Software Migration and Architecture Evolution with Industrial Platforms: A Multi-Case Study

Konstantinos Plakidas[1], Daniel Schall[2], and Uwe Zdun[1]

[1] Software Architecture Research Group
University of Vienna, Austria
firstname.lastname@univie.ac.at
[2] Siemens Corporate Technology, Vienna, Austria
firstname.lastname@siemens.com

Abstract. The software industry increasingly needs to consider architecture evolution in the context of industrial ecosystem platforms. These environments feature a large number third-party offerings with a high variety and complexity of design and technology options. The software architects working on platform migration and in-platform evolution scenarios in such environments require support to find and utilize optimal offerings, ensure design compatibility with various technical and non-technical constraints, and optimize architectures. Based on a multi-case study of three industrial cases, we have derived an architecture knowledge model that provides a basis for supporting software architects in platform migration and in-platform evolution scenarios.

1 Introduction

A common scenario in modern software industry practice is the migration and architectural evolution of legacy systems, usually developed inside a single organization, to cloud platforms. This evolution process typically aims to utilize the various offerings of these platforms, as well as incorporating third-party products from related software ecosystems or integrating devices as part of the Internet of Things (IoT). Unlike the familiar contours of in-house development, architects find themselves confronted with a new production environment that offers a large number and variety of offerings and deployment options, and that is highly dynamic. As a result, industrial platform migration and in-platform evolution is an increasingly challenging undertaking [9].

For the software architects involved in this process, this presents three major challenges: the discovery of a new target environment's parameters (e.g., available technologies, offerings, and constraints); the restructuring and optimization of an application for the target environment; and the subsequent management of its structure across a lifecycle that can span several platforms and deployment configurations. In each case, the architect's decisions are heavily dependent on context—best practices for a specific application domain, available products and technologies, relevant regulations, desired qualities, and so on. All these constantly change over time and across different use cases and platforms.

Ideally, this context should be captured as knowledge, kept up-to-date, and made available for use by the architects during the decision process. While there are several approaches in the literature on capturing knowledge about architecture evolution decisions [2], they have not yet found widespread adoption in practice, and the community is actively researching on how to make them more lightweight and easier to use [8]. In practice, the evolution process is labor-intensive, error-prone and time-consuming, especially in an enterprise-level application that involves multiple teams, constraints, and features developed over longer periods of time.

This study aims to contribute towards filling this gap by providing a light-weight and reusable approach that enables architects to perform an exploratory analysis of their options in a structured manner. The focus lies not on detailed implementation, but on providing "just enough architecture" [3] for the broad outlines and main design decisions of a project—e.g., programming languages, technologies, architectures—that once taken are "costly to change" [8]. Based on three industrial system cases demanding significant architectural evolution, we performed a multi-case study to derive elements and relationships of an architecture knowledge model, which was then used to support software architects in platform migration and in-platform evolution scenarios.

The paper is structured as follows: Sec. 2 discusses related work. Next, Sec. 3 introduces our research method and the three industrial cases. Sec. 4 describes our approach, and Sec. 5 discusses the results and concludes the paper.

2 Related Work

Softrware architecture is expected to support the evolution of software systems to keep pace with the shifts in their technical and business environment [13,2]. Accordingly, correct understanding and representation of the architecture are fundamental for a systematic evolution process [13]. Research has produced a large number of patterns and architectural styles, which serve to address recurring design problems [7]; this has been extended to cover new paradigms such as cloud-based architectures [4,10] or microservices [16,6]. Nevertheless, research in the field is still far off from the ideal of "capturing architectural knowledge in a single [...] handbook, which codifies knowledge to make it widely available" [8], as the various approaches are isolated and fragmented. While valuable on their own, in practice many of the approaches require much input from the stakeholders and result in a "collection of documents." This is an overhead that people usually prove unwilling to invest in, especially if the value of the outcome is unclear. Our approach is intended to be a more light-weight alternative with regard to discovery and management of offerings which limits the decisions taken during the architecting process to only the relevant set of constraints. The decision space is limited to manageable proportions by providing only compatible options and their driving forces and consequences.

3 Study Design

The work reported in this paper follows a multi-case study research method. We followed the available guidelines for such case studies in industry [17] for the design of our study. The *research questions* we defined for this study are:

- **RQ1**: to investigate the minimal set of elements and relationships required by software architects to sufficiently represent and specify a software system at a relatively high abstraction level suitable for brownfield development
- **RQ2**: to investigate the minimal set of elements and relationships required to represent the contexts of in-platform evolution and platform migration
- **RQ3**: to investigate how the models resulting from RQ1 and RQ2 can be used to support architectural decisions

The overall objective was to limit the effort in modelling (compared to existing methods such as [2]). The model must be detailed enough to represent the case study context, while generic enough to avoid overfitting. The main facilities offered by the model would be the management of knowledge, by creating a central knowledge model that can be used to represent any software product, and more importantly the exploitation of that knowledge, by exploiting the links between elements in a knowledge repository to select subsets based on specific criteria. This would allow the architect to browse for "suitable" offerings (i.e., having a desired set of functional and non-functional attributes) for each step of the evolution process; if such were not found, or were not available, the model should be able to provide suggestions for adaptation of existing offerings, or the development of new ones, by exploiting the knowledge base.

In our prior work, we performed a set of *comprehensive studies of the literature* on software architecture decision making [12], quality attributes in such decisions [12], and in software ecosystems [14]. In addition, for this work we exploratively studied the practitioner literature on migration practices and patterns in cloud and IoT platforms (e.g., [4,1,11,15]). As a result we (1) *hypothesised a minimum necessary set of model elements and relations.* Next we defined (2) a *case study protocol template* used for all cases (see [17]) and (3) *sought cases among industrial software systems* with sufficiently complex migration and evolution scenarios. We then (4) *selected three systems for which we could gain access to detailed documentation and key stakeholders*: a Geospatial Analytics System, a Water Management System, and an Edge-Cloud Analytics System. For space reasons, we can not report on them in detail. A case study protocol and detailed model description can be found in a technical report [3]. For each case we first *consulted the available documentation and plans for migration*, and then *consulted architects of the system to close any gaps in our understanding.* We used coding techniques and the constant comparison method borrowed from Grounded Theory [5] to *code the qualitative data* for context elements and relations. After the data had been coded in the first case, we formally modelled the whole case using the resulting model, then applied the same method to

[3] `http://doi.org/10.5281/zenodo.1288459`

the second system (and after that in the same manner the third system) and thereby iteratively refined the codes, model elements, and relations. Next, we re-modelled the first system (and after that the second and the third system) with the resulting model and resolved any arising inconsistencies. The result is (5) a *semiformal model for platform migration and in-platform evolution*, and three derived case models.

4 Migration and Architecture Evolution

4.1 Evolution Attributes and Process

An application can be considered as comprising its concrete realization (architecture, software components, etc.), and an associated set of attributes (functional and non-functional requirements, etc.) and constraints (dependencies, licenses, legal limitations, etc.) that provide a context that describes and governs its function and usage. The two facets are interdependent: the introduction of a new software component affects the attributes, and predetermined attributes and constraints can affect which components are compatible in a design situation.

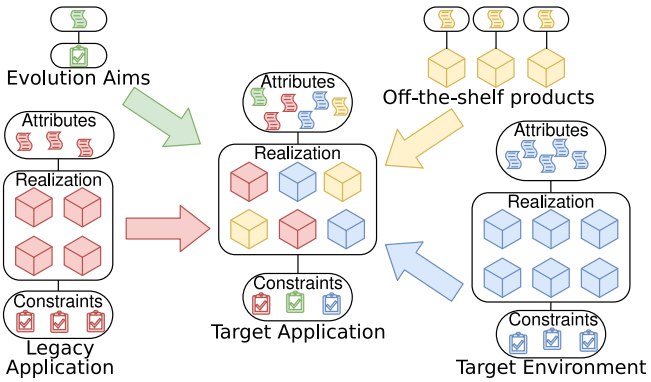

Fig. 1. Element sets involved in a software migration process (simplified). The Target Application results from a set of decisions trying to fulfil the attributes and constraints of the Evolution Aims while remaining compatible with the Target Environment. Legacy components, Target Environment offerings and off-the-shelf products are available for use as long as they satisfy these constraints.

Software evolution can then be described as the transformation of the specific realization with specific attributes of a *Legacy Application* $\mathcal{L}$ into a new realization with its own attributes, the *Target Application* $\mathcal{T}$. The latter is often deployed in a new *Target Environment* $\mathcal{E}$, as in the specific case of software migration. As shown in Figure 1, $\mathcal{T}$ results from a mix of different element sets. The coonstraints will result from those carried over from $\mathcal{L}$, the constraints of $\mathcal{E}$,

and whatever additional constraints our *Evolution Aims* $\mathcal{A}$ dictate. In addition, $\mathcal{A}$ and $\mathcal{L}$ provide a minimum set of attributes, that the application must realize. The evolution process is then a search for components and configurations (realization) that are compatible with both the attributes and constraints sets of $\mathcal{T}$. Depending on the context, this idealized view has to be modified: some of the sets may be empty, $\mathcal{A}$ may be minimally described, $\mathcal{L}$ may be insufficiently documented, etc. The relative experience and preferences of architects are also an unknown factor. As a result, the problems and choices that may emerge during the transformation process can not be anticipated beforehand.

Consequently, our focus has been reduced to a minimal core: a single evolution step, either moving (importing) the component from one environment to another, or adapting it (refactoring) to satisfy specific requirement(s). In a migration context, moving effectively copies a component from $\mathcal{L}$, or from some list of $\mathcal{O}$, and imports it into a new environment ($\mathcal{T}$ as deployed in $\mathcal{E}$). The presence of the import results in a set of mismatches with the constraints, attributes, and existing state of $\mathcal{T}$, setting off a sequence of adaptation steps in what is in essence an experimentation cycle. If a satisfactory solution to each mismatch is found (or it is considered an affordable trade-off), the next component import from $\mathcal{L}$ takes place, gradually building up $\mathcal{T}$. If the mismatches of a specific import cannot be resolved, then alternative equivalent elements can be imported and tested from the offerings of $\mathcal{E}$ or $\mathcal{O}$. The context of each adaptation decision is thus limited to the imported component and its immediate *operational environment*, and a concurrent adaptation of both the implementation and the attributes takes place, resulting in the final $\mathcal{T}$.

4.2 Migration Scenarios and Model Attributes

Based on the finding from the previous sections, we examined three systems, each representing scenarios typically encountered in industry:

- Migrating a legacy monolithic system into a cloud platform, given a set of business, technical, and legal constraints
- Re-architecting a legacy monolithic system into a cloud-deployed microservice-based architecture
- Requirement-based dynamic selection and allocation of system components on a cloud-edge platform

The three systems, as well as the resulting detailed model itself, are presented in more detail in the online appendix[3]; here we only present an overview. The model comprises a generic *domain knowledge* representation model which includes five sets (*Capabilities*, *Applications*, *Architecture*, *Technology*, *Constraints*), and a *software description* model using the elements defined in the former to provide a concrete definition of software products. The main goal of our model is to limit the possible options presented to the architect to manageable levels. The *selection of suitable solutions*, which lies at the root of the trial-and-error approach described above, is carried out by matching individual attributes, either

for compatibility (e.g., compatible interfaces or licenses, written in a programming language supported by $\mathcal{E}$) or to discover alternatives (e.g., products with the same functionality). The latter process can require traversing of the domain knowledge *type hierarchy* (e.g., the various categories of data bases or file systems for persistent storage functionality), or of the *versions of a specific product* (e.g., the versions of an ecosystem platform and packages supported).

The more attributes are matched, the narrower the resulting option set. In practice there are usually mismatches which have to be resolved either by resorting to close analogues, or to integrating solutions (e.g., Enterprise Integration Patterns or specific plugins and extensions) which can be used to "bridge" the operational environment of the component with the requirements being pursued. Using this matching approach, a component can be adapted to its new environment. Its external attributes (e.g., its functionality, external interfaces, implementation languages) then become attributes of the composite system. Conversely, since the only thing required to use a component are its external attributes (interfaces, functionality, constraints), *placeholder components* can be defined, whose exact implementation is left undefined. This means that a placeholder can either be dynamically instantiated by using one of multiple compatible solutions (e.g., a placeholder "SQL database"), be implemented by some third party to specification ("compatibility by design"), or can represent a wrapper for an otherwise non-compatible component.

Finally, the model provides concrete architectural guidance by associating attributes and constraints, as well as technologies or applications, with specific architecture patterns and strategies, as well as by indicating the (in)compatibility between patterns. From practical experience with using the model during the microservice-based re-architecting, we realized that the more inexperienced users are overwhelmed by the breadth of architectural options, showing that thearchitecture perspective on its own is unsuitable as an entry point to the design process, and that it had to be refined through combination with other constraints and attributes. This led to the creation of architecture templates, representing common configurations of patterns in combination with components types, functionalities, and constraints.

5 Discussion and Conclusions

Based on the data from a comprehensive literature study and practitioner reports, software evolution of three industrial case scenarios, and interaction with key stakeholders of these projects, we have derived a model for easing software architecture evolution decisions. The scenarios were used to evolve the model, as well as test and validate its functionality. Perforce, such a model operates on a number of assumptions that may not always exist in practice. We assume that a common language between stakeholders exists, so that the same term (e.g., a *Capability* or *Pattern*) will be commonly understood and used in the same way. Likewise, we assume that the descriptions provided to populate the knowledge repository and instanciate our models are accurate and up-to-date. A further

problem, which is common to such approaches, is the analysis of the impact of, and tradeoffs between, multiple quality attributes. These are hard to quantify, and vary with context. Thus we can only present a rough ranking of attribute importance and impact, but it is left to the architect to evaluate them. Likewise, the cumulative impact of the individual decisions can only be assessed at the end of the evolution process. The model can support, but not replace the architect. Factors such as personal preferences, past experience, and existing commitments to some technology, can not be anticipated. Nevertheless, our experience working with the model shows that it provides a number of benefits. It creates a common, centrally managed, knowledge repository, which provides a consistent reference model and a framework that links software products, software architecture aspects, business requirements and constraints, and technologies, and allows the easy discovery of interrelations.

The model is also extensible, as new elements, domains, and views can be added easily, while maintaining the same structure. The recursive structure which the *software description* supports means that a variety of offerings can be represented and recomposed at will, with varying levels of detail depending on the context: from a basic template to a complete description. The ability to define templates enforcing consistency in certain key areas is fundamental for industrial ecosystems, and can be used to provide architectural guidance. Furthermore, although developed in the context of software migration, we believe that the model is in practice generalizable for all cases of architecture evolution from greenfield to brownfield, which has much the same requirements and involves the same elements.

Using the model first requires populating the knowledge repository. Though this process can be assisted by tools, it still represents a considerable investment of time and effort. This is an inherent disadvantage of all such approaches, but we believe that the resulting benefits, once this repository is established, outweigh the investment, especially from the view of a keystone organization that has to manage large collections of offerings, and ensure a minimum level of consistency and compliance among the various participants within an ecosystem. The model has the advantage of needing only a high-level description of its elements and features to work; it does not require a full-fledged architecture reconstruction. The full and accurate description of individual components can be deferred to a later time, if and when necessary for their further decomposition. We also expect that, in the context of large ecosystems, software products will share many common elements, encouraging frequent reuse of the generated models, or, analogous to our architecture templates, the creation of prototype applications or application modules. Working directly with the model is often not practical, as the number of attributes involved grows geometrically; this was most clearly seen in the WMS scenario, where the large pattern set had to be structured in pre-defined combinations to become usable. It is therefore our future research plan to realize a web-based decision support tool for the model.

Acknowledgements. This work was partially supported by FFG project DECO (no. 864707) and Austrian Science Fund (FWF) project ADDCompliance.

References

1. New whitepapers on cloud migration: Migrating your existing applications to the AWS cloud (November 2010), https://aws.amazon.com/blogs/aws/new-whitepaper-migrating-your-existing-applications-to-the-aws-cloud/
2. Capilla, R., Jansen, A., Tang, A., Avgeriou, P., Babar, M.A.: 10 years of software architecture knowledge management: Practice and future. J. Syst. Softw. 116, 191–205 (2016)
3. Fairbanks, G.H.: Just Enough Software Architecture. Marshall & Brainerd (2010)
4. Fehling, C., Leymann, F., Retter, R., Schupeck, W., Arbitter, P.: Cloud Computing Patterns. Springer (2014)
5. Glaser, B., Strauss, A.: The Discovery of Grounded Theory. Aldine (1967)
6. Gupta, A.: Microservice design patterns. http://blog.arungupta.me/microservice-design-patterns/ (April 2015)
7. Harrison, N.B., Avgeriou, P., Zdun, U.: Using patterns to capture architectural decisions. IEEE Software 24(4), 38–45 (Jul 2007)
8. Hohpe, G., Ozkaya, I., Zdun, U., Zimmermann, O.: The software architect's role in the digital age. IEEE Software 33(6), 30–39 (Nov 2016)
9. Hwang, J., Huang, Y.W., Vukovic, M., Anerousis, N.: Enterprise-scale cloud migration orchestrator. In: 2015 IFIP/IEEE International Symposium on Integrated Network Management (IM). pp. 1002–1007 (May 2015)
10. Jamshidi, P., Pahl, C., Chinenyeze, S., Liu, X.: Cloud Migration Patterns: A Multi-cloud Service Architecture Perspective, pp. 6–19. Springer International Publishing (2015)
11. Jamshidi, P., Pahl, C., Mendonça, N.C.: Pattern-based multi-cloud architecture migration. Software: Practice and Experience 47(9), 1159–1184 (2017)
12. Lytra, I.: Supporting Reusable Architectural Design Decisions. Ph.D. thesis, University of Vienna (2015)
13. Medvidovic, N., Taylor, R.N., Rosenblum, D.S.: An architecture-based approach to software evolution. In: Proceedings of the International Workshop on the Principles of Software Evolution. pp. 11–15 (1998)
14. Plakidas, K., Schall, D., Zdun, U.: Evolution of the R software ecosystem: Metrics, relationships, and their impact on qualities. J. Syst. Softw. 132, 119–146 (2017)
15. Reinfurt, L., Breitenbücher, U., Falkenthal, M., Leymann, F., Riegg, A.: Internet of things patterns. In: Proceedings of the 21st European Conference on Pattern Languages of Programs. pp. 5:1–5:21. ACM, New York, NY (2016)
16. Richardson, C.: Microservices.io. http://microservices.io/
17. Runeson, P., Host, M., Rainer, A., Regnell, B.: Case Study Research in Software Engineering: Guidelines and Examples. Wiley Publishing (2012)