-oriented methodology for requirements engineers
Benefits	Supporting self-adaptive systems to discover, reason about and manage requirements during runtime	Changing system's functionality during runtime without reloading or restarting the system	Providing means to produce economical software solutions that exhibit robustness, flexibility and adaptability	Self-reconfiguring of DASs, and generating safe adaptations unanticipated at design time	Defining standalone service contracts and assemble functions from abstract, reusable, stateless services
Top-down versus bottom-up approach	Top-down	Top-down	Top-down	Top-down	Top-down
Decentralized control	Not supported	Architectural styles provide decentralized control of the architecture	Not supported	Not supported	Not applicable
Self-* features	Self-adaptation	Self-adaptation	Targets all self-* properties including self-adaptation	Self-reconfiguration	Not applicable as not targeting the AC initiative
Context-awareness	Goal-oriented requirement models extended with the RELAX language to model uncertainty	Set of architectural styles provides runtime software adaptation and evolution	Not supported	Goal models to model sources of uncertainty	Not applicable
Reflexivity	Requirements reflection supports self-adaptation by using a runtime goal model and qualitative and quantitative reasoning on it	Not supported	Not supported	Not supported	Not applicable

Table 4 continued

Research area	DASs-based methods that support self-* properties				Quality-driven software engineering methods
Method	Requirements reflection	Architectural styles for runtime adaptation	Digital evolution of behavioral models for autonomic systems	Evolutionary computation for DASs	Quality-driven analysis process based on URDAD
Comparison characteristic					
Quality attributes	Not supported	Dynamic adaptability	Not supported although it is mentioned that the method can produce software that exhibit robustness, flexibility, adaptability	Not supported	Comprehensive set of quality requirements and drivers specified for both quality model and process
Evolvability	Evolution of the requirements model and synchronization with the architecture	Runtime software evolution by a range of architectural styles with respect to a four-element evaluation framework	Digital evolution-based method used for generating behavioral models	Evolutionary algorithms used at design time for reconfiguration of DASs, and at runtime to generate safe adaptations which are unanticipated at design time	Not supported
Interoperability	Method applied to synthesize emergent middleware to achieve interoperability	Not supported	Not supported	Not supported	Not supported
Scalability	Not supported	Not supported	Model abstraction and incremental development features address potential scalability issues	Not supported	Not supported
Method validation	Synthesize emergent middleware to achieve interoperability; method matured in several research papers	No details of method validation have been reported and the work is at the conceptual level	Validation based on two case studies, i.e., autonomous robot navigation system and an intelligent and adaptable flood monitoring and warning system	Validation is based on a simulation on an intelligent vehicle system, and experiments on robotics	Internal consistency validated by using it to design a service-oriented analysis and design methodology

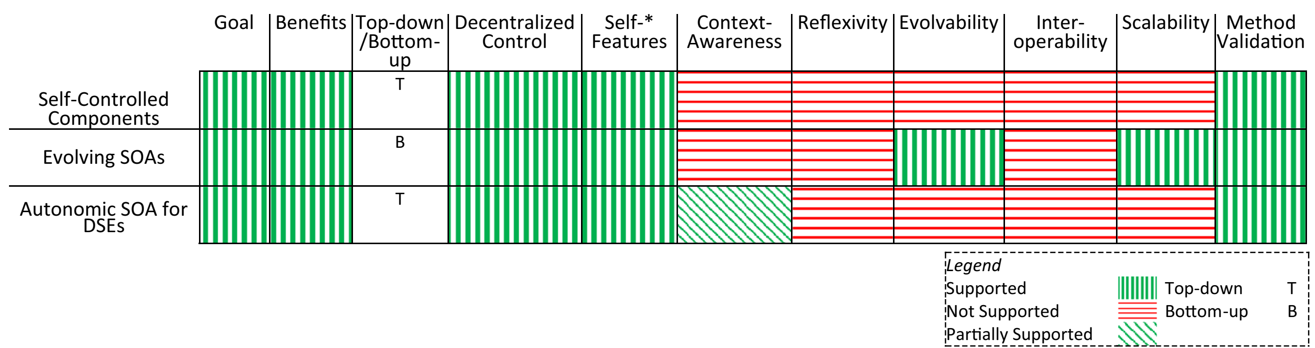

Fig. 4 AC methods in digital service ecosystems compared against the framework characteristics

The authors apply a use case called Springoo to show how the adoption of SCCs helps the service composition to provide a guarantee of QoS. Springoo is a web application providing online merchant applications using Apache/Jonas/MySQL components. The architecture has been partially *validated* using the grid component model/ProActive middleware to provide the monitors, QoS control and MAPE components. While QoS guarantees have been comprehensively provided in their work, a detailed *context-awareness* model to support autonomous adaptation activities is missing. Also, *reflexivity* and *quality attributes* of *evolvability*, *interoperability* and *scalability* of the ecosystem have not been addressed in [28,75].

Evolving SOAs

Briscoe and De Wilde [29] have presented a largely *bottom-up method* called an ecosystem-oriented architecture (EOA) of digital ecosystems by extending SOA with distributed evolutionary computing, thus allowing services to recombine and evolve over time and increasing its effectiveness for the users. Here, the word ecosystem is more than just a metaphor. Digital ecosystems, which are digital counterparts of biological ecosystems, have been defined as software systems that exploit the properties of biological ecosystems. They can automatically solve dynamic and complex problems, such as robustness, scalability and self-organization.

The architecture of the digital ecosystem provides a two-level optimization scheme, and there a high level of *decentralized control* has been supported. The underlying tier of distributed agents consists of a decentralized peer-to-peer network. The second optimization level is based on an evolutionary, genetic algorithm that operates locally on single habitats (peers). The *self-* properties* supported are *self-organization* and *self-management* (see Table 2). The digital ecosystem here is a multi-agent system, which uses distributed evolutionary computing to combine appropriate agents so that user requests for applications are satisfied. In this architecture, each service is a habitat and the network of habitats creates the digital ecosystem. The continuous changing requirements and their complexity in contextual, adaptive environments are the driving force for the *evolution* and self-organization of agents. The authors consider two main models from several variants of distributed evolutionary computing: the coarse-grained island model and the fine-grained diffusion neighborhood model. However, they propose to use a reconfigurable network topology so that habitat connectivity can be dynamically adapted based on the observed migration paths of the agents in the habitat network.

One of the key features and benefits of authors' work is the support for *scalable architectures* in order to meet user requests for applications. This has been fulfilled by applying a fundamental paradigm shift where a push-oriented approach has been used instead of a pull-oriented one. In a push-oriented approach, the digital ecosystem composes applications preemptively and upon request. On the other hand, a pull-oriented approach generates applications only upon request in SOAs. This method has been *validated* using a simulation of an EOA-based digital ecosystem. This EOA-based simulation has been compared against a simple SOA with a distributed UDDI (Universal Description, Discovery and Integration) service registry. The results indicate that with the increasing number of services, the digital ecosystem outperformed the traditional SOA system. Maintaining the self-organizing *evolution* of digital ecosystems in a *scalable architecture* is a main benefit in the authors' method. However, Briscoe and De Wilde [29] have not considered *context-awareness*, *reflexivity* and *interoperability* in their method.

Autonomic SOA for DSEs

In [30,76–78], following a *top-down approach*, the SOA has been extended with the AC initiative (*autonomic SOA)* to achieve a more adaptive and a robust architecture for DSEs. This is to keep up with the dynamic changes of requirements and environment. The authors elaborate on the design and implementation model of an autonomic SOA using a case study in computational engineering. Compared to traditional SOA, the *autonomic SOA* technique includes an autonomic manager and a knowledge base, which provides the ability to adapt to changes.

	Goal	Benefits	Top-down /Bottom-up	Decentralized Control	Self-* Features	Context-Awareness	Reflexivity	Evolvability	Inter-operability	Scalability	Method Validation
SAPERE	Supported	Supported	B	Supported	Supported	Supported	Not Supported	Supported	Partially Supported	Partially Supported	Supported
CASCADAS	Supported	Supported	T,B	Supported	Supported	Supported	Not Supported	Supported	Not Supported	Supported	Supported
BIONETS	Supported	Supported	B	Supported	Supported	Supported	Not Supported	Supported	Not Supported	Supported	Supported
Self-reconfiguration for Service Ecosystems	Supported	Supported	T	Supported	Supported	Partially Supported	Not Supported	Not Supported	Not Supported	Not Supported	Partially Supported

Fig. 5 AC methods in service ecosystems compared against the framework

The proposed architecture of the autonomic, *self-organizing* SOA contains three layers: presentation layer, process layer and service/resource layer. The presentation layer, which is the top-most layer of the architecture, provides an interface for various users. The processing layer, which is the middle layer, performs and coordinates autonomic functionality. The bottom resource layer provides utilization of the distributed resources using web services. It is provided as a typical SOA framework that contains a service registry and service providers. The service registry's functionality has been extended by introducing a knowledge base for the autonomic processes. The actual autonomic concept has been provided by an autonomic manager in the processing layer. The autonomic manager performs the autonomic cycle of monitoring, analysis, planning and execution over the knowledge base (MAPE-K loop), and this provides some degree of *decentralized control*. The authors have used Unified Modeling Language (UML)-based metamodeling concept to model the proposed architecture as a UML class diagram (basic metamodel for *context-awareness*). There several domain-independent stereotypes have been used to specify the relationships (e.g., request, call, instance, publish, find and bind) between the components (e.g., user, autonomic manager, composer, brokers, service registry and service provider) in the architecture. Also, a sequence diagram has been derived of the service architecture.

The method has been *validated* using a case study in computational engineering [76], and an implementation of a work-in-progress prototype of the proposed SOA has been presented. However, *reflexivity* and *quality attributes (e.g., evolvability*, *interoperability*, *scalability)* have not been addressed in the method.

5.2 AC methods in service ecosystems

In the following, we discuss four primary methods that apply the AC initiative in the domain of service ecosystems. As mentioned in Sect. 2.2, service ecosystems are socio-technical complex systems where service providers can reach shared goals and utilize the services of other members in the ecosystem to gain added value [44,45]. Service ecosystems are closely related to the research areas of Internet of Services (IoS) and service value networks. IoS considers the Internet as a global platform for retrieving, combining and utilizing interoperable services. Meanwhile, service value networks [79] provide business value through agile and market-based composition of complex services. This is from a pool of service modules by the use of a universally accessible network orchestration platform. The reviewed four primary methods are:

- *SAPERE* [15–20]
- *CASCADAS* [21–23]
- *BIONETS* [24–27]
- *service reconfiguration for service ecosystems* [80].

Figure 5 and Table 3 provide a comparison of these four primary methods.

SAPERE

In [15–20], the authors propose a nature-inspired reference architecture called *SAPERE (Self-Aware Pervasive Service Ecosystems)*, which can be a useful guide in the design and implementation of self-adaptive pervasive service ecosystems. They identify several research challenges emerging from the convergence of cyber-physical worlds, such as comprehensive situation-awareness, top-down vs. bottom-up design, power of masses, decentralized control, and diversity and evolvability [19].

The authors explain how the SAPERE middleware infrastructure supports the SAPERE model and framework. SAPERE middleware has followed a *bottom-up approach* getting inspiration from natural systems. *Decentralized control* is exhibited by the bottom-up emergence of self-organized patterns of coordinated behaviors. The *self-**

properties supported are *self-adaptation*, *self-organization* and *self-management* (see Table 3). In the SAPERE framework, pervasive services are modeled and deployed as autonomous individuals in an ecosystem of other services and devices. All of these interact according to a limited set of self-organizing, self-adaptive coordination laws called eco-laws. The provisioning of distributed pervasive services is realized by a variety of adaptive, self-organizing patterns (*context-awareness* support). The authors survey and analyze a number of natural metaphors that can be adopted in the modeling and architecting of innovative pervasive service ecosystems. This is to support spatiality, adaptability, openness and long-lasting *evolvability* of the ecosystem. The key metaphors introduced are physical, chemical, biological and social, and the key differences between them are the way the species, space and eco-laws are modeled and implemented. They have discussed how diversity and evolution of the ecosystem can be supported by these four metaphors. On *interoperability*, this has only been partially addressed in [20] where they explain on a mechanism on how to explicitly externalize knowledge out of services and use it to carry out interactions. The authors highlight *scalability* as one of the main challenges of data storage and analysis in pervasive and mobile computing [17].

The authors' method has matured and evolved in many research papers [15–20]. The middleware implemented has been *validated* in the context of exemplary use cases on information and guidance services in a smart museum. Although the need for *interoperability* and *scalability* has been highlighted as important characteristics in the reference architecture, it is not clear how the implemented middleware infrastructure supports these qualities. Also, *reflexivity* has not been supported in their method.

CASCADAS

In the EU project *CASCADAS (Component-ware for Autonomic Situation-aware Communications, and Dynamically Adaptable Services)* [21–23], the authors introduce a model of an autonomic component to support the evolution of the ecosystem through self-awareness and self-organization. The architecture of the ecosystem is based on distributed autonomic components called *autonomic communication elements* (ACE). The internal behavior of ACE is described by means of a declarative representation called the *self-model*.

CASCADAS has elements of both *top-down* and *bottom-up approaches* where autonomic mechanisms have been included using a top-down approach while bio-inspired mechanisms are provided through a bottom-up approach. A high level of *decentralized control* is supported as self-organization capabilities are part of the ACE autonomic behavior defined within the self-model. The *self-* properties* supported are *self-awareness*, *self-organization* and *self-management*. Their work supports a detailed level of *context-awareness* with its self-model, which is defined as a set of extended finite state machines. These state machines include rules for modifying them to adapt ACE behavior to the changes of internal and environmental conditions. Explicit support for quality attributes has not been mentioned in [21–23], but *evolvability* of the ecosystem is provided by programming the self-model of the ACEs. Using experiments, the authors have shown that the ACE architecture is *scalable* in several dimensions, such as memory, threads and communication delay. Thus, the applicability of the ACE model in large autonomic communication scenarios is clear.

The CASCADAS method has been experimentally *validated* using simulations of a use case concerning a decentralized server farm, as part of a complex service ecosystem. But *reflexivity* and *interoperability* have not been supported in CASCADAS [21–23].

BIONETS

The *BIONETS (BIOlogically inspired autonomic NETworks and Services)* project [24–27], which is a European Commission FET (Future and Emerging Technologies) initiative on Situated and Autonomic Communications, aims at enabling autonomic pervasive computing environments through the introduction of biologically inspired approaches. The project uses evolutionary techniques embedded in the system components as means to achieve full autonomic behavior. BIONETS looks at how nature and biology in particular (e.g., chemical computing, artificial embryogenies and evolutionary games) can be used to achieve self-chop features through open-ended evolution [24]. The authors describe four main challenges stemming from Future Internet scenarios: scale, heterogeneity, complexity and dynamicity [24]. The overall goal of BIONETS is provisioning of a service ecosystem for autonomic services. This service ecosystem needs to be able to fulfill user demands and needs in a transparent, efficient manner by exploiting the unique features of pervasive computing and communication environments.

Like the SAPERE method, BIONETS also follows largely a *bottom-up approach* where it gets inspiration from nature to build a distributed autonomic system based on local interactions. *Decentralized control* has been provided to allow services to adapt and evolve at the component level and global ecosystem level. BIONET places greater emphasis on four specific AC initiative properties, which are *self- configuration, self-healing*, *self-optimization* and *self-protection* [24] (see Table 3). There are three main actors in BIONET networks with respect to devices: T-Nodes, U-Nodes and access points [25]. T-Nodes gather data from the environment and are read by U-Nodes, which are complex, powerful devices passing by the T-Nodes. U-Nodes use T-Nodes to interact with the environment and gather information to run the context-aware services (*context-awareness* support). Access

points are complex powerful devices that act as proxies between BIONETS networks and IP networks. The BIONET project is built on two main pillars of networks and services, which converge to provide a full autonomic environment for network services. The latter is provided by self-evolving services, which is a bio-inspired platform, centered on the notion of *evolution*. *Evolution* here builds on the notion of self-organization, and it has been considered at two levels: single components (micro) and global ecosystem (macro). At the single component level, each service is able to design and build its own protocol stack and its own network. On the other hand, at the global ecosystem level, the interactions among service entities provide the means for rapid service evolution at the same time maintaining global stability properties. BIONETS achieves *scalability* through an autonomic and localized peer-to-peer service-driven communication paradigm [25].

Lahti et al. [26] present a *validation* of the BIONETS concepts as a simulation case and proof-of-concept implementation for a service mobility framework. However, like CASCADAS, *reflexivity* and *interoperability* have not been defined in their framework (see Table 3).

Self-reconfiguration for service ecosystems

Li et al. [80] propose an AC method to enable a service-based system to continue adjusting its configuration by means of an autonomic loop of monitoring, analyzing, planning and executing actions. Their *top-down approach* shows how AC initiative can be implemented to perform *self-reconfiguration* for service-based systems to satisfy two common metrics of non-functional requirements, i.e., response time of services and the system resource consumption. The focus of reconfiguration here is to satisfy non-functional requirements, and support for functional requirements, business requirements, constraints and quality attributes have not been mentioned. Their method focuses on the geometry configuration of service-based systems as opposed to dynamic reconfiguration exploited in traditional, distributed systems. The authors have used heuristics [80] to formalize a basic model of configuration and reconfiguration definitions (*context-awareness* support).

The main AC functions implemented to support self-reconfiguration of a service-based system include the following MAPE feedback loop activities: monitor to initiate reconfiguration; analyze to diagnose the configuration; plan to select reconfiguration; and execute for implementing reconfiguration. In addition, knowledge has been presented as a configuration of service-based systems described using architecture description standards, goals or policies. These MAPE loops provide some degree of *decentralized control* of the service-based system.

The authors have used preliminary experiments to *evaluate* their method. The method has been demonstrated using a service ecosystem, which provides mechanisms to dynamically change the location of services on machines while executing service requests. The service ecosystem here is a resilient service-operating environment in which the deployed services (e.g., grid services or web services) can be dynamically migrated in response to changing demand on resources to guarantee service-level agreements and to optimize resource utilization. However, their method [80] does not support *reflexivity* and any *quality attributes (e.g., evolvability*, *interoperability, scalability)*. Also, it has not matured in several research papers, and therefore, it is difficult to establish the applicability of their method more clearly (see Table 3).

5.3 DASs-based methods that support self-* properties

In the following, we discuss four primary methods selected for comparison from the *DASs-based methods that support self-* properties* research area. These are selected to be most relevant to our study, or these provide valuable lessons that can be learnt and applied from the DASs domain to the present context. The methods are from the perspective of service engineering, and these can be from requirements engineering and architecting phases of the software lifecycle (see Fig. 2). The four primary methods are:

- *requirements reflection* [81–84]
- *architectural styles for runtime adaptation* [85,86]
- *digital evolution of behavioral models for autonomic systems* [87–91]
- *evolutionary computation for DASs* [92,93].

DASs continuously monitor their environment and adapt behavior in response to changing environmental conditions [94]. In these systems, reconfiguration of software may need to be performed at runtime (e.g., software uploaded or removed) in order to handle new environmental conditions. Example domains that apply DASs include automotive systems, telecommunication systems, power grid management systems and ubiquitous systems.

Requirement reflection method supports runtime representation of requirements for DASs. Although there are several existing methods on requirements specification of DASs [94–96], the *requirements reflection* method supports the synchronization between requirements and architecture from which the current study can learn and draw parallels to the notion of *reflexivity* introduced here for DSEs. Thus, it has been selected for comparison here. In the same manner, at the architectural level, *architectural styles for runtime adaptation* method has comprehensive support for *context-awareness* modeling with their architectural styles for DASs.

Recently, there has been considerable interest within the software engineering research community (e.g., [87–93]) to

	Goal	Benefits	Top-down /Bottom-up	Decentralized Control	Self-* Features	Context-Awareness	Reflexivity	Evolvability	Inter-operability	Scalability	Method Validation
Requirements Reflection	Supported	Supported	T	Not Supported	Supported	Supported	Supported	Partially Supported	Partially Supported	Not Supported	Supported
Architectural Styles for Runtime Adaptation	Supported	Supported	T	Supported	Supported	Supported	Not Supported	Partially Supported	Not Supported	Not Supported	Not Supported
Digital Evolution of Behavioral Models for Autonomic Systems	Supported	Supported	T	Not Supported	Supported	Not Supported	Not Supported	Partially Supported	Not Supported	Supported	Supported
Evolutionary Computation for DASs	Supported	Supported	T	Not Supported	Supported	Supported	Not Supported	Partially Supported	Not Supported	Not Supported	Supported
Quality-driven Analysis Process based on URDAD	Supported	Supported	T	Not Applicable	Not Applicable	Not Applicable	Not Applicable	Not Supported	Not Supported	Not Supported	Supported

Legend: Supported; Not Supported; Partially Supported; Not Applicable; Top-down T; Bottom-up B

Fig. 6 DAS-based methods and quality-driven software engineering methods compared against the framework

apply evolutionary computation techniques for handling the threat of *uncertainty* [97] on adaptation capabilities of DASs. In [97], a taxonomy of potential sources of uncertainty from the DASs perspective has been presented with techniques for mitigating them. Evolutionary computation is a subfield of computer science which applies the basic principles of genetic evolution to problem-solving [91]. Digital evolution [98] is a branch or form of evolutionary computation. In digital evolution, self-replicating computer programs exist in a user-defined computational environment and are subject to mutations and natural selection. In this context, we analyze two primary methods, (1) *digital evolution of behavioral models for autonomic systems* and (2) *evolutionary computation for* DAS. Compared to other related methods in evolutionary computation, these methods support self-* properties and, more importantly, they have matured in several research papers.

See Fig. 6 and Table 4 for a comparison of these four primary methods against the framework.

Requirements reflection

In [81–84], the authors following a *top-down approach* introduce a method for *requirements reflection*, which means making requirements available as runtime objects. Requirements reflection is important as future software systems will be self-managing and these systems need to adapt continuously to changing environmental conditions. Requirements reflection can support such self-adaptive systems by making requirements first-class runtime entities, allowing software systems to reason about, understand, explain and modify requirements at runtime. It supports *self-adaptation* by using a runtime goal model and qualitative and quantitative reasoning about how the goal model's organization changes over time.

Bencomo [81] classifies uncertainty and adaptations that a self-adaptive system has to face as foreseen, foreseeable and unforeseen. Several research challenges on requirements engineering of self-adaptive systems have been identified, such as dealing with uncertainty, runtime representation of requirements, *evolution* of the requirements model and synchronization with the architecture, and dynamic generation of software [81–84]. In order to deal with uncertainty, they use and extend goal-oriented requirements modeling (*context-awareness* support) with the RELAX language [99,100], which has been developed to support modeling and reasoning about uncertainty in design time and runtime models. Runtime representation of requirements has been achieved by providing language support for representing, navigating and manipulating instances of a metamodel for goal modeling such as the KAOS metamodel [101]. In order to facilitate *requirements reflection* and synchronization between the goals and the architecture, the authors propose a two-layer model, that is, a base layer that consists of runtime requirements objects and a metalayer that allows the dynamic manipulation of requirements objects. This results in two layers—one for requirements and one for architecture—and each has a casually connected base layer and a metalayer. For the dynamic generation of software, they recommend the use of generation and transformational techniques in software engineering.

The authors' research has matured in many research papers. In [84], their method has been applied to synthesize emergent middleware to achieve *interoperability* in the context of the CONNECT project [102]. In emergent middleware, mediators are synthesized from runtime models, which provide support to reason about interoperability issues. However, the authors do not mention on *decentralized control* and any *scalability* features of the architecture (see Table 4).

Architectural styles for runtime adaptation

Taylor, Medvidovic and Oreizy [85,86] present a *top-down* method on architectural styles for runtime software evolu-

tion. Runtime software *evolution* or dynamic adaptation is the ability of a software system's functionality to be changed during runtime without reloading or restarting the system [85]. Architectural styles are "named collections of architectural design decisions that (1) are applicable in a given development context, (2) constrain architectural design decisions that are specific to a particular system within that context, and (3) elicit beneficial qualities in each resulting system" [85]. The architectural styles considered in [85] are REST (representational state transfer), event-based, service-oriented and peer-to-peer, and these styles can be used to provide *decentralized control* of the architecture.

The main targeted *self-* property* is *self-adaptation* while architectural styles can be used to provide comprehensive level of *context-awareness* modeling. They assess a range of styles with respect to a four-element evaluation framework called BASE introduced previously in [86]. The BASE framework provides means for evaluating, comparing and combining techniques for runtime adaptation. The BASE framework can be applied to differentiate techniques based on the system model they operate on and on how the four key aspects of runtime change are confronted, i.e., behavior, asynchrony, state and execution context. Architectural styles provide a technique for representing quality properties in architectural models and supporting quality-aware architecture modeling process. Architectural styles and patterns promote different quality attributes, and in [85], the quality attribute—dynamic adaptability—has been supported.

The authors do not specify any details of validating their method [85]. The authors' work [85] does not support *reflexivity*, *interoperability* and *scalability* features of the framework. There are several other existing methods that leverage architectural styles to enable dynamic adaptation, such as the Rainbow framework [103], and Kramer and Magee's layered reference architecture for self-adaptation [104] which includes mechanisms to swap out components and/or connectors at runtime.

Digital evolution of behavioral models for autonomic systems

By leveraging the Darwinian evolution, in [87–91] the authors propose a software development methodology capable of producing self-* software. They investigate the application of digital evolution to the design of software that exhibit *self-* properties*. In their method, a population of computer programs can be found in a user-defined computational environment, and it is subject to mutations and natural selection. Applying digital evolution in DASs provides means to produce economical software solutions that exhibit robustness, flexibility and adaptability.

The authors' method has been applied to generate behavioral models that capture autonomic system behavior. Their model-driven engineering process for DASs follows a *top-down approach* using several phases, such as goals, requirements, design models and implementation. A digital evolution-based tool called Avida-MDE (Avida for model-driven development) has been developed for generating behavioral models, which satisfy requirements specified as scenarios and properties. The authors propose a development model with three stages: cultivation, evaluation and deployment [87]. The Avida-MDE tool extends the Avida digital evolution platform in three ways to support state diagram generation. They are: first, defining search space by providing instinctual knowledge, which is information available to an organism at birth; second, generating behavioral models using this instinctual knowledge; and third, evaluating an organism based upon how well its generated behavioral model satisfies the requirements using model checking tools. The authors highlight two *scalability* challenges and present how their method will scale when used with larger applications. The two challenges are, first, allowing organisms to evolve large and increasingly complex diagrams, and second, model checking of the diagrams to verify that the functional properties are satisfied [88]. These potential scalability challenges have been addressed through the model abstraction and incremental development features of their method [88].

The method has been *validated* using two main case studies. First, it has been applied to generate behavioral models, describing the navigation behavior of an autonomous robot navigation system [87,89,91]. Second, in [90], the method has been validated by applying it to an adaptive flood warning system. Their work has matured in several research papers. However, *decentralized control, context-awareness*, *reflexivity* and *interoperability* issues have not been defined in their method (see Table 4).

Evolutionary computation for DASs

In a related method to the preceding method, in [92,93] the authors describe a process and a suite of tools to support the development of DASs. Their *top-down approach* starts with requirements and moves through reconfigurable designs at runtime. They exploit the power of evolutionary computation into model-based development and runtime support of high-assurance DASs.

The authors have defined uncertainty that can arise in three different aspects of cyber-physical systems: physical environment, cyber environment and components themselves. The sources of uncertainty in these aspects can happen at runtime, design time and requirements, and the authors try to address uncertainty with three enabling technologies: model-based development, assurance and dynamic adaptation. They highlight several evolutionary computation methods, such as genetic algorithms, genetic programming, artificial life and digital evolution, and evolved artificial neural networks [92,93]. Novel *evolutionary algorithms* have been harnessed at both design and runtime. For example,

their method has been applied to evolve collective communication algorithms for a variety of distributed behaviors that include synchronization, quorum sensing, constructing networks, responding to attacks and reaching consensus [92,93]. At design time, evolutionary algorithms have been used to better explore possible conditions requiring a DAS to *self-reconfigure.* At runtime they have been applied to generate safe adaptations, which are unanticipated at design time. The authors use OLYMPUS [92] for handling environmental uncertainty in a DAS where a goal model (*context-awareness)* is defined as a point of reference at runtime.

The authors have devised a *case study* to demonstrate how a goal model supports the modeling, monitoring and reconfiguring an intelligent vehicle system (IVS), which needs to perform adaptive cruise control, lane keeping and avoid collisions [92]. A simulation has been built using the Webots simulation platform to demonstrate a version of the IVS application. Additionally, their work has been validated using experiments conducted in the context of robotics [93]. However, *decentralized control, reflexivity*, *interoperability* and *scalability* features are missing in their method [92,93].

5.4 Quality-driven software engineering methods

In DSEs, service systems and applications are integrated solutions from several service providers. Therefore, in order to achieve the intended quality of a digital service, quality goals of all the supporting services need to be satisfied too. Therefore, dealing quality in an ecosystem is a complex process which stresses the need for solid *quality-driven software engineering* methods.

Quality-driven software engineering methods (e.g., [105–114]) emphasize the importance of addressing quality attributes in the earliest possible phases of the software lifecycle like requirements engineering and architecture design. In Sect. 2.2.1, we defined quality and quality attributes and classified quality attributes as execution and evolution qualities. Quality attributes are gathered, categorized and documented as at least equally important requirements as functional requirements. The gained knowledge is used in requirements engineering and software architecture design phases. In [106], authors describe five key industrial software architecture design methods and compare how they address quality requirements in architecture design. The compared methods are *Attribute-Driven Design* [107], *Siemens' 4 Views (S4V)* [108], *Rational United Process 4+1 Views* [109], *Business Architecture Process and Organization* [110], and *Architectural Separation of Concerns* [111]. The goal of this research area is not to provide an exhaustive analysis of scientific literature on quality-driven software engineering methods, which is out of scope here. Yet, we analyze a key method, *quality-driven analysis process based on URDAD*, which has been selected for comparison as it provides comprehensive support for quality attributes from a service engineering perspective (i.e., a service-oriented methodology used by requirement engineers). Also, the classification of different *stakeholders* with quality requirements for both process and quality model can provide valuable insights on how quality can be similarly classified in an ecosystem-based method which has many users.

Table 4 (see also Fig. 6) presents a comparison of this method against the framework.

Quality-driven analysis process based on URDAD

Solms et al. [105] propose a *top-down*, quality-driven analysis and design process based on URDAD (Use-Case, Responsibility-Driven Analysis and Design). URDAD is a service-oriented methodology used by requirements engineers to design services. This method provides comprehensive support for *quality attributes* where the authors provide a set of quality requirements and drivers specified for both quality model and process. Quality drivers are activities that improve one or more process or model quality criteria [105]. URDAD is used to generate the computation independent models of the model-driven architecture with sufficient details so that it can be used directly as platform-independent models. The authors have defined a domain-specific language for URDAD, called URDAD-DSL, which can be used to specify syntactically correct URDAD models. They identify the stakeholders and their quality requirements for both *process* and *model* quality. Then, for each quality criterion a set of quality drivers have been provided, and how quality drivers are embedded within the URDAD methodology has been demonstrated. The authors identify the main stakeholders for the *model* as requirement engineers, architects, developers, quality assurance staff, project managers and clients. The corresponding quality requirements for the requirements model are simplicity, completeness, modifiability, consistency, decoupling, cohesion, reusability and traceability. Meanwhile, stakeholders defined for *process* are project managers, requirements engineers and clients, and their quality requirements are low cost, repeatability, estimatability, trainable, measurability, consistency and isolation.

The authors validate the internal consistency of their method by using it to design a service-oriented analysis and design methodology, which generates its own URDAD metamodel and process. Their work is not targeting the AC initiative; thus, support for *decentralized* control, *self-* properties*, *context-awareness* and *reflexivity* is not applicable. Although a comprehensive set of quality requirements and drivers have been specified for both quality model and process, quality attributes of *evolvability*, *interoperability* and *scalability* have not been mentioned in their method [105] (see Table 4).

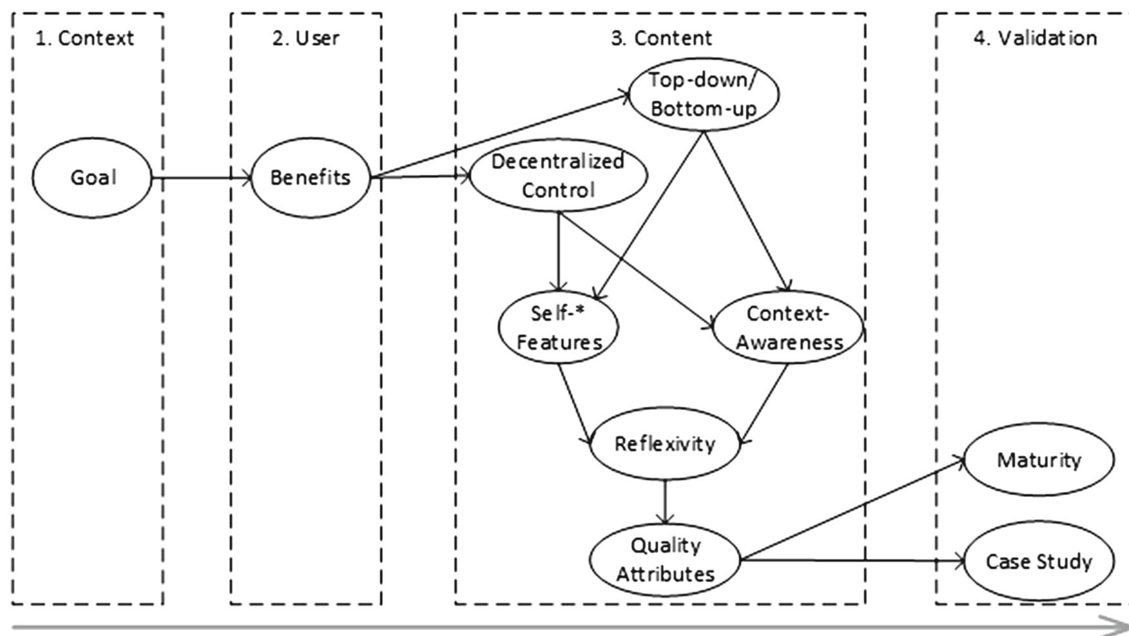

Fig. 7 Dependencies between the framework characteristics

6 Results of the survey

Given the characterization of the state of research, several observations can be made. In the following, these observations are summarized and discussed. In this process, we identify some open problems and insights to future work. Table 5 provides a summary of the results of this survey.

6.1 Main findings, open problems and future work

The comparison framework (Sect. 4) defined several characteristics which have been used to analyze and review the methods, i.e., *top-down or bottom-up approach*, *decentralized control*, *self-* features*, *context-awareness*, *reflexivity*, *quality attributes (e.g., evolvability*, *interoperability, scalability)* and *method validation*. We can identify dependencies between these different characteristics (see Fig. 7). This will assist in structuring the scientific literature and identifying the status of state of the art in the research areas with respect to the framework characteristics, that is, what has been achieved so far and what is missing. More importantly, this can serve as an effective starting point for defining a road map for service engineering digital service ecosystems with AC capabilities.

As discussed in Sect. 4, we have used the NIMSAD framework to examine a method from four different points of view sequentially: *context*, *user*, *method content* and *evaluation*. Here, we can identify a sequential dependency order between these categories (see Fig. 7). When we examine the individual characteristics within the *method content* category, *top-down vs bottom-up approach* and *decentralized control* precede all other characteristics, as these characteristics are reviewed from a higher level. Two characteristics are fundamentally important for *reflexivity*: *self-awareness* (self-* features) and *context-awareness*. *Reflexivity* is the capability of making intelligent decisions based on self-awareness (Sect. 4). We only consider *quality attributes* that are significant from the ecosystem viewpoint, such as evolvability, interoperability and scalability. Reflexivity can be identified as a technique that can be exploited to support *evolvability* of the DSE. Therefore, as shown in Fig. 7, the dependency between *reflexivity* and *quality attributes* can be described.

- The *goal* (from *context*) and *benefits* (from *user*) characteristics are specific to a particular method. These are only used to describe the purpose and advantages of using a method, and not directly used to analyze the results of the primary methods.
- In DSEs, both *top-down* and *bottom-up approaches* can be beneficial where a top-down approach can be used to engineer specific local functionalities while the latter can be adopted to engineer large-scale behaviors. Most analyzed primary methods (eight) have been top-down approaches, which achieve specific functionalities or behavior through explicit design. From the reviewed primary methods, *evolving SOAs*, *SAPERE* and *BIONETS* follow largely a bottom-up approach.

Sometimes the line between these two approaches is not clear and a method can incorporate techniques from both alternatives, and this has been exhibited by the *CASCADAS* method.

- As mentioned in Sect. 4, adaptation logic can be applied in a decentralized, centralized or hybrid manner [37]. A method needs to define models and tools to support *decentralized control* so that both collective adaptation and adaptation by its subparts can be provided. In general, decentralized systems are bottom-up and the components contained in them interact locally according to simple rules, thus emerging the global behavior of the overall system. A hybrid system has both centralized and decentralized elements within it. As a result, both collective adaptation and adaptation by subparts are provided. We note that all the methods in the DSE and service ecosystem domains provide some level of decentralized control. To this end, some popular techniques that have been applied are MAPE loops (e.g., *self-controlled components, autonomic SOA for DSEs*), or techniques getting inspiration from natural (e.g., *SAPERE*) or bio-inspired mechanisms (e.g., *CASCADAS*). However, except for the *architectural styles for runtime* adaptation method, decentralized *control* is largely missing in the primary methods reviewed from the DASs domain where the level of attention it has received is far less compared to the service ecosystem and DSE domains.
- *Self-configuration*, *self-healing*, *self-optimization* and *self-protection* are four self-* features considered as fundamental for any autonomic system [42]. The BIONETS project supports all these four characteristics, and *self-adaptation* and *self-management* have been the most supported self-* properties in the reviewed primary methods. *Quality-driven analysis process based on URDAD* method does not target the AC initiative, so support for self-* properties is not applicable there (see Table 5).

An open issue and challenge when applying the AC initiative to DSEs can be the open-loop structure of an ecosystem. Problems can be created by the open-loop structure of DSEs when incorporating autonomic software systems that have a closed-loop system. Typically an autonomic system consists of a closed-loop system. This means due to continuous changes of the system, it modifies itself at runtime using feedback. Software systems that incorporate closed-loop mechanisms allow them to adapt themselves to changing conditions, thus reducing human effort in the computer interaction. However, due to the open-loop structure of DSEs, there is a need for continuous human supervision, which is a challenge.

- The need for *context-awareness* is significant in complex adaptive systems such as DSEs [3]. There needs to be more comprehensive levels of awareness than the traditional context-aware computing models to support and drive the autonomous adaptation activities. Given the prominent place of *context-awareness* in complex adaptive systems, more efforts need to be directed at designing and developing these models in DSEs. This is because it is virtually absent from the literature except for the *autonomic SOA for DSEs* method, which defines a basic UML metamodel for adaptation handling. In contrast, support for *context-awareness* has received more attention in the service ecosystems domain. To this end, *CASCADAS* method defines a comprehensive context-awareness model called a self-model, which is defined as a set of extended finite state machines. In addition, *SAPERE* method defines a variety of adaptive and self-organizing patterns to provision distributed pervasive services. However, their work targets middleware infrastructures of pervasive services ecosystems.
- We see the opportunity and potential in combining the notions of *self-awareness* and *context-awareness* to obtain a further understanding of the situation in the context of *reflexivity* introduced for DSEs in the current study. In Sect. 4, we introduced the notion of *reflexivity* and compared it with existing *reflection* techniques. *Reflexivity* is an important characteristic of an autonomic self-managed system, which means that the system must have knowledge of its components to make intelligent decisions based on self-awareness [64]. For this, two characteristics are fundamental, which are *self-awareness* and *context-awareness*. One of the major deficiencies of the surveyed methods is that they did not commit to supporting reflexivity except for one work at the requirements level—*requirements reflection*. However, in there reflexivity has been applied to synchronize requirements model with architecture while in DSEs this needs to be applied at the larger ecosystem level to support the evolution of the ecosystem.
- With respect to support for *quality attributes*, the method —*quality-driven analysis process based on URDAD*— provides comprehensive support from a service engineering perspective where the authors describe a set of quality requirements and drivers specified for both quality model and process. The different *stakeholders* identified there can provide valuable insights on how quality can be similarly classified in an ecosystem-based method such as in the DSE domain which has many users.

Quality attributes can be categorized as execution and evolution quality attributes (Sect. 2.2.1). One challenge is to extend this characterization of quality criteria and properties to provide some metrics to measure them targeting the DSEs domain. Some of these quality attributes may be more easily guaranteed at runtime than at design time. In [115], authors discuss several adaptation properties defined as assurance criteria on the adaptation process, and mapped to quality attributes measurable at runtime for both the target system and the adaptation mechanism. However, it is targeting self-

Table 5 Results summary

Research area	Primary method	Framework characteristics										
		Goal	Benefits	Top-down versus bottom-up	Decentralized control	Self-* features	Context-awareness	Reflexivity	Quality attributes			Method validation
									Evolvability	Interoperability	Scalability	
AC methods in DSEs	Self-controlled components [28,75]	√	√	T	√	√	X	X	X	X	X	√
	Evolving SOAs [29]	√	√	B	√	√	X	X	√	X	√	√
	Autonomic SOA for DSEs [30,76–78]	√	√	T	√	√	•	X	X	X	X	√
AC methods in service ecosystems	SAPERE [15–20]	√	√	B	√	√	√	X	√	•	•	√
	CASCADAS [21–23]	√	√	T, B	√	√	√	X	√	X	√	√
	BIONETS [24–27]	√	√	B	√	√	√	X	√	X	√	√
	Self-reconfiguration for service ecosystems [80]	√	√	T	√	√	•	X	X	X	X	•
DASs-based methods that support self-* properties	Requirements reflection [81–84]	√	√	T	X	√	√	√	•	•	X	√
	Architectural styles for runtime adaptation [85,86]	√	√	T	√	√	√	X	•	X	X	X

Table 5 continued

Research area	Primary method	Framework characteristics										
		Goal	Benefits	Top-down versus bottom-up	Decentralized control	Self-* features	Context-awareness	Reflexivity	Quality attributes			Method validation
									Evolvability	Interoperability	Scalability	
	Digital evolution of behavioral models for autonomic systems [87–91]	√	√	T	X	√	X	X	•	X	√	√
	Evolutionary computation for DASs[92,93]	√	√	T	X	√	√	X	•	X	X	√
Quality-driven software engineering methods	Quality-driven analysis process based on URDAD [105]	√	√	T	*	*	*	*	X	X	X	√

√ supported, X not supported, • partially supported, * not applicable, T top-down approach, B bottom-up approach

adaptive systems and not the DSE domain. In this survey, we have focused on quality attributes that are important from the ecosystem viewpoint of service engineering of digital services like *evolvability*, *interoperability* and *scalability*.

- An ecosystem is dynamic and it *evolves* all the time as new members, services and value networks emerge [3]. Therefore, the service engineering methods need to support the evolution of the ecosystem. However, except for the *evolving SOAs*, *CASCADAS*, *BIONETS* and *SAPERE*, the other methods analyzed here do not support evolution from the ecosystem's perspective but from other viewpoints (e.g., individual system level). For example, *evolving SOAs* method introduces an ecosystem-oriented architecture of digital ecosystems; and there, evolutionary, genetic algorithms have been used to combine appropriate agents so user requests for applications are satisfied. In *CASCADAS*, evolvability of the ecosystem is programmed by the self-model of the ACEs. Meanwhile, in *BIONETS*, evolution builds on the notion of self-organization, where it has been considered at the single components level and global ecosystem level. However, a method that explores and combines *reflexivity* as a technique to support evolution of the ecosystem is completely missing in the literature. In the DASs domain, the primary methods do not support evolution from an ecosystem perspective; thus, this has been shown as partial satisfaction of that characteristic (see Table 5).
- One noteworthy quality attribute that is almost completely absent in the analyzed primary methods is *interoperability*, which can be of two types—*service interoperability* and *pragmatic interoperability* (see Sect. 4). In DSEs, proper service engineering techniques are required to develop digital services that are *interoperable*, available and easily consumed by considering specific capabilities of the ecosystem [3]. In *requirements reflection*, the authors have applied their method to dynamically synthesize emergent middleware that ensures interoperation between heterogeneous networked systems. However, in that method, *interoperability* has been considered from the network systems viewpoint (partial satisfaction of criteria, Table 5). Meanwhile, *pragmatic interoperability* has not been considered in any of the analyzed primary methods.
- A component model for a DSE can potentially include a very large scale of target scenarios. As a result, it must promote *scalability* of both design and execution complexity with *software engineering scalability* and *performance scalability*, respectively. The *evolving SOAs* method follows a push-oriented approach to support scalable architectures to meet user requests for applications. In *BIONETS*, scalability is achieved through an autonomic and localized, peer-to-peer service-driven communication paradigm. In *SAPERE*, authors have highlighted scalability as one of the main challenges of data storage and analysis in pervasive and mobile computing. However, it is not clear how it has been supported in the implemented middleware infrastructure [15]. While *software engineering scalability* has been addressed in some methods as mentioned above, *performance scalability* is absent in the analyzed primary methods.
- With respect to *method validation*, in most cases, the validation of the method is empirically based. One common aspect in all the primary methods is that they do not present well-designed quantitative or qualitative evaluations, but mainly focus on their own experience in the use of ad hoc methods or informal case studies. In this context, one of the main shortcomings and challenge we note is the lack of actual industrial case studies and scenarios on ecosystem-based digital services. The case studies need to exhibit situations for frequently evolving requirements; dynamic nature of the ecosystem; and digital services developed by several partners. An example of an industrial case study is the ICARE project [3,46] mentioned in Sect. 2.2. Yet, scenarios that identify afore mentioned complex situations in DSEs are completely missing in scientific literature. Also, there is a lack of publications covering some of the reviewed methods (e.g., *self-reconfiguration for service ecosystems*), and as a result, the applicability of those methods could not be clearly established.

In this manner, first identifying the dependencies between the framework characteristics and then analyzing what has been achieved so far and what is missing on the framework characteristics can provide an effective starting point for defining a road map for service engineering digital service ecosystems with AC capabilities.

As given in Table 5, it is clear that none of the analyzed primary methods entirely fulfills the requirements defined in the comparison framework. The goal is to compare the primary methods and not the projects. There are three main reasons for a method not fulfilling a particular characteristic of the comparison framework. They are: (1) the application domain—ecosystem-based service engineering is a natural progress of networking and pervasive computing, and therefore, the primary methods—SAPERE, CASCADAS and BIONETS—cover most of the characteristics of the framework; (2) the purpose of the method is more focused and naturally covers only one part of the framework topics. Thus, the intended usage context of the method is narrower than the comparison framework; and (3) the timing—newer publications are more relevant and consider the changes of the ecosystems, at least partially. According to the framework, the most suitable methods are

CASCADAS and *requirements reflection. CASCADAS* satisfies all the requirements of the framework except reflexivity and interoperability, and *requirements reflection* specifically supports reflexivity. In order to advance the findings of this survey, the use of solid modeling languages, techniques, tools and practices are highly desirable. Toward this end, *models@runtime* or *runtime models* [115] can be a key technique which can be explored toward addressing the shortcomings of the existing primary methods. The *requirements reflection* method has explored models@runtime to support synchronization between goal-based requirements and the architecture (evolution) and dynamically generate software artefacts at execution time. In the DSEs' context, future studies are needed to investigate on how models@runtime could be employed to mitigate uncertainty through runtime adaptation and evolution, and to the provision of digital services with autonomic capabilities.

Models@runtime are up-to-date abstractions of the running system [115]. They constitute a core concept for enabling adaptation and tackling uncertainty by reflecting the system and its context at runtime. Runtime models provide reflective capability as they are casually connected to the system being modeled. They can be utilized for two purposes: (1) supporting reasoning about uncertainty and leveraging self-adaptation and (2) supporting the generation of software artifacts themselves, using model-driven engineering at execution time [84]. Both runtime adaptation and evolution are necessary to address the frequent changes imposed by DSEs. Runtime adaptation and runtime evolution are two interwoven activities that influence each other [86]. Evolution can be understood as a longer sequence of modifications to a software system over its lifetime [116]. On the other hand, adaptations can be seen as modifications of the software system performed in an automated way.

7 Conclusions

This survey article presented a review of the scientific literature on AC methods in DSEs. Based on systematic queries in four leading scientific databases and Google Scholar, 349 articles were analyzed, out of which 12 primary methods were selected to be most relevant to our study from a service engineering perspective, which were then clustered, succinctly described and compared.

A comparison framework was defined as a contribution, which can be used as a guide for comparing the different scientific methods selected. To this end, the framework proposed several characteristics, such as *top-down vs. bottom-up approach*, *decentralized control*, *self-* features*, *context-awareness*, *reflexivity*, *quality attributes (e.g., evolvability*, *interoperability, scalability)* and *method validation*. The main contribution of this article is the comparison of the primary methods selected from the four thematic research areas. The comparison process using the framework was straightforward and uncomplicated. The framework is a valuable tool for searching for an applicable method on service engineering of digital service ecosystems with AC capabilities. It is evident from the comparison that none of the analyzed methods supports all the requirements of the framework. Furthermore, looking at the state of the art, none of the methods addresses in a generic and adaptive way the service engineering of DSEs, especially an ecosystem-based method on applying the AC initiative is missing in the DSE domain. This survey also introduced a technique called *reflexivity* for DSEs which can be further explored to address uncertainty and frequent changes imposed by DSEs.

We note two main perspectives for future research: (1) investigate how models@runtime can be employed as a technique to mitigate uncertainty through runtime adaptation and evolution in DSEs, and to provision digital services with autonomic capabilities. The reflective capability of runtime models can be explored to address the uncertainty and frequent changes imposed by DSEs at different lifecycle phases such as requirements, architecture design and runtime; (2) develop real industrial case studies and scenarios that exhibit situations for evolving requirements in DSEs, dynamic nature of DSEs and digital services developed by ecosystem members.

Acknowledgements This work was carried out during the tenure of an ERCIM "Alain Bensoussan" Fellowship Programme. This research has also been supported by a grant from Tekes—the Finnish funding agency for technology and innovation, and VTT as part of the Digital Health Revolution Programme.

Open Access This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

References

1. Chang E, West M, Hadzic M (2006) A digital ecosystem for extended logistics enterprises. In: Gaudes A (ed) Proceedings of the 11th international workshop on telework, August 28, International Telework, Academy, Fredericton, Canada, pp 32–40
2. Boley H, Chang E (2007) Digital ecosystems: principles and semantics. In: Proceedings of the international conference on digital ecosystems and technologies (DEST '07), 21–23 February, Cairns, Australia. IEEE, pp 398–403
3. Immonen A, Ovaska E, Kalaoja J, Pakkala D (2015) A service requirements engineering method for a digital service ecosystem. Serv Oriented Comput Appl 10(2):151–172
4. Kephart JO, Chess DM (2003) The vision of autonomic computing. Computer 36(1):41–50
5. Kephart JO (2005) Research challenges of autonomic computing. In: Proceedings of the 27th international conference on software engineering (ICSE 2005), 15–21 May. IEEE, pp 15–22

6. Parashar M, Hariri S (2004) Autonomic grid computing. In: Proceedings of the international conference on autonomic computing (ICAC'04), 17–18 May, New York. IEEE, pp xiv–xiv
7. Agarwal M, Bhat V, Liu H, Matossian V, Putty V, Schmidt C et al (2003) AutoMate: enabling autonomic applications on the grid. In: Proceedings of the autonomic computing workshop, 25 June. IEEE, pp 48–57
8. Rolón M, Martínez E (2012) Agent-based modeling and simulation of an autonomic manufacturing execution system. Comput Ind 63(1):53–78
9. McKee G, Varghese B (2012) Robotic ecologies for deep space outposts. In: Petrovic I, Korondi P (eds) Proceedings of 10th IFAC symposium on robot control, 5–7 September, Dubrovnik, Croatia. International Federation of Automatic, Control, pp 455–460
10. Golnaraghi F, Kuo BC (2010) Automatic control systems, 9th edn. Wiley, London
11. Wirsing M, Hölzl M, Tribastone M, Zambonelli F (2013) ASCENS: engineering autonomic service-component ensembles. In: Beckert B, Damiani F, de Boer FS, Bonsangue MM (eds) Formal methods for components and objects. Springer, Berlin, pp 1–24
12. Abeywickrama DB, Bicocchi N, Zambonelli F (2012) SOTA: towards a general model for self-adaptive systems. In: Proceedings of the 2012 IEEE 21st international workshop on enabling technologies: infrastructure for collaborative enterprises (WETICE'12), 25–27 June, Toulouse, France. IEEE, pp 48–53
13. Mayer P, Klarl A, Hennicker R, Puviani M, Tiezzi F, Pugliese R et al (2013) The autonomic cloud: a vision of voluntary, peer-2-peer cloud computing. In: Proceedings of the 2013 IEEE 7th international conference on self-adaptation and self-organizing systems workshops, 9–13 September, Philadelphia, USA. IEEE, pp 89–94
14. Amoretti M (2010) Towards fully autonomic peer-to-peer systems. Proc Comput Sci 1(1):2639–2648
15. Castelli G, Mamei M, Rosi A, Zambonelli F (2015) Engineering pervasive service ecosystems: the SAPERE approach. ACM Trans Auton Adapt Syst 10(1):1:1–1:27
16. Viroli M, Pianini D, Montagna S, Stevenson G, Zambonelli F (2015) A coordination model of pervasive service ecosystems. Sci Comput Program 110:3–22
17. Conti M, Das SK, Bisdikian C, Kumar M, Ni LM, Passarella A et al (2012) Looking ahead in pervasive computing: challenges and opportunities in the era of cyber-physical convergence. Pervasive Mob Comput 8(1):2–21
18. Zambonelli F, Viroli M (2011) A survey on nature-inspired metaphors for pervasive service ecosystems. Int J Pervasive Comput Commun 7(3):186–204
19. Zambonelli F, Mirko V (2010) From service-oriented architectures to nature-inspired pervasive service ecosystems. In: Omicini A, Viroli M (eds) Proceedings of the 11th national workshop from objects to agents (WOA 2010), 5–7 September, Rimini, Italy. CEUR-WS.org
20. Quitadamo R, Zambonelli F, Cabri G (2007) The service ecosystem: dynamic self-aggregation of pervasive communication services. In: Proceedings of the first international workshop on software engineering for pervasive computing applications, systems, and environments (SEPCASE '07), 20–26 May, Minneapolis, USA. IEEE
21. Manzalini A, Brgulja N, Moiso C, Minerva R (2012) Autonomic nature-inspired eco-systems. In: Gavrilova M, Tan CJK, Phan C (eds) Transactions on computational science XV. Springer, Berlin, pp 158–191
22. Manzalini A, Brgulja N, Minerva R, Moiso C (2012) Specification, development, and verification of CASCADAS autonomic computing and networking toolkit. In: Cong-Vinh P (ed) Formal and practical aspects of autonomic computing and networking: specification, development, and verification, pp 65–96
23. Baresi L, Di FA, Manzalini A, Zambonelli F (2009) The CASCADAS framework for autonomic communications. In: Vasilakos AV, Parashar M, Karnouskos S, Pedrycz W (eds) Autonomic communication. Springer, Berlin, pp 147–168
24. Miorandi D, Carreras I, Altman E, Yamamoto L, Chlamtac I (2008) Bio-inspired approaches for autonomic pervasive computing systems. In: Liò P, Yoneki E, Crowcroft J, Verma DC (eds) Bio-inspired computing and communication. Springer, Berlin, pp 217–228
25. Simon V, Bacsardi L, Szabo S, Miorandi D (2007) BIONETS: a new vision of opportunistic networks. In: Proceedings of the IEEE wireless rural and emergency communications conference (WRECOM'07), 1–2 October, Rome, Italy. IEEE
26. Lahti J, Rivas H, Huusko J, Könönen V (2010) simulation and implementation of the autonomic service mobility framework. In: Altman E, Carrera I, El-Azouzi R, Hart E, Hayel Y (eds) Bioinspired models of network, information, and computing systems. Springer, Berlin, pp 65–76
27. De Pellegrini F, Miorandi D, Linner D, Bacsardi L, Moiso C (2007) BIONETS architecture: from networks to SerWorks. In: Proceedings of the 2nd bio-inspired models of network, information and computing systems (BIONETICS 2007), 10–12 December, Budapest. IEEE, pp 255–262
28. Aubonnet T, Henrio L, Kessal S, Kulankhina O, Lemoine F, Madelaine E et al (2015) Management of service composition based on self-controlled components. J Internet Serv Appl 6(1):15. doi:10.1186/s13174-015-0031-7
29. Briscoe G, De Wilde P (2006) Digital ecosystems: evolving service-orientated architectures. In: Proceedings of the 1st bio-inspired models of network, information and computing systems conference (BIONETICS 2006), 11–13 December, Madonna di Campiglio. IEEE, pp 1–6
30. Bhakti MAC, Abdullah AB, Jung LT (2010) Autonomic, self-organizing service-oriented architecture in service ecosystem. In: Proceedings of the 2010 4th IEEE international conference on digital ecosystems and technologies (DEST), 13–16 April, Dubai. IEEE, pp 153–158
31. Nami MR, Bertels K (2007) A survey of autonomic computing systems. In: Proceedings of the 3rd international conference on autonomic and autonomous systems (ICAS'07), 19–25 June, Athens. IEEE
32. Huebscher MC, McCann JA (2008) A survey of autonomic computing—degrees, models, and applications. ACM Comput Surv 40(3):7:1–7:28
33. Khalid A, Haye MA, Khan MJ, Shamail S (2009) Survey of frameworks, architectures and techniques in autonomic computing. In: Proceedings of the fifth international conference on autonomic and autonomous systems (ICAS'09), 20–25 April, Valencia. IEEE, pp 220–225
34. Zhao Z, Gao C, Duan F (2009) A survey on autonomic computing research. In: Proceedings of the Asia-Pacific conference on computational intelligence and industrial applications (PACIIA 2009), 28–29 November, Wuhan, pp 288–291
35. Rahman M, Ranjan R, Buyya R, Benatallah B (2011) A taxonomy and survey on autonomic management of applications in grid computing environments. Concurr Comput Pract Exp 23(16):1990–2019
36. Macías-Escrivá FD, Haber R, del Toro R, Hernandez V (2013) Self-adaptive systems: a survey of current approaches, research challenges and applications. Expert Syst Appl 40(18):7267–7279
37. Krupitzer C, Roth FM, VanSyckel S, Schiele G, Becker C (2015) A survey on engineering approaches for self-adaptive systems. Pervasive Mobile Comput 17:184–206

38. Stevenson A (2010) Oxford dictionary of English, 3rd edn. Oxford University Press, Oxford
39. Randall AL, Walter RC (2003) Overview of the small unit operations situational awareness system. In: Proceedings of the military communications conference (MILCOM'03), 13–16 October, Boston, USA, vol 1. IEEE, pp 169–173
40. Ganek AG, Corbi TA (2003) The dawning of the autonomic computing era. IBM Syst J 42(1):5–18
41. Horn P (2016) Autonomic computing: IBM's perspective on the state of information technology. 2001. http://people.scs.carleton.ca/~soma/biosec/readings/autonomic_computing.pdf. Accessed 9/12
42. IBM (2016) An architectural blueprint for autonomic computing [white paper]. 2005. http://www-03.ibm.com/autonomic/pdfs/AC%20Blueprint%20White%20Paper%20V7.pdf. Accessed 9/12
43. Lalanda P, McCann JA, Diaconescu A (2013) Autonomic computing: principles, design and implementation. Springer, Berlin
44. Ruokolainen T (2013) A model-driven approach to service ecosystem engineering [Doctoral dissertation]. University of Helsinki, Helsinki, Finland
45. Ruokolainen T, Ruohomaa S, Kutvonen L (2011) Solving service ecosystem governance. In: Proceedings of the 15th IEEE international enterprise distributed object computing conference workshops (EDOCW), 29 August–2 September, Helsinki, Finland. IEEE, pp 18–25
46. ICARE (2016) Innovative Cloud Architecture for Real Entertainment project. 2015. https://itea3.org/project/icare.html. Accessed 9/12
47. Pantsar-Syväniemi S, Ervasti M, Karppinen K, Väätänen A, Oksman V, Kuure E (2015) A situation-aware safety service for children via participatory design. J Ambient Intell Humaniz Comput 6(2):279–293
48. IEEE (1993) IEEE standard for a software quality metrics methodology. IEEE Std 1061-1992
49. ISO/IEC 25010 (2011) Systems and software engineering—systems and software quality requirements and evaluation (SQuaRE)—system and software quality models. ISO/IEC 25010:2011
50. Matinlassi M, Niemelä E (2003) The impact of maintainability on component-based software systems. In: Proceedings of the 29th euromicro conference, 1–6 September, Belek, Turkey. IEEE, pp 25–32
51. Ovaska E, Kuusijarvi J (2014) Piecemeal development of intelligent applications for smart spaces. IEEE Access 2:199–214
52. Ovaska E, Evesti A, Henttonen K, Palviainen M, Aho P (2010) Knowledge based quality-driven architecture design and evaluation. Inf Softw Technol 52(6):577–601
53. Pantsar-Syväniemi S, Purhonen A, Ovaska E, Kuusijärvi J, Evesti A (2012) Situation-based and self-adaptive applications for the smart environment. J Ambient Intell Smart Environ 4(6):491–516
54. Evesti A, Suomalainen J, Ovaska E (2013) Architecture and knowledge-driven self-adaptive security in smart space. Computers 2(1):34–66
55. Immonen A, Palviainen M, Ovaska E (2014) Requirements of an open data based business ecosystem. IEEE Access 2:88–103
56. Jayaratna N (1994) Understanding and evaluating methodologies: NIMSAD, a systematic framework. McGraw-Hill, Inc, New York
57. Kitchenham B, Charters S (2007) Guidelines for performing systematic literature reviews in software engineering. Keele University and University of Durham joint report, Technical report no. EBSE-2007-01
58. Stol K, Ali Babar M (2010) A comparison framework for open source software evaluation methods. In: Ågerfalk P, Boldyreff C, González-Barahona JM, Madey GR, Noll J, editors. Open Source Software: New Horizons: 6th International IFIP WG 2.13 Conference on Open Source Systems. Springer, Berlin, pp 389–394
59. Matinlassi M (2004) Comparison of software product line architecture design methods: COPA, FAST, FORM, KobrA and QADA. In: Proceedings of the 26th international conference on software engineering (ICSE'04), pp 127–136
60. Weyns D, Schmerl B, Grassi V, Malek S, Mirandola R, Prehofer C et al (2013) On patterns for decentralized control in self-adaptive systems. In: de Lemos R, Giese H, Müller HA, Shaw M (eds) Software engineering for self-adaptive systems II: international seminar, Dagstuhl Castle, Germany, 24–29 October, 2010 Revised selected and invited papers, Springer, Berlin, pp 76–107
61. Abeywickrama DB, Hoch N, Zambonelli F (2014) Engineering and implementing software architectural patterns based on feedback loops. Scalable Comput Pract Exp 15(4):291
62. Abowd GD, Dey AK, Brown PJ, Davies N, Smith M, Steggles P (1999) Towards a better understanding of context and context-awareness. In: Proceedings of the 1st international symposium on handheld and ubiquitous computing, Karlsruhe, Germany. Springer-Verlag, London, UK, pp 304–307
63. Hong J, Suh E, Kim S (2009) Context-aware systems: a literature review and classification. Expert Syst Appl 36(4):8509–8522
64. Müller HA, O'Brien L, Klein M, Wood B (2006) Autonomic computing. Carnegie Mellon University, USA. Report No.: CMU/SEI-2006-TN-006
65. Malenfant J, Jacques M, Demers FN (1996) A tutorial on behavioral reflection and its implementation. In: Proceedings of the reflection conference (Reflection'96), April, San Francisco, USA, pp 1–20
66. Bobrow DG, Gabriel RP, White JL (1993) CLOS in context—the CLOS in context—the shape of the design. In: Paepcke A (ed) Object-oriented programming: the CLOS perspective. MIT Press, Cambridge, pp 29–61
67. McKinley PK, Sadjadi SM, Kasten EP, Cheng BHC (2004) Composing adaptive software. Computer 37(7):56–64
68. Tisato F, Savigni A, Cazzola W, Sosio A (2001) Architectural reflection realising software architectures via reflective activities. In: Emmerich W, Tai S (eds) Engineering distributed objects. Springer, Berlin, pp 102–115
69. Henttonen K, Matinlassi M, Niemelä E, Kanstrén T (2007) Integrability and extensibility evaluation from software architectural models—a case study. Open Softw Eng J 1(1):1–20
70. Ruokolainen T, Kutvonen L (2009) Managing interoperability knowledge in open service ecosystems. In: Proceedings of the 13th enterprise distributed object computing conference workshops (EDOCW 2009), 1–4 September, Auckland, New Zealand. IEEE, pp 203–211
71. Liu X, Li Y (2011) Advanced design approaches to emerging software systems: principles, methodology and tools. IGI Global, Hershey
72. Li W, Badr Y, Biennier F (2012) Digital ecosystems: challenges and prospects. In: Proceedings of the international conference on management of emergent digital ecosystems (MEDES'12), 28–31 October, Addis Ababa, Ethiopia. ACM, New York, USA, pp 117–122
73. Quitadamo R, Zambonelli F (2008) Autonomic communication services: a new challenge for software agents. Auton Agents Multi-Agent Syst 17(3):457–475
74. Dill S, Kumar R, Mccurley KS, Rajagopalan S, Sivakumar D, Tomkins A (2002) Self-similarity in the Web. ACM Trans Internet Technol 2(3):205–223
75. Aubonnet T, Simoni N (2014) Self-control cloud services. In: Proceedings of the IEEE 13th international symposium on network computing and applications, 21–23 August, Cambridge, USA. IEEE, pp 282–286

76. Bhakti MAC, Abdullah AB (2010) Design of an autonomic services oriented architecture. In: Proceedings of the international symposium in information technology (ITSim 2010), 15–17 June, Kuala Lumpur, Malaysia. IEEE, pp 805–810
77. Bhakti MAC, Abdullah AB (2009) Towards self-organizing service oriented architecture. In: Proceedings of the innovative technologies in intelligent systems and industrial applications conference (CITISIA 2009), 25–26 July, Kuala Lumpur, Malaysia, pp 458–461
78. Bhakti MAC, Abdullah AB (2009) Nature-inspired self-organizing service oriented architecture: a proposal. In: Proceedings of the international conference on information technology in Asia (CITA 2009),6–9 July, Kuching, Sarawak, Malaysia, pp 276–279
79. Blau B, Kramer J, Conte T, van Dinther C (2009) Service value networks. In: Proceedings of the IEEE conference on commerce and enterprise computing (CEC '09), 20–23 July, Vienna. IEEE, pp 194–201
80. Li Y, Sun K, Qiu J, Chen Y (2005) Self-reconfiguration of service-based systems: a case study for service level agreements and resource optimization. In: Proceedings of the IEEE international conference on web services (ICWS 2005), 11–15 July, Orlando, Florida, USA, vol 1. IEEE, pp 266–273
81. Bencomo N (2013) Requirements for self-adaptation. In: Lämmel R, Saraiva J, Visser J (eds) Generative and transformational techniques in software engineering IV. Springer, Berlin, pp 271–296
82. Bencomo N, Whittle J, Sawyer P, Finkelstein A, Letier E (2010) Requirements reflection: requirements as runtime entities. In: Proceedings of the 32nd ACM/IEEE international conference on software engineering, 2–8 May, Cape Town, South Africa. IEEE, pp 199–202
83. Sawyer P, Bencomo N, Whittle J, Letier E, Finkelstein A (2010) Requirements-aware systems: a research agenda for RE for self-adaptive systems. In: Proceedings of the 18th IEEE international requirements engineering (RE) conference, 27 September–1 October, Sydney, Australia. IEEE, pp 95–103
84. Bencomo N, Bennaceur A, Grace P, Blair G, Issarny V (2013) The role of models@run.time in supporting on-the-fly interoperability. Computing 95(3):167–190
85. Taylor RN, Medvidovic N, Oreizy P (2009) Architectural styles for runtime software adaptation. In: Proceedings of the joint working IEEE/IFIP conference on software architecture 2009 and European conference on software architecture 2009, 14–17 September, Cambridge, UK. IEEE, pp 171–180
86. Oreizy P, Medvidovic N, Taylor RN (2008) Runtime software adaptation: framework, approaches, and styles. In: Proceedings of the 30th international conference on software engineering (ICSE'08), 10–18 May, Leipzig, Germany. ACM, New York, USA, pp 899–910
87. Beckmann BE, Grabowski LM, McKinley PK, Ofria C (2008) Autonomic software development methodology based on Darwinian evolution. In: Proceedings of the international conference on autonomic computing (ICAC '08), 2–6 June, Chicago, USA. IEEE, pp 203–204
88. Goldsby HJ, Cheng BHC, McKinley PK, Knoester DB, Ofria CA (2008) Digital evolution of behavioral models for autonomic systems. In: Proceedings of the international conference on autonomic computing (ICAC '08), 2–6 June, Chicago, USA. IEEE, pp 87–96
89. Goldsby HJ, Cheng BHC (2008) Avida-MDE: a digital evolution approach to generating models of adaptive software behavior. In: Keijzer M (ed) Proceedings of the 10th annual conference on genetic and evolutionary computation, 12–16 July, Atlanta, USA. ACM, New York, USA, pp 1751–1758
90. Goldsby H, Cheng BC (2008) Automatically generating behavioral models of adaptive systems to address uncertainty. In: Czarnecki K, Ober I, Bruel J, Uhl A, Völter M (eds) Model driven engineering languages and systems. Springer, Berlin, pp 568–583
91. McKinley P, Cheng BHC, Ofria C, Knoester D, Beckmann B, Goldsby H (2008) Harnessing digital evolution. Computer 41(1):54–63
92. Cheng BHC, Ramirez A, McKinley PK (2013) Harnessing evolutionary computation to enable dynamically adaptive systems to manage uncertainty. In: Proceedings of the 1st international workshop on combining modelling and search-based software engineering (CMSBSE), 20–23 May, San Francisco, USA. IEEE, pp 1–6
93. McKinley PK, Cheng BHC, Ramirez AJ, Jensen AC (2012) Applying evolutionary computation to mitigate uncertainty in dynamically-adaptive, high-assurance middleware. J Internet Serv Appl 3(1):51–58
94. Cheng BC, Sawyer P, Bencomo N, Whittle J (2009) A goal-based modeling approach to develop requirements of an adaptive system with environmental uncertainty. In: Schürr A, Selic B (eds) Model driven engineering languages and systems. Springer, Berlin, pp 468–483
95. Goldsby HJ, Sawyer P, Bencomo N, Cheng BHC, Hughes D (2008) Goal-based modeling of dynamically adaptive system requirements. In: Proceedings of the 15th annual IEEE international conference and workshop on the engineering of computer based systems (ECBS 2008), 31 March–4 April, Belfast, Northern Ireland. IEEE, pp 36–45
96. Ramirez A, Cheng BC (2011) Automatic derivation of utility functions for monitoring software requirements. In: Whittle J, Clark T, Kühne T (eds) Model driven engineering languages and systems. Springer, Berlin, pp 501–516
97. Ramirez AJ, Jensen AC, Cheng BHC (2012) A taxonomy of uncertainty for dynamically adaptive systems. In: Proceedings of the 7th international symposium on software engineering for adaptive and self-managing systems (SEAMS'12), 4–5 June, Zurich, Switzerland. IEEE, pp 99–108
98. Ofria C, Wilke CO (2004) Avida: a software platform for research in computational evolutionary biology. Artif Life 10(2):191–229
99. Whittle J, Sawyer P, Bencomo N, Cheng BHC, Bruel J (2010) RELAX: a language to address uncertainty in self-adaptive systems requirement. Requir Eng 15(2):177–196
100. Whittle J, Sawyer P, Bencomo N, Cheng BHC, Bruel J (2009) RELAX: incorporating uncertainty into the specification of self-adaptive systems. In: Proceedings of the 17th IEEE international requirements engineering conference (RE'09), 31 August–4 September, Atlanta, USA. IEEE, pp 79–88
101. Van Lamsweerde A, Darimont R, Letier E (1998) Managing conflicts in goal-driven requirements engineering. IEEE Trans Softw Eng 24(11):908–926
102. CONNECT Project (2013) https://www.connect-forever.eu/. Accessed 9/12, 2016
103. Garlan D, Cheng Shang-Wen, Huang An-Cheng, B Schmerl, Steenkiste P (2004) Rainbow: architecture-based self-adaptation with reusable infrastructure. Computer 37(10):46–54
104. Kramer J, Magee J (2007) Self-managed systems: an architectural challenge. In: Proceedings of the future of software engineering (FOSE '07) special track at the ICSE'07 Conference, 23–25 May, Minneapolis, USA. IEEE, pp 259–268
105. Solms F, Gruner S, Edwards C (2011) URDAD as a quality-driven analysis and design process. In: Fujita H, Gavrilova T (eds) New trends in software methodologies, tools and techniques: Proceedings of the 9th international conference on new trends in software methodology tools, and techniques (SoMeT 2011. IOS Press, pp 141–158
106. Hofmeister C, Kruchten P, Nord RL, Obbink H, Ran A, America P (2007) A general model of software architecture design derived from five industrial approaches. J Syst Softw 80(1):106–126

107. Bass L, Clements P, Kazman R (2003) Software architecture in practice, 2nd edn. Addison-Wesley Longman Publishing Co., Inc, Boston
108. Hofmeister C, Nord R, Soni D (2000) Applied software architecture. Addison-Wesley Longman Publishing Co., Inc, Boston, USA
109. Kruchten PB (1995) The 4+1 view model of architecture. IEEE Softw 12(6):42–50
110. America P, Rommes E, Obbink H (2004) Multi-view variation modeling for scenario analysis. In: van der Linden F (ed) Software product-family engineering. Springer, Berlin, pp 44–65
111. Ran A (2000) ARES conceptual framework for software architecture. In: Jazayeri M, Ran A, van der Linden F (eds) Software architecture for product families principles and practice. Addison-Wesley, Boston
112. Evesti A, Niemela E, Henttonen K, Palviainen M (2008) A tool chain for quality-driven software architecting. In: Proceedings of the 12th international software product line conference (SPLC '08), 8–12 September, Limerick, Ireland. IEEE, pp 360–360
113. Bosch J (2000) Design and use of software architectures: adopting and evolving a product-line approach. Addison-Wesley, New York
114. de Bruin H, van Vliet H (2003) Quality-driven software architecture composition. J Syst Softw 66(3):269–284
115. Cheng BC, Eder K, Gogolla M, Grunske L, Litoiu M, Müller H et al (2014) Using models at runtime to address assurance for self-adaptive systems. In: Bencomo N, France R, Cheng BC, Aßmann U (eds) Models@run.time. Springer, Berlin, pp 101–136
116. Heinrich R, Jung R, Schmieders E, Hasselbring W, Metzger A, Pohl K, et al (2015) Run-time architecture models for dynamic adaptation and evolution of cloud applications. Kiel University, Kiel, Germany. Technical report no. 1503