Sustainability Evaluation of Software Architectures: A Systematic Review

Heiko Koziolek

1 Industrial Software Systems, ABB Corporate Research, Ladenburg, Germany

heiko.koziolek@de.abb.com

ABSTRACT

Long-living software systems are sustainable if they can be cost-efficiently maintained and evolved over their entire life-cycle. The quality of software architectures determines sustainability to a large extent. Scenario-based software architecture evaluation methods can support sustainability analysis, but they are still reluctantly used in practice. They are also not integrated with architecture-level metrics when evaluating implemented systems, which limits their capabilities. Existing literature reviews for architecture evaluation focus on scenario-based methods, but do not provide a critical reflection of the applicability of such methods for sustainability evaluation. Our goal is to measure the sustainability of a software architecture both during early design using scenarios and during evolution using scenarios and metrics, which is highly relevant in practice. We thus provide a systematic literature review assessing scenario-based methods for sustainability support and categorize more than 40 architecture-level metrics according to several design principles. Our review identifies a need for further empirical research, for the integration of existing methods, and for the more efficient use of formal architectural models.

1. INTRODUCTION

Software systems with a life span of more than 15 years must be designed and implemented carefully so that they are prepared for maintenance and evolution. During their life-time such systems inevitably undergo many corrective, adaptive, enhance, and preventive changes. This is especially pronounced in the industrial automation domain, where software systems are embedded in complex technical hardware/software environments. Software architectures are a major driver for the sustainability (i.e., cost-efficient longevity) and evolvability [12, 73], because they influence how quickly and correctly a developer is able to understand, analyse, extend, test, and maintain a software system. Evaluating and improving the sustainability of a software architecture is thus a major concern for software architects.

While researchers have proposed many scenario-based evaluation methods [25], it is not well understood how they support improving the sustainability of a system. In practice many architects still mainly rely on experience and prototyping to support their design decisions [10]. For implemented architectures, architecture-level code metrics assessing modularization quality can add valuable information to a sustainability evaluation [18], but an overview and systematic validation of such metrics is missing. Thereby, architecture-level code metrics are still sparsely used in practice.

Existing literature reviews for architecture evaluation methods [26, 9, 38, 11] focus mainly on scenario-based methods to evaluate early software architecture designs and do not analyse their suitability for sustainability evaluation. Other surveys [11, 59, 19] provide more breadth but do not include architecture-level metrics either. Reviews of architecture-level metrics cannot be found in literature, as related studies (e.g., [57]) focus on class-level OO metrics (e.g., McCabe [49], Halstead [33], Chidamber [24]) and neglect metrics for higher-level code structures.

The contribution of this paper is a structured literature review on methods and metrics for evaluating the sustainability of software architectures. Our review carefully analyses existing scenario-based methods for their suitability to evaluate sustainability and additionally provides a survey and analysis of more than 40 architecture-level metrics. An integration of scenario-based and metrics-based methods is useful to provide a continuous, pro-active approach towards evolution problem throughout the entire system life-cycle. Our survey is intended to help practitioners to select a method reflecting their specific requirements, and to help researchers to identify gaps and pointers for future work in the existing body of work. Our review also provides the base for a possible integration of both kinds of methods in a combined and even more valuable approach.

The remainder of this paper is as follows [39]: Section 2 defines the most important terms and motivates the need for a new review. Section 3 states our research questions, list the data sources, inclusion criteria and data collections. Section 4 then presents the results of the review, which shall answer the formally stated research questions. Section 5 discusses the results and provides implications for research and practice. Finally, Section 6 concludes the paper.

2. BACKGROUND

This section first defines the terms 'sustainability' (Section 2.1) and 'software architecture' (Section 2.2) and then discusses related surveys (Section 2.3).
2.1 Sustainability

To define the term 'sustainability' in the context of software architecture, we first introduce the notion of a long-living software system:

**Definition 1.** A software-intensive system is long-living if it must be operated for more than 15 years. Thus, a long-living software system usually needs to operate longer than its technical infrastructure, which for example consists of COTS, middleware, operating systems, and databases. In the industrial automation domain longevity requirements are rooted in the large investments in the controller, network, and field devices of the system. Besides changes of the technical infrastructure, a long-living software system faces new or changing functional and extra-functional customer requirements, changing business strategies, and potentially corrective and preventive maintenance during its life-cycle. The manageable and predictable operation of the system in terms of costs, customer requirements, and technological changes characterizes a sustainable system.

**Definition 2.** A long-living software system is sustainable if it can be cost-efficiently maintained and evolved over its entire life-cycle.

The term 'sustainable' is derived from the Latin word 'sustinere' (tenere: to hold; sus, up) and thus best grasps our intended connotation. Here, the term is not used in the sense of environment-friendliness as in other domains, but instead in the sense of cost-effective longevity and endurance. We do not use the terms 'maintainability' (ISO/IEC 25000), 'modifiability' [14], or 'evolvability' [15, 19], because they arguably include the notions of longevity and cost-effectiveness only to a limited extent.

The opposite of a sustainable software system is a long-living system that cannot be adapted to changing requirements and environments due to unjustifiable costs or even technical infeasibility. The architecture of a sustainable system may evolve during its life-cycle, but the fulfillment of customer requirements within timing, budget, and quality constraints must be assured.

Sustainability at least comprises the attributes maintainability (i.e., analysability, stability, testability, understandability), modifiability, portability, and evolvability. A sustainable software architecture can be achieved through adherence to design principles (e.g., modularity, separation of concerns, conceptual integrity) throughout the entire life-cycle. It requires pro-active planning for the long life-time of the system, which can be achieved by periodic evaluations of evolution scenarios.

2.2 Software Architecture

In the following, we use the software architecture definition of ISO/IEC 42010-2007: "Architecture is the fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution.". Software architecture descriptions consist of multiple views (e.g., functional, concurrency, development) dealing with higher-level software structures (i.e., components, modules, subsystems; not classes or methods).

A scenario is a brief description of a single interaction of a stakeholder with a system [25]. Scenario-based methods provide techniques for eliciting, documenting, and evaluating software architecture related scenarios against the requirements. Scenarios assessing the sustainability of an architecture are often called change, evolution, or exploratory scenarios [14]. A change scenario may impact multiple components. Undesired and costly ripple effects [66] can occur if the change to a component causes changes in dependent components. Thus, loose coupling between system components is a desirable property of a sustainable architecture to avoid such effects.

2.3 Related Reviews

Several authors have provided reviews on scenario-based and metrics-based evaluation methods.

Dobrica and Niemelä [26] classified eight early scenario-based evaluation methods for their activities, goals, addressed quality attributes and other criteria. In a similar survey, Barbar and Gorton [9] added several practical classification criteria, such as maturity stage, process support, and resources required. They also pointed out that several of the formerly reviewed methods were already dormant or merged with other methods. The same authors later analyzed the state-of-practice in software architecture evaluation [8, 10] and found that scenarios are used by 54% of the polled software architects. Kazman et al. [38] criticized the bottom-up classification of scenario-based methods and thus proposed a new top-down classification, which they used to compare the ATAM [25] and ALMA [14] method. None of these surveys focused on sustainability or analysed metrics-based methods.

Barcelos et al. [11] again classified scenario-based methods but also included a small number of measurement approaches without however analysing their suitability for sustainability analysis. In the same manner, Roy and Graham [59] surveyed scenario-based evaluation methods, but provided only limited comparison of the metrics-based methods. Breivold and Crnkovic [19] provided a broad structured literature review on architecture evolvability, which also included experience-based methods, design methods, and knowledge management techniques but provided no sustainability analysis on the scenario-based and metrics-based evaluation methods.

While there are numerous reviews and discussions on class-level metrics reported in literature [24, 57], there is no systematic review on metrics-based software architecture evaluation methods. Sarkar et al. [64] categorized several modularization metrics for higher-level software structures according to their adherence to well-known design principles, such as similarity of purpose, encapsulation, and layering. We reuse these generic principles in our review, but include more metrics. Riaz et al. [57] provided a systematic review of class-level OO metrics [49, 33, 24] and their usefulness for maintenance prediction, but did not take architecture-level metrics into account. Finally, Ducasse et al. [27] surveyed methods and tools for software architecture reconstruction, which can be helpful for the sustainability evaluation of implemented architectures.

3. REVIEW METHOD

3.1 Research Questions

The goal of our study is to review software architecture evaluation methods and metrics for the purpose of assessing their industrial applicability in the context of sustainability analysis from the perspectives of the software architect and
software analyst. From this goal, we derive three research questions:

- **RQ1:** How do scenario-based architecture evaluation methods used in industry support sustainability evaluation?
- **RQ2:** Which architecture-level metrics have been proposed to analyse the sustainability of software architectures?
- **RQ3:** What implications can be derived for the industrial and research communities from the findings?

The motivation for RQ1 is to analyse the usefulness of existing methods for the specific purpose of sustainability evaluation, as most scenario-based methods were defined for the broader scope of a generic architecture evaluation. The aim is to identify gaps in currently used methods and to analyse the potential of combining scenario-based and metrics-based methods. Concrete sustainability criteria analysed will be described in Section 3.3.

As no systematic reviews on architecture-level metrics currently exist, RQ2 first asks for an overview and classification of metrics proposed in literature. It is important to note that we restrict our study exclusively at architecture-level metrics dealing with subsystems, components, and interfaces. Class-level metrics have the potential to analyse and improve the sustainability of a software system as well, but are out of scope of our study.

Finally, RQ3 aims at a discussion of the findings, potential for future research, and possible improvements and new directions for sustainability evaluation.

### 3.2 Data sources and search strategy

**Search Process:** We searched for software architecture evaluation methods in several books [40, 16, 25, 12, 47, 60, 73] and the journal and conference proceedings listed in Tab. 1. We used the following search engines for our reviews: ACM Digital Library, Google Scholar, IEEE Xplore, Elsevier ScienceDirect, and SpringerLink. We searched for term "software architecture" together with the following keywords: bad smells, evolvability, evolution, maintainability, maintenance, qualitative evaluation quantitative evaluation, scenario-based evaluation, metrics, modifiability, modularization, and sustainability.

<table>
<thead>
<tr>
<th>Journals</th>
<th>Conferences &amp; Workshops</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACM Trans. on Software Eng. and Meth. (TOSEM)</td>
<td>ACM Symp. on the Foundations of Softw. Eng. (FSE)</td>
</tr>
<tr>
<td>Communications of the ACM (CACM)</td>
<td>ACM/IEEE Int. Conf. on Softw. Eng. (ICSE)</td>
</tr>
<tr>
<td>Elsevier Information and Software Technology (IST)</td>
<td>IEEE Int. Symp. on Software Metrics (METRICS)</td>
</tr>
<tr>
<td>Elsevier ScienceDirect, and SpringerLink</td>
<td>Elsevier Science of Computer Programming (SCP)</td>
</tr>
<tr>
<td>IEEE Computer</td>
<td>Elsevier Journal of Information Systems Technology (JIST)</td>
</tr>
<tr>
<td>Europ. soften and Conference (ESEC)</td>
<td>Elsevier Journal of Softw. Maint. and Evolution (JSME)</td>
</tr>
<tr>
<td>Wiley Journal of Systems Engineering (JSE)</td>
<td>Europ. Conf. on Software Architecture (ECSA)</td>
</tr>
<tr>
<td>Wiley Journal of Softw. Maint. and Evolution (JSME)</td>
<td>Europ. Conf. on Software Architecture (ECSA)</td>
</tr>
<tr>
<td>Int. Conf. on the Quality of Softw. Architecture (QoSA)</td>
<td>Int. Conf. on the Quality of Softw. Architecture (QoSA)</td>
</tr>
<tr>
<td>ERCIM Workshop on Software Evolution (EVOL)</td>
<td>IEEE Computer</td>
</tr>
<tr>
<td>Springer Empirical Software Engineering (EMSE)</td>
<td>IEEE Transactions on Software Engineering (TSE)</td>
</tr>
<tr>
<td>Int. Worksh. on Principles of Softw. Evolution (IWPSE)</td>
<td>IEEE Int. Symp. on Software Metrics (METRICS)</td>
</tr>
<tr>
<td>Communications of the ACM (CACM)</td>
<td>Elsevier Information and Software Technology (IST)</td>
</tr>
<tr>
<td>ACM/IEEE Int. Conf. on Softw. Eng. (ICSE)</td>
<td>Communications of the ACM (CACM)</td>
</tr>
</tbody>
</table>

**Inclusion and Exclusion Criteria:** More than 20 scenario-based architecture evaluation methods have been proposed and categorized in various surveys [26, 9, 38, 10]. We exclude methods for which no industrial application has been reported or which target other quality attributes exclusively. We also exclude methods that are considered dormant [9] (e.g., SAAM, SBAR, SAMMCS), or for which no applications (e.g., case studies) or extensions have been reported for several years.

For metrics-based approaches, we only consider metrics concerning high-level software structures and exclude classical complexity metrics (e.g., McCabe [49], Halstead [33]) or class-level object-oriented metrics (e.g., Chidamber [24], Harrison [34]). We also exclude process metrics or other development metrics as our focus is on software architecture.

**Quality Assessment:** For quality assessment we include as primary studies only publications from the books cited before and the journals and conferences listed in Tab. 1. For both scenario-based methods and architectural metrics, we checked whether they have been applied in an industrial setting.

### 3.3 Data Collection

The following describes which data we extracted from the primary studies to evaluate scenario-based methods and metrics for their suitability in the context of sustainability evaluation.

**Scenario-based methods:** For the scenario-based methods, we can reuse most of the criteria from the former surveys [26, 9, 38] in our context. Each evaluation method serves a specific goal and requires a certain architectural description. Furthermore, tool support, process support, knowledge bases, and the form of validation are generic requirements. Some additional, sustainability-specific requirements can be stated as follows:

- **Support for change scenario specification:** A useful method should provide templates for specifying change scenarios and guidelines for finding and eliciting such scenarios. A mature method could provide change scenario patterns from former applications, which may speed up scenario definition. To effectively support change scenario evaluation, a method should offer repeatable techniques for determining the consistency and completeness of change scenarios.

- **Support for analyzing ripple effects:** An ideal method should provide (semi-)automatic support for analyzing ripple effects. The possibility for such analyses is largely determined by the level of formalization of the architecture. Interface descriptions and dependencies between components are required. If possible, the analyzed architectural documentation should provide service effect specifications [56] for individual software components, which determine the control and data flow through components on an architectural level.

- **Support for analyzing variation and extension points:** Variability must be introduced with care as too much variability can compromise the evolution of a system, while too few variability makes a system inflexible [71]. Thus, a sustainability evaluation method should provide means to identify, analyze, and constrain variation and extension points.

- **Support for improving the architecture (heuristics):** A complete analysis method provides recommendations and techniques on how to improve the architecture. This is especially helpful for less experienced architects. An integration of an evaluation method with documented best practices in styles, patterns, and tactics is desirable [12, 38].

- **Support for trade-off analysis:** Optimizing a software architecture for sustainability inevitably leads to trade-offs with other quality attributes. For example, the performance of a sustainable system might be compromised,
if a system uses lots of indirection to ensure low coupling. Thus, a sustainability evaluation method should provide techniques to allow the analysis of trade-offs between sustainability and other quality attributes.

- **Support for legacy systems**: While many scenario-based methods target early design stages of a software system, the use case of assessing legacy systems is much more common in practice, especially for long-living software systems. Thus the interoperability of a method with architecture reconstruction tools is desirable [27].

- **Support for existing artifacts and tools**: To facilitate a broader adoption of any architecture analysis method, an integration with existing artifacts and tool chains is essential. For example, if a method provides analytical tools, they should be able to process existing design document (e.g., in UML). As architecture evaluation relies on the functional and extra-functional requirements of a system, interfacing with respective requirements management tools is desirable.

- **Return on investment (ROI)**: Empirical evidence of a significant cost/benefit ratio is essential for any method to achieve broad adoption in industry. To better characterize any method, it should be clear how many resources are required and what kinds of results can be expected.

We also collected data about supplemental approaches and tools for the scenario-based methods, as well as metrics for ranking evolution scenarios.

**Architectural Metrics**: Architectural metrics often measure the modularization quality of a system under the assumption that a good modularization leads to better understandability, analysability, and maintainability and thus sustainability. We extracted the name of each metric, its abbreviation, an intuitive description, and the required inputs from the primary studies. The latter helps to analyse whether the metric can be determined automatically using tools. Additionally, we categorized the metrics according to common modularization design principles [64]:

- **Similarity of purpose**: This principle states that functions and data structures serving similar purposes or aiming at common goals should be grouped in a single module, package, or subsystem. It implies high cohesion within such a module.

- **Encapsulation**: Information hiding improves understandability and analysability. Modules should have an explicitly defined, restricted API through which all inter-module call traffic should be routed.

- **Independent compilability, extendibility, testability**: These principles allow modules to grow in parallel and to be tested independently. The possibility for an independent evolution of modules is important for cost-effective development.

- **Acyclic dependencies**: As cyclic dependencies negate many of the benefits of modularization, they should be avoided where possible. In layered architectures, control flow should be directed only from upper layers to lower layers but not vice versa.

- **Size**: While there are no universally agreed module sizes, it is plausible that modules should neither be too large nor too small. A uniform size distribution is desirable to improve the maintainability of a system. This however can hardly be planned when implementing systems from scratch, but is rather a desirable property when clustering legacy systems.

### 3.4 Data Synthesis

Data synthesis involves collating and summarizing the results of the included primary studies. We chose a descriptive synthesis and display the information extracted from the primary studies in tables.

### 4. RESULTS

This section reports on the results for research questions RQ1 (Section 4.1) and RQ2 (Section 4.2). RQ3 will be tackled in Section 5.2.

#### 4.1 Scenario-based Methods (RQ1)

Research question RQ1 asks for the suitability of current scenario-based methods for sustainability evaluation. We found more than 20 scenario-based evaluation methods in literature. Our exclusion criteria limit the scope to ATAM [25] and ALMA [14], because these are the only active methods that have been applied in a number of industrial case studies in different domains [9]. For example, SAAM [25] is no longer supported by its creator as it has been superseded by ATAM. Additionally, its various derivates are dormant or have been merged into other methods [9]. PASA and SALUTA have been applied in industry but target performance and usability exclusively.

##### 4.1.1 ATAM

The goals of ATAM [25] are to identify trade-offs between different quality attributes for a system and reveal sensitivity points in an architecture. ATAM involves presenting and discussing the architectural design in a 2 day workshop attended by the system's major stakeholders. The participants define scenarios (i.e., regular use-cases, growth, and exploratory scenarios) to evaluate the architecture against and discuss the technical constraints of the system. Concerning tool support, there is a web-based tool available [44], which is however not regularly used. There are no specific modeling tools targeted by ATAM, but the method requires a logical and module view of the architecture.

ATAM supports sustainability evaluation as follows: the definition of change scenarios is assisted with the quality attribute scenario template and procedures for a quality attribute workshop (QAW) to determine them. The method description explicitly mentions that growth and exploratory scenarios, which can represent change scenarios, should be described. Stakeholders have to determine ripple effects manually, as the granularity of architectural description used in ATAM does not allow a formal analysis. There is no special support for analysing variation and extension points. ATAM offers some sustainability improvement recommendations in the form of change-oriented architectural styles and so-called modifiability tactics. For trade-off analysis, the stakeholders rank different quality attributes in a so-called utility tree. To support analysing legacy systems, the SEI proposed the DALI workbench, which however has not been applied on systems greater than 50 KLOC and seems to be dormant since 2001. There is no special support for existing tool chains. The industrial maturity has been proven in more than 20 industrial case studies [25, 13, 17]. The effort for applying ATAM is estimated between 30-70 person days [25], but it was never attempted to quantify its
benefits.

**Evaluation:** ATAM was not specifically designed for sustainability evaluation, but more generically for assessing risks and trade-offs between quality attributes. Nevertheless, it offers many helpful techniques in the context of sustainability evaluation (e.g., utility trees trading off sustainability with other attributes, quality attribute scenarios to document expected changes, abstract modifiability tactics). Other authors have noted that ATAM does not allow for an in-depth analysis of scenarios [38], because of the strict timing constraints of the workshop. However, identified sustainability risks could be further evaluated with other methods after ATAM. In general ATAM is not perceived as a light-weight evaluation method, because it requires to conduct expensive workshops with all major stakeholders.

### 4.1.2 ALMA

The goals of ALMA [14] are to predict maintenance effort, to assess risks, or to compare candidate architecture w.r.t. to modifiability. There is no specific workshop described with the method but the required information is collected by an analyst through stakeholder interviews. Based on a software architecture description the analyst elicits and evaluates change scenarios for the system together with the stakeholders (e.g., architects, developers, customers). Maintenance effort is predicted by estimating the expected extent of code changes per evolution scenario in lines of code. Risk analysis involves weighting the impact of evolution scenarios, and candidate architecture comparison is based on calculating an overall score for implementing the evolution scenarios for all candidates.

ALMA supports sustainability evaluation as follows: a top-down and bottom-up technique for finding change scenarios is sketched and the authors suggest partitioning the scenarios into equivalence classes. There is no documentation template provided. Determining ripple effects is explicitly foreseen but relies on the experience of the involved architects and developers, which can be misleading [42]. There is no support for analysing variation and extension points and there is no guidance on how to improve the architecture after evaluation. Trade-off analysis is not in the scope of ALMA, which exclusively focuses on modifiability. The architecture of a legacy system has to be extracted manually or using other methods into the required views and there is no tool support. The authors have applied ALMA in seven industrial case studies [14] until 2004, but no third party applications are known and effort estimations for a ROI calculation are missing.

**Evaluation:** ALMA was specifically designed for modifiability evaluation, which is closely related to sustainability evaluation. It offers some helpful techniques for change scenario elicitation but still relies heavily on the experience of the involved stakeholders as there is no guidance on how to improve the architecture. Some experience reports with ALMA offer interesting insights to sustainability evaluation [41, 42]: During change scenario elicitation architects are biased towards the scenarios they already had in mind when designing the architecture. Determining the components affected by a change scenario is often straightforward, but determining ripple effects is not. Often, stakeholders miss important change scenarios during architectural evaluation. Furthermore, the architecture and the change scenarios may be based on an incorrect initial requirements specification, which then invalidates them. In conclusion, support for architectural improvements, trade-off analysis, and extension points would be desirable for ALMA besides tool support.

### 4.1.3 Supplemental methods and applications

The following supplemental approaches are not among the primary studies of our systematic review for various reasons, but could be combined with ATAM and ALMA for improved sustainability analysis and are therefore listed here for completeness:

- Kazman et al. [37] combined ATAM and CBAM, and added a more in-depth model-based analysis for selected risks to form the "Analytic Principles and Tools for the Improvement of Architectures" (APTI) method. They also used the method in a case study to analyse the variation points of an architecture.
- Shen and Madhavji [68] describe the "Evolutionary Scenario Development Method" (ESDM), which includes an elaborate template for change scenario specification and shall be used in combination with ATAM.
- Olumofin and Misic [52] introduce the "Holistic Product Line Architecture Assessment" (HooPLA) method extending ATAM for SPL evaluation and provide a qualitative analytical treatment of variation points using scenarios.
- Breivold and Crnkovic [20] propose the "Architecture Evolvability Analysis" (AREA) method, which evaluates change stimuli against fine-grained sustainability attributes (e.g., analysability, testability) and defines respective refactorings and test cases. The method was prototypically applied for assessing a single change scenario of an industrial system.

In addition to the proposed extensions to scenario-based evaluation methods, there are several case study reports in literature [48, 32, 17].

### 4.1.4 Metrics for Scenario Ranking

Several authors have also proposed metrics to rank evaluation scenarios and use them for maintenance effort predictions:

- Avritzer and Weyuker [7] created a list of potential project issues based on architectural reviews and ranked them to get a simple project risk metric. These issues are however only loosely related to the structure of the system.
- Paulish and Bass [54] decompose evolution scenarios into smaller tasks and asked developers for the task efforts.
- Liu and Wang [43] propose two metrics (Impact On the Software Architecture (IOSA), Adaptability Degree of Software Architecture (ADSA)) based on probabilities for evolution scenarios and their impact in terms of affected lines of code or function points. Tarvainen [72] applies these metrics in a case study.
- Stammel and Ruessner [69] propose the KAMP tool, which lets architects decompose evolution scenarios into smaller change actions based on a formal architectural model and uses summarized implementation effort estimations for scenario ranking.
- Anwar et al. [6] compute the "maintenance effort" per evolution scenario based on probability weights and estimated LOC using COCOMO II.
In conclusion scenario ranking methods combine classical cost estimation techniques (e.g., COCOMO, Function Points) with evolution scenarios to enable early maintenance effort predictions.

### 4.1.5 Conclusions RQ1

Table 2 summarizes the sustainability support of our primary studies ATAM and ALMA. While ATAM is more refined and offers more features, ALMA is more specifically designed for sustainability evaluation. In practice, a combination of ATAM, ALMA, and the supplemental methods and scenario-ranking approaches is advisable. It is common that the methods are not used exactly as they are documented [10], but that different parts of them are recombined as needed. Nevertheless, existing scenario-based methods do not provide systematic analysis of ripple effects, integration with reverse engineering tools, or knowledge management support and could benefit from more formal models.

<table>
<thead>
<tr>
<th>Name</th>
<th>General criteria</th>
<th>ATAM</th>
<th>ALMA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architectural Description</td>
<td>Process, data-flow, uses, physical &amp; module view</td>
<td>Any</td>
<td>n/a</td>
</tr>
<tr>
<td>Process support</td>
<td>Comprehensive</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>Tool support</td>
<td>Manual, based on experience</td>
<td>Manual, based on experience</td>
<td>n/a</td>
</tr>
<tr>
<td>Knowledge repository</td>
<td>Recommended</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>Validation</td>
<td>&gt;20 industrial case studies</td>
<td>Industrial case studies (none since 2004)</td>
<td>n/a</td>
</tr>
<tr>
<td>Utilities</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Goals</td>
<td>Sensitivity &amp; tradeoff-analysis</td>
<td>Change impact analysis, predicting maintenance effort</td>
<td>Change impact analysis, predicting maintenance effort</td>
</tr>
<tr>
<td>Sustainability criteria</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Change scenario spec.</td>
<td>Quality attribute scenario template</td>
<td>Top-down, bottom-up method, no template</td>
<td>n/a</td>
</tr>
<tr>
<td>Ripple effect analysis</td>
<td>Manual, based on experience</td>
<td>Manual, based on experience</td>
<td>n/a</td>
</tr>
<tr>
<td>Variation/extension point analysis</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>Architecture improve.</td>
<td>Tactics, Styles</td>
<td>Manual, based on experience</td>
<td>n/a</td>
</tr>
<tr>
<td>Trade-off analysis</td>
<td>Based on utility tree with preferences</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>Legacy systems</td>
<td>No support</td>
<td>No support</td>
<td>n/a</td>
</tr>
<tr>
<td>Existing artifacts and tools</td>
<td>No explicit support</td>
<td>No explicit support</td>
<td>n/a</td>
</tr>
<tr>
<td>Return on Investment</td>
<td>Costs: 30-70 person days</td>
<td>Costs: unknown</td>
<td>Benefits: not quantified</td>
</tr>
</tbody>
</table>

Table 2: Comparison of scenario-based evaluation methods

### 4.2 Metrics-based Methods (RQ2)

Concerning RQ2 this section summarizes architecture-level metrics suites for software architectures. We briefly describe each approach (Section 4.2.1) and provide a categorization in Tab. 3. Furthermore, we list some recent approaches for defining and detecting architecture bad smells, which are closely related to architecture-level metrics (Section 4.2.2).

#### 4.2.1 Architecture-level Metrics Suites

Most work in the area of architecture-level metrics derives from the module concept described by Parnas [53] and the notions of coupling and cohesion [70, 74]. Software complexity metrics [49, 33] as well as class-level object-oriented metrics [24, 57, 1] are out of scope for our study. Our systematic review extracted the following metrics suites and approaches from literature:

**M1** Briand et al. [22, 21] provide a generic formalization for metrical notions independent of any programming paradigm, which includes coupling and cohesion besides complexity-based and class-based metrics. The authors also formally define coupling and cohesion between modules.

**M2** Lakos [40] defines a metric called Cumulative Component Dependency (CCD), which is the sum of required dependencies by a component within a subsystem. Derived metrics are the average component dependency (ACD) and the normalized CCD (NCCD). They can be determined by tools such as SonarJ or STAN.

**M3** Mancoridis et al. [45, 51] introduce a clustering tool called BUNCH, which tries to optimize the proposed modularization quality metric. This metric is based on a partitioned module dependency graph and computed by the difference of the average inter- and intra-connectivity of the partitions.

**M4** Allen and Khoshgoftaar [3, 4, 2] propose an information-theory based approach to define coupling and cohesion metrics. Opposed to the former count-based measures, their definitions are based on the entropy in a module interconnection graph, which accounts for patterns in the relationships. They found that the information-theory based metrics were able to make finer distinctions than the count-based metrics.

**M5** Martin [47] defines metrics for software packages, i.e., groups of related classes (e.g., java packages, C++-projects, low-level modules). These include package affection coupling, efferent coupling, abstractness, instability, distance from main sequence, and package dependency cycles. Several tools support measuring these metrics (e.g., JDepend, CppDepend, STAN).

**M6** Sant’anna et al. [62] propose a set of 11 concern-driven metrics to measure the modularity of a software system. An architectural concern is defined as a partition of system components with a common goal (e.g., GUI, persistence, distribution). Examples for the metrics are concern diffusion (i.e., counts the number of components or interfaces for a given concern), coupling between concerns (i.e., counts for a concern how components relate itself and other concerns), or interface complexity. The authors compare the modularization of aspect-oriented and non aspect-oriented systems using their metrics in three case studies.

**M7** Sarkar et al. [64] create a set of 12 API-based and information-theoretic metrics for measuring modularization quality. The metrics rely on the definition of APIs between modules, module size thresholds, and concept term maps and explicitly exclude any object-oriented features. Each metric is defined between 0 and 1 where higher values are better. Some metrics are derived from the works of Lakos [40] and Martin [47]. The metrics were applied on a number of open source systems (e.g., Apache, MySQL, Mozilla) as well as a 12 MLOC commercial system [65], but the authors do not provide their tools publicly.

**M8** Sarkar et al. [63] extend their former work on module metrics with 9 additional metrics concerning object-oriented relationships (e.g., inheritance, association) between higher-level modules in large OO-systems. These metrics for example measure the extent of the fragile base-class problem or inheritance relations between higher-level modules. They were measured for eight open source systems between 30 KLOC and 2.5 MLOC, where a human
modularization achieved significantly higher values than a randomized modularization based on assigning classes arbitrarily to modules.

M9 Sangwan et al. [61] introduce the complexity measurement framework Structure 101, which uses a metric called excessive structural complexity (XS), which is computed as the product of the degree of cyclical dependencies violations (metric 'tangled') and a multi-level complexity metric (‘fat’), which can also be determined on the package or module level.

M10 Anan et al. [5] propose an approach that is similar to Allen and Khoshgoftaar [4] and use information-theory to measure the coupling between modules. The authors compute the entropy of an architectural slicing (i.e., a module layer) and condense the values to a metric called 'architecture maintainability effort'. The metrics were applied for a number of artificial module dependency graphs.

M11 Sethi et al. [67] base their metrics not on source code, but instead on an augmented constraint network and design structure matrix derived from a higher-level UML component diagram, which also captures architectural concerns and design rules. They define metrics such as decision volatility (i.e., impact of environmental conditions on a design decision) and concern overlap. They evaluate the metrics on eight object-oriented and aspect-oriented redesign scenarios and find that the metrics have different effects, but also lead to higher design volatility.

Besides mapping the metrics to design principles as in Tab. 3, they could also be classified according to their required inputs. Some metrics can be determined by simply analysing source code, while others require additional inputs, e.g., concept maps, design decisions, or module size thresholds. Some metrics are similar or overlapping (e.g., M6 is derived from Martin’s Instability, NTDM is an adaption of CCD).

4.2.2 Architecture Bad Smells
Several authors have proposed to transfer the idea of "bad smells" in code [29] to higher-level software structures, where they are called "architecture smells" [58]. It is also conceivable to deduce a condensed metric for sustainability by weighting a number of found architectural smells. They are typically used in the evaluation of implemented architectures [18].

Kazman and Burth [36] propose the so-called "Interactive Architecture Pattern Recognition" (IAPR) approach for detecting architecture smells. The approach is supported by a tool for diagnosis and exploration. In a case study, the authors demonstrate the detection of cyclical dependencies, layering violations, communication constraint violations, classes with high fan-out or high fan-in. The approach can be considered a precursor to recent architecture smell discovery techniques, but was not followed-up by the authors.

Roock and Lippert [58] provide an extensive catalog of architectural smells. They include smells in dependency graphs (e.g., static cycles), inheritance hierarchies (e.g., parallel inheritance), packages (e.g., too large), subsystems (e.g., overgeneralization), and layers (e.g., upward references).
also list tools for locating architecture smells and describe best practices on how to perform large-scale refactorings.

Garcia et al. [30] describe four architectural smells (i.e., connector envy, scattered parasitic functionality, ambiguous interfaces, and extraneous adjacent connectors) and discuss their quality impacts and trade-offs. They regard tool support for smell detection as future work.

4.2.3 Conclusions RQ2

Our review has provided a list of more than 40 architecture-level metrics, which could assist sustainability evaluation of implemented architectures. A mix of these metrics as well as the architecture bad smells should be identified for a given project, as no single metric is able to characterize the overall sustainability of the implementation. The metrics can be combined with class-level metrics [57], process metrics, and other development metrics [35] and can be monitored during system evolution (e.g., as done in the ISIS approach [55]). Normalizing the metrics between 0 and 1 [64] improves understandability. Most metrics are currently based on plausibility and have not been systematically validated empirically. Their concrete value towards sustainability improvement is thus still unknown.

5. DISCUSSIONS

5.1 Principle findings

Our survey analysed the capabilities of scenarios-based methods and architecture-level metrics for evaluating the sustainability of software architectures. While there are many scenario-based methods proposed, only ATAM and ALMA have so far been used repeatedly in industrial case studies in different domains. Since ALMA (published in 2004), no new method was validated extensively. ALMA too lacks newer experience reports. Practitioners still mostly rely on experience and prototypes instead of following a scenario-based method step-by-step [10]. Nevertheless applying the methods likely results in more sustainable software architectures. For an effective sustainability architecture review, the best techniques from ATAM, ALMA, and supplemental methods should be combined.

Besides the plethora of class-level OO metrics, there is now also a growing number of metrics on the architectural level. These mostly focus on the modularization quality of an architecture and require an implementation for measurement. A comprehensive set of metrics was proposed by Sarkar et al. [64, 63], but tool support is still missing. There is potential to define metrics based on architecture documentation (e.g., UML component diagrams [67]). Desired values for the metrics as well as systematic empirical evaluations are mostly missing. Nevertheless, reviewers use metrics informally during the evaluation of implemented architectures [18].

No end-to-end method for sustainability evaluation from requirements to maintenance is available.

5.2 Implications for research/practice (RQ3)

Based on our review, we identified the following implications for research and practice:

- More empirical research needed: both scenario-based methods and architecture-level metrics should be evaluated more thoroughly and repeatedly in empirical studies especially regarding their return on investment. This should extend the current amount of controlled experiments, field studies, and surveys [28]. Besides a sound scientific assessment of benefits and drawbacks, empirical studies can also be helpful to overcome the limited application of the analysed approaches in practice [10], as they might be used to advocate the methods better. A return on investment quantified through empirical studies could greatly speed up their adoption in practice. Another open question is how much effort should be invested upfront in the definition of evolution scenarios since the evolution of a system is often unpredictable [42] and thus might lead to wasted efforts.

- Integration of methods: while the combination of different scenario-based methods or the combination of multiple metrics can be helpful, there is also potential for combining the qualitative and quantitative approaches formerly presented. As quantitative methods are usually only applicable on implemented architectures, combined methods should target legacy systems. Initial attempts at combined methods have been made by Briand and Wüst [23] (SAMM + coupling measures) and Kazman et al. [37] (ATAM + rate monotonic analysis and variability analysis). Methods from ALMA for change scenario elicitation could be combined with ATAM techniques for trade-off analysis. A metrics assessment report evaluation step could be integrated into the scenario-based approaches. Additionally, architecture recovery tools [27] or automated improvement tools [46] could be integrated with the methods.

- Effective use of formal models: scenario-based evaluations are usually based on early, informal architecture models. Thus, tool-supported analysis (e.g., for ripple effects, undesired dependencies, other non-functional properties) is hardly feasible at this stage. As more formal models require a higher specification effort, methods and tools to rapidly construct and analyse such models would be desirable. Modeling tools could provide aids to ask the architects precisely for the required information for a certain analysis (e.g., more refined interface specifications or dependencies) given an early architectural model. Modelling notations and tools should support iterative refinements based on the successively increasing available information during architectural design.

- Codify experiences: the outcome of most scenario-based evaluation methods is still largely determined by the experience of the participants. Patterns and tactics are a way to encode design knowledge and make good design practice available even to inexperienced architects. Knowledge bases should be created to capture and reuse the experiences from former sustainability evaluations. Also architecture-level metrics could be accompanied with a catalogue or guidelines for architecture-level refactorings aiming at improving specific metric values.

- Explore other approaches: Using scenarios during early design stages and metrics during implementation and maintenance seem to be the most popular methods to improve software architecture evolution. Other approaches (e.g., based on simulation or other techniques) are not explored extensively. Garlan et al. [31] propose a method and tool called AEvol for assisting software architects in designing and analysing architectural evolution paths. Mens et al. [50] propose an approach and tool called Evolve to link architectural descriptions to im-
plementation artifacts and making the evolution intrinsic to the architectural description. These approaches still need refinement and maturation but provide pointers for future directions of developing sustainable software architectures.

5.3 Strength and weaknesses of this review

Our review is the first review on architecture evaluation including both scenario-based and measurement-based approaches. It thus provides a more holistic perspective. Compared to former reviews, we use a more restricted scope (i.e., sustainability evaluation) and include more recent studies. Our survey includes architecture-level metrics, which were formally not reviewed systematically.

However, some limitations are inherent to our review. We use very selective inclusion criteria and potentially exclude promising studies, which have not been evaluated thoroughly. Our survey might be biased towards more popular evaluation methods and metrics also since we focus on a restricted set of renowned books, journals, and conference proceedings. We have not applied the methods and metrics ourselves and can thus not be sure of their applicability.

6. CONCLUSIONS

This paper provided a systematic review on scenario-based evaluation methods and architecture-level metrics for the sustainability of software architectures. We analysed the suitability of existing methods for sustainability analysis and assembled a list of more than 40 architecture-level metrics. We discussed implications for practice and research.

Practitioners can use our review to tailor their own sustainability evaluation method based on the referenced methods and tools. Researchers can identify gaps in the body of work and create systematic sustainability evaluation methods.

Our review identifies a need for more empirical studies on architecture evaluation. Scenario-based methods should better validate their potential return on investment and metrics require a validation of their relevance for sustainability. Additionally the existing methods and metrics should be combined in integrated approaches that offer even more benefits. Using more formal architecture models could provide more automated analyses during architecture analyses.

7. REFERENCES


