Computing (2017) 99:129–145
DOI 10.1007/s00607-016-0520-y

Architectural refactoring for the cloud: a decision-centric view on cloud migration

Olaf Zimmermann[1]

Received: 3 August 2016 / Accepted: 30 September 2016 / Published online: 21 October 2016

Abstract Unlike code refactoring of programs, architectural refactoring of systems is not commonly practiced yet. However, legacy systems typically have to be refactored when migrating them to the cloud; otherwise, these systems may run in the cloud, but cannot fully benefit from cloud properties such as elasticity. One reason for the lack of adoption of architectural refactoring is that many of the involved artefacts are intangible—architectural refactoring therefore is harder to grasp than code refactoring. To overcome this inhibitor, we take a task-centric view on the subject and introduce an architectural refactoring template that highlights the architectural decisions to be revisited when refactoring application architectures for the cloud; in this approach, architectural smells are derived from quality stories. We also present a number of common architectural refactorings and evaluate existing patterns regarding their cloud affinity. The final contribution of this paper is the identification of an initial catalog of architectural refactorings for cloud application design. This refactoring catalog was compiled from the cloud patterns literature as well as project experiences. Cloud knowledge and supporting templates have been validated via action research and implementation in cooperation with practitioners.

Keywords Architectural decisions · Architectural patterns · Cloud computing · Knowledge management · Reengineering · Refactoring · Software evolution and Maintenance

Mathematics Subject Classification 68N99

✉ Olaf Zimmermann
ozimmerm@hsr.ch

[1] University of Applied Sciences of Eastern Switzerland (HSR FHO), Oberseestrasse 10, 8640 Rapperswil, Switzerland

1 Context and overview

Motivation and overview Software-intensive systems often have to be reengineered, e.g., due to unpredictable context changes and technology innovations that occur during system lifetime. Furthermore, modern development approaches (e.g., those summarized as agile practices) advise software engineers to evolve their designs and implementations in small, frequent iterations that embrace change. Many reengineering activities affect the software architecture of these systems; service-oriented architectures and cloud deployment pose particular challenges. Given the success of the agile practice of code refactoring, it is rather surprising that architectural refactoring has not taken off yet—refactoring to patterns (in response to design smells) has focussed on the code level so far [23]; in cloud migration, other reengineering techniques have been used [16,19,29].

This paper approaches reengineering from a different view than previous attempts. It first positions architectural refactoring as a task-centric technique for restructuring an existing architecture (along with its representations) that revisits the architectural decisions made in the context of quality stories and architectural smells. Next, the paper establishes a decision- and task-centric template for architectural refactorings and instantiates it in several examples. The paper also scores existing patterns regarding their cloud affinity and outlines a catalog of cloud application refactorings based on two cloud user stories. Finally, it gives an outlook on how practitioners and researchers can apply and advance a practice of architectural refactoring for the cloud.

Architectural refactoring and its context An architectural refactoring revisits certain architectural decisions and selects alternate solutions to a given set of design problems. An AR may alter the internal structure of a system (in any architectural viewpoint [26]), but does not change the external behaviour of this system (i.e., functional capabilities and interface contracts at the system and/or service boundary). This positioning puts less emphasis on structure than previous ones—and focusses on design rationale and related tasks instead. In this setting, decision making in itself is seen as a set of interrelated engineering tasks; the revision of a group of architectural decisions causes additional *re*engineering tasks. Such tasks include:

- Tasks to realize structural changes in a design. Such architectural changes are similar to code refactorings, but have a larger scale and scope, e.g., they deal with components, subsystems and systems of systems with their interfaces ("All architecture is design but not all design is architecture. Architecture represents the significant design decisions that shape a system, where significant is measured by cost of change" [4]).
- Implementation and configuration tasks in development and/or operations, (depending on the viewpoint the architectural refactoring pertains to).
- Documentation and communication tasks resulting from architectural decisions, e.g., modelling activity, technical writing assignment, or design workshop preparation/facilitation/post-processing.

In the context of legacy system modernization and refactoring for the cloud, some of the architectural decisions to be revisited may have been made and executed a long time ago; if so, the rationale for the original decision as well as the information

about the executed tasks might have been lost. On agile projects, the design might be documented mostly tacitly, e.g., in code, in agile planning tools, or in the individual memory of project team members. Hence, architectural refactoring cannot concentrate solely on formal specifications (such as architectural models); it must treat tasks (of the types introduced above) as first-class citizens in its practices and supporting tools.

2 Defining architectural smells and architectural refactoring

Generally speaking, the goal of a refactoring is to improve a certain quality while preserving others. For instance, code refactoring is defined as a technique for restructuring an existing body of code that alters its internal structure without changing its functionality ("a refactoring is a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior" [10]). Code refactorings can work with machine-readable entities such as packages, classes and methods; hence, they can leverage mature data structures from complier construction such as symbol tables and Abstract Syntax Trees (ASTs). On the architecture level, we deal with architecture documentation and the manifestation of the architecture in the code, as well as other runtime artefacts such as configuration files; in cloud deployments, some of these artefacts may change rapidly, e.g., the detailed application deployment topology changes every time an auto-scaling capability is triggered in order to leverage cloud elasticity. Hence, there is no single architecture AST; *Architectural Refactoring (AR)* has to deal with (a) components and connectors that might be modelled, sketched, or only represented implicitly in code and configuration files, (b) design rationale represented in template-based decision logs or in unstructured text, and (c) less obvious carriers of architectural knowledge such as meeting minutes and work items in collaboration tools and/or task management systems. When applications are (re-)architected for the cloud, additional artefacts such as service provisioning and elasticity management scripts come into play.

Taking inspiration from these definitions and context information, but also from work in the architectural knowledge management community [1], we define an *architectural smell* as the observation or the suspect that something in architecture design and its implementation is no longer adequate (i.e., good enough) under the actual requirements and current constraints for the system (e.g., its non-functional requirements including quality attribute scenarios [2]); these requirements and constraints may differ from the originally specified ones. Such architectural smell might be captured explicitly, e.g., as an element (item) of technical debt or technical risk, but it does not have to; it might also remain tacit.

An AR then is a planned and coordinated set of deliberate architecture design activities that addresses one or more particular architectural smells and improves at least one quality attribute while leaving the scope and functionality of the system unchanged. According to this definition, an AR can possibly have a negative influence on other quality attributes, due to conflicts and related trade-offs (e.g., performance vs. security). An AR manifests itself through direct or indirect changes to architectural artefacts such as code and architecture documentation (as enumerated and listed above). These architectural changes can be represented as interrelated project tasks, which have

to be executed jointly and consistently. An AR has all-or-nothing-semantics; a single update transaction on the project workspace including code and documentation should be performed when executing an AR (to preserve conceptual integrity of the design and consistency of its documentation).

2.1 Example: De-SQL (Doodle)

In their technology blog, the chief technicians at Doodle explain why and how they moved from MySQL to MongoDB after several years of production use of their collaborative online calendar scheduling service [7].

The architectural smell in this example was that it took very long to upgrade large production databases (several GB) from the current SQL database scheme to a new one. The affected quality attributes were the development and operations teams' productivity, as well as performance and scalability of database and data access layer. The root cause for the symptoms behind the smell was that relational database management systems are not designed for this particular usage scenario—they can handle it, but do not expose optimal quality attribute characteristics. The solution at Doodle was to revisit architectural decisions on database paradigm, data access layer (APIs) and database provider. A decision was made to use the schemaless, document-oriented paradigm (one flavour of NoSQL) and MongoDB as database provider. A trade off could be observed between better migration management, at the cost of new approach to database administration and the need for a new API; furthermore, the transaction boundaries had to be redesigned (e.g., commit and rollback operations, compensation) because MongoDB does not have the same consistency management characteristics as MySQL, the relational database management system used so far.

In summary, the decision for the schema-less NoSQL provider MongoDB instead of MySQL brought more flexibility. Downsides were additional and increased administra-ti-on and coding effort. This example qualifies as an AR according to our definition: it revisits certain ADs (and has tasks of various types attached to it), but it is not a code refactoring as it deals with middleware selection and configuration [32].

2.2 Supporting concept and method extension: quality stories

An AR needs a baseline, a design goal. Non-Functional Requirements (NFRs) and quality attribute scenarios serve this purpose in practitioner methods and software engineering literature [2]. These requirements engineering concepts continue to be useful in our AR context. With inspiration from agile user stories [6], we additionally propose to specify *quality stories* as a means of establishing refactoring goals.

Figure 1 presents a quality story template that also calls out the most relevant personas, i.e., stakeholders of ARs. Architectural smells arise from a lack of fulfilment of one of more of the quality goals specified in the "so that" part of the quality story.

Figure 2 establishes the architectural refactoring goal in the Doodle example as a quality story formatted according to the template from Fig. 1.

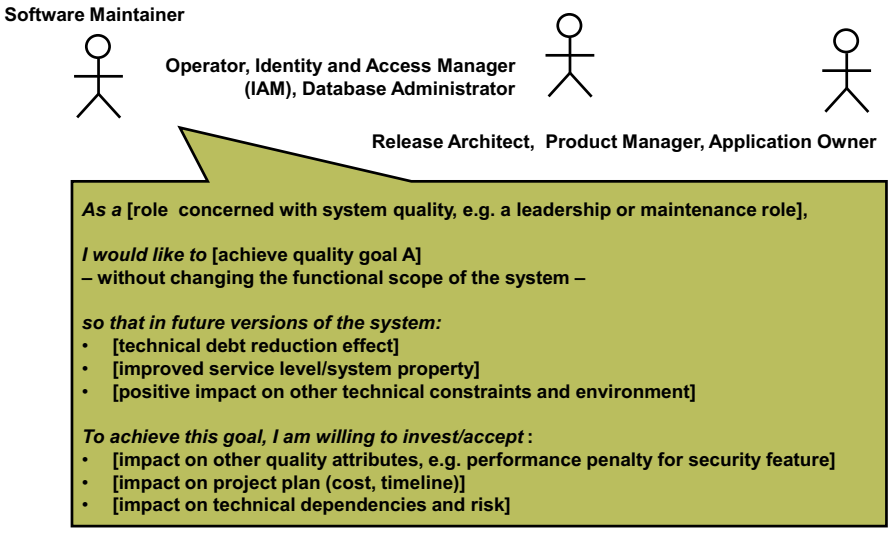

Fig. 1 Quality story template (structure) with personas

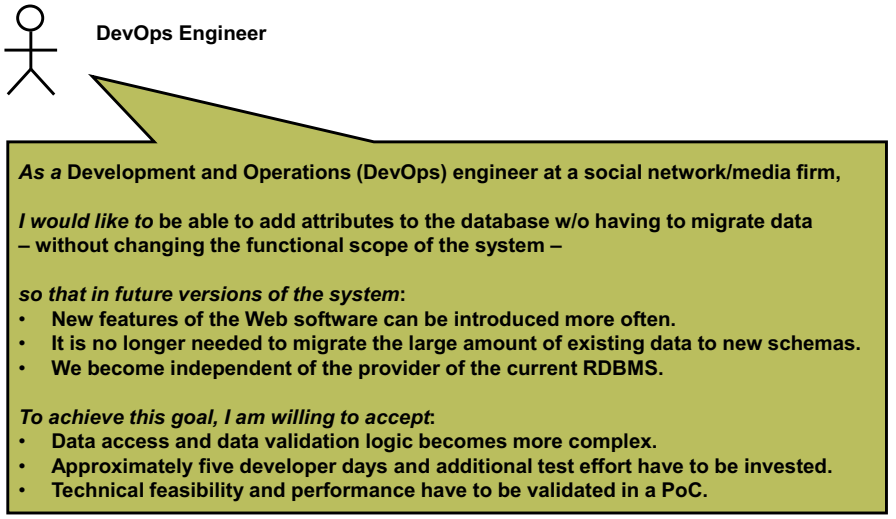

Fig. 2 Quality story template applied to database example

2.3 AR elements: from pattern format to task- and decision-orientation

Table 1 defines an AR template that calls out and elaborates upon the key elements of an AR, including the Architectural Decisions (ADs) [35] to be revisited.

This AR template has been inspired by templates used in the patterns community. For instance, design forces are listed as well as an evolution outline, which is similar to the solution sketch found in many pattern descriptions. The template also contains several novel elements, e.g., the ADs to be revisited and the AR execution tasks. The

Table 1 Decision- and task-centric architectural refactoring template

Architectural Refactoring Name	How can the AR be recognized and referenced easily? The name should be expressive, e.g., use a metaphor. Unlike pattern names (which typically are nouns), it should be able to be used as a verb in a sentence (just like names of code refactorings).
Context	Where (and under which circumstances) is this AR eligible? The context section may include information about the viewpoint and/or abstraction/refinement level in an enterprise architecture management framework such as The Open Group Architecture Framework (TOGAF) or a software engineering process such as Unified Process (UP).
Stakeholder concerns and quality attributes (design forces)	Which non-functional requirements and constraints are affected/impacted by this AR?
Architectural smell	When and why should this AR be considered?
Architectural decision(s)	Typically more than one solution exists for a given design problem. So applying an AR means revisiting one or more ADs; which ones?
Evolution outline	Which design elements does the AR comprise of (e.g., patterns for conceptual ARs)? This is the center piece of the AR, providing a solution sketch. Since the AR describes a design change, two solution sketches may be provided (one illustrating the design before the AR is applied, one the design resulting from the application of the AR). The evolution outline may optional also cover pitfalls to avoid (e.g., elements of risk) during the application of the AR.
Affected architectural elements	Which design model elements have to be changed (e. g., components and connectors (if modelled explicitly)? This is a link to the structural design space, which might have been modelled explicitly, sketched informally or is only represented (hidden) in code.
Execution tasks	How can the AR be applied? Some of these steps can possibly be automated, like the execution of many code refactorings; but not all of them as ARs operate on a higher level of abstraction. The task description might be formatted according to the metamodels and guidelines in agile planning tools and/or full-fledged design methods.

motivation for the selection of these seven AR elements is partially given in questions and explanations appearing in the right column in Table 1. Additional rationale is:

- Architectural patterns and architectural decisions have demonstrated to be efficient and effective knowledge carriers and education means [5,11,35]; substantial related knowledge engineering (a.k.a. harvesting, mining) experience has been gained and published. For instance, the patterns and AD and Architectural Knowledge Management (AKM) communities also point out the importance of finding good names [34].
- Terminology such as architectural viewpoint, stakeholder concern, quality attribute, and architectural element are well established in the software architecture community and, as a consequence, also defined in the ISO/IEC/IEEE 42010 standard for architectural descriptions [18].
- The context metaphor and its importance for knowledge sharing and design guidance have been described by Kruchten [25], who is one of the authors of the UP

Table 2 Architectural refactoring template instantiated for Doodle example Architectural refactoring template instantiated for Doodle example

Architectural Refactoring	***De-SQL***
Context	Logical viewpoint and deployment viewpoint Both conceptual level (database paradigm) and asset level (MySQL vs. MongoDB) of abstraction
Stakeholder concerns and quality attributes (design forces)	Flexibility (w.r.t. data model changes), data integrity, migration time
Architectural smell	It takes rather long to migrate existing data after an update to the data model (database schema)
Architectural decision(s)	• Choice of data modeling paradigm (current decision is: relational) • Choice of metamodel and query language (current decision is: SQL) • Choice of database management system (current decision is: MySQL)
Evolution outline	• Use document-oriented database such as MongoDB instead of RDBMS such as MySQL • Update configuration of transaction management policies and adapt database administration
Affected architectural elements	Database tier (e.g., server process, backup and restore facilities); data access layer (e.g., patterns for commands and queries, connection management)
Execution tasks	• Design document layout (i.e., the pendant to the machine-readable SQL DDL) • Write new data access layer, implement SQLish query capabilities within project • Decide on transaction boundaries (if any), implement support for them • Document the changes to database administration (e.g., command-line DDL/DML, backup)

and the 4 + 1 views on software architecture [24]. The notion of an (architectural) smell takes inspiration from code refactorings, and has been suggested already in the early work on architectural refactoring by Stal [27,28].

- Our own project experience in professional services and research and development, as well as action research conducted on multiple projects (1994 to present) suggests that a direct link from software architecture to project management exists in practice; however, such link has not been proposed in the scientific literature on software architecture yet.

AR template and one example were first published in [32]; in this paper, we extend the AR concepts and their validation with more examples, an application to cloud application development and cloud migration, and introduce a tool implementation.

Table 2 applies the template from Table 1 for the Doodle example to identify the knowledge elements that are apparent and important to know to be able to apply the same AR in a similar project context.

2.4 Generic, domain-independent architectural refactorings (ARs)

ARs residing on a rather high level of abstraction, but not yet specific to service-oriented computing or cloud application development can easily be identified in the software architecture literature (when applying the concepts and definitions from above). Some examples of such general-purpose ARs are:

- *Introduce concurrency* (*parallel processing*) Move from single program execution thread to multithreading (e.g., in mid-tier of business application to remove architectural smells such as poor throughput, blocking of input channels, and frequent timeouts.
- *Introduce cache* See Table 3 below for a full description.
- *Move responsibility* See Table 4 below.
- *Downsize mid-tier container middleware* E.g., replace Java Enterprise Edition (JEE) Application Server with a different inversion-of-control and dependency injection container to reduce management overhead, learning effort and cost (while possibly sacrificing system transaction management, application security and portability promoted by standardized JEE APIs).
- *Rightsize integration middleware* Replace custom wrapper and standard middleware with more modern or less expensive asset to improve Quality-of-Service (QoS) or cut cost while preserving the advantages of loose coupling and messaging (such as request throttling, asynchrony, etc.). This AR could also be called Change Messaging Channel Implementation (in enterprise application integration).

No basic Create, Read, Update, Delete (CRUD) operations such as "Add Architectural Element" or "Remove Architectural Element" appear in the AR list. The same philosophy is followed in code refactoring catalogs and tools; for instance, "Add New Class" does not qualify as an AR in Eclipse, but "Extract Method" does [30]. Some of the ARs in a logical, functional viewpoint [26] have an impact on component collaborations and ARs in other viewpoints, e.g., the deployment viewpoint.

Tables 3 and 4 apply the template from Table 1 to two of the common ARs, using a card-oriented layout to facilitate readability.

Concept validation To validate our template, we conducted action research initially and presented the resulting AR compilation to practicing architects from different companies with good feedback [33]. As an additional validation activity, a software engineer captured a subset of his experience gained in software development and maintenance roles in a Swiss bank in the form of ARs that comply with the template in Table 1; he also developed a prototypical Web-based *Architectural Refactoring Tool (ART)* whose metamodel draws upon the AR template from above. ART is available on GitHub and described in detail in [3].

Furthermore, we summarized an existing pattern-based reengineering method that introduces multi-tenancy support into legacy application so that such applications can be deployed to cloud offerings [12]. In total, 24 ARs have been captured so far; additional modelling and knowledge engineering work leveraging the template is considered as future work. All template elements were considered useful by knowledge engineers and knowledge consumers; in three subsequent ART tool implementations,

Table 3 An example of a general architectural refactoring (Introduce Cache)

***Architectural Refactoring:* Introduce Cache**	
Context (viewpoint, refinement level): • Logical, platform-specific refinements	*Quality attributes and stories (forces):* • Performance (response time)
Smell (refactoring driver): • A data store cannot handle concurrent queries (read requests) in reasonable time	
Architectural decision(s) to be revisited: • Lookup strategy • Data structure selection • Location of data store including replication (in memory, on disk)	
Refactoring (solution sketch/evolution outline): • Add an intermediate data structure such as memcached to speed up lookups • Design cache interface and behavior (e.g., cache size and cache cleanup policies) and cache item identifier (e.g., URI for HTML page/request caching)	
Affected components and connectors (if modelled explicitly): • Cached data and its master data store • Clients accessing this data • IT infrastructure hosting the cache (e.g., memory and/or disk storage)	
Execution tasks (in agile planning tool and/or full-fledged design method): • Analyze read-write access profile • Measure improvement potential of caching in a PoC, assess impact on test and operations (tech. risk) • Implement cache, test cache usage, document caching policies and configuration options	

Table 4 Another example of a general architectural refactoring (Move Responsibility)

***Architectural Refactoring:* Move Responsibility**	
Context (viewpoint, refinement level): • Logical viewpoint, CRC card	*Quality attributes and stories (forces):* • Cohesion and coupling metrics
Smell (refactoring driver): • A component seems to be overloaded and cluttered with diffuse features in its external interface	
Architectural decision(s) to be revisited: • Approach to modularization and component partitioning • Use of industry reference models • API design guidelines	
Refactoring (solution sketch/evolution outline): • Assess cohesion and coupling of a particular component (are responsibilities semantically related?) • Move a responsibility that breaks cohesion to another component (note: this can be an existing component or a new one; one or more responsibilities can be moved at once)	
Affected components and connectors (if modelled explicitly): • Component that currently provides a certain service (i.e., operation/feature) • Component that will take over this responsibility	
Execution tasks (in agile planning tool and/or full-fledged design method): • Updates to CRC cards in word processor, drawing tool, documentation wiki • Edit operations in UML or Architecture Description language (ADL) modeling tool • Updates to component realizations in code (note: architecturally evident coding style to be followed)	

minor adjustments were realized, but the overall approach confirmed to be adequate (i.e., providing value and being practically applicable).

2.5 Domain-specific ARs: enterprise applications, messaging, SOA

Layered enterprise applications and Web application development efforts typically work with domain-specific refinements of general architectural concepts such as the rather generic components and connectors the examples in Sect. 2.4 dealt with. For instance, an HTTP session store might be a logical component in the presentation layer of the mid-tier of a Web application [11]. Hence, the common ARs from Sect. 2.4 can be refined into domain- and style-specific ARs.

Some examples of such domain-specific ARs, which can only be identified and not fully described in this paper (due to space constraints), are:

- Push application and/or session state management from server own to database, from client or server down to database (i.e., to support horizontal scaling).
- Pull session state management up (from server to client, from database to server).
- Enrich Web client with domain logic.
- Streamline Web client (reduce client workload and processing capabilities).
- Change container technology, change dependency injection type.
- De-normalize relational database.
- Normalize relational database.
- Partition database (a.k.a. introduce sharding or add shard).
- Swap hardware type and/or hardware provider.

Candidate ARs related to messaging and enterprise application integration patterns [17] are:

- Increase number of competing consumers (in message endpoint).
- Replace Publish-Subscribe Channel with (dynamic) Recipient List.
- Collapse filters (processors), merge pipes (channels).
- Change endpoint theme (consumption strategy).
- Change aggregation strategy, change aggregation algorithm.

When refactoring towards Service-Oriented Architecture (SOA) [20], both traditional, enterprise-scale SOAs and, more recently, emerging microservices architectures, the following ARs are applicable:

- Expose component interface as a remote service a.k.a. introduce remote façade with Data Transfer Object (DTO) in a service layer [11].
- Replace scalar parameters with DTO in service interface (contract).
- Switch to service provider with different Service-Level Agreement (SLA) to improve Quality of Service (QoS).
- Transition from normalized to partitioned/replicated master data to NoSQL storage of transactional and reference data.

Each bullet item represents one candidate AR. While these ARs remain to be fully documented in the template from Table 1, their rather short but still expressive names are designed to indicate their task- and decision-oriented character already.

3 Architectural refactoring in cloud application development

In this section we move towards an *Architectural Refactoring Catalog for Cloud Migration (ARC)*. To do so, we apply the concepts and content from the previous sections to cloud computing. To be more precise, we focus on Cloud Application Development (CAD) including the modernization of software architectures with the goal to be able to deploy them to public, private, or hybrid cloud cloud offerings with an x-as-a-service service model (with x in infrastructure, platform, or software) [9].

Two user stories for cloud application development and cloud migration serve as our first step; next, we reference and revisit the ideal properties of cloud applications from the literature and score existing patterns w.r.t. their cloud affinity. With this baseline established, we present cloud ARs and discuss their applicability.

3.1 Cloud user stories and IDEAL cloud application properties

The goals of cloud application development and cloud migration can be specified as:

- "As a developer and owner of a novel Web application who is unsure about user reception and business value of this software, I would like to rapidly deploy my application into production without having to invest into hardware, data center space and operations staff (and be able to scale it up on demand) so that I can get user feedback to improve my software and the business model—without investing too many resources and taking unnecessary financial risk. To do so, I need to know the characteristics of cloud-native application architectures." (Native cloud development story.)
- "As a developer who maintains and operates an existing application on behalf of a client, I would like to move the on-premises production site into a cloud so that I no longer have to worry about security updates and other administrative tasks on the operating system and the middleware level—and my client has to spend less on operations, which frees resources to develop new features. To do so, I need to find out how my application architecture has to be refactored to be ready for the cloud (first and foremost, it should be able to run in the cloud; as a second step, it should take advantage of cloud features such as elasticity)." (Cloud modernization story.)

We refer the reader to the Cloud Computing Patterns book and website as well as supporting publications for an introduction to the IDEAL cloud application properties (serving as our cloud refactoring goals here): isolated state, decomposition (and distribution), elasticity, automated deployment, and loose coupling [9,15].

3.2 Patterns of enterprise application architecture (PoEAA) scoring

In [11] from 2003, a number of patterns for layered enterprise applications are described; most of these patterns continue to be relevant today. Table 5 evaluates selected patterns with respect to their cloud affinity (indicated by the IDEAL properties referenced in Sect. 3.1).

Table 5 Patterns of enterprise application architecture (PoEAA) and ideal CAD properties

PoEAA pattern [11]	Suitability for cloud	Comment (impact on IDEAL properties)
Client Session State	Yes and no	As good or bad as in traditional deployment (security? performance?)
Server Session State	No (I in IDEAL violated)	Also hinders scale out
Database Session State	Yes	Can use session database (e.g. NoSQL key-value storage)
Model-View-Controller	Yes (with persistent model)	Web frontends are cloud-affine
Front Controller	Yes (Web frontends)	See above
Page Controller	Yes (Web frontends)	See above
Application Controller	Yes (Web frontends)	See above
Other presentation layer patterns	Yes (Web frontends)	See above
Transaction Script	Yes	Procedures should be self-contained (stateless interactions)
Domain Model	Depends on complexity of domain model	Object tree in main memory might limit scale out (and database partitioning)
Table Module	No or implementation dependent	Big data sets problematic unless partitioned (e.g., via sharding and map-reduce processing)
Service Layer	Yes	SOA and REST design principles should be adhered to, e.g. no object references in domain model, but only instances of Data Transfer Object (DTO) in interface (larger discussion required)
Remote Façade	Yes	Can be introduced for cloud enablement of existing solutions; can wrap calls to Platform-as-a-Service (PaaS) provider to support maintainability and portability
Active Record	Limited	Good when RDB exists in cloud or when records have simple structures; complex structures can be difficult to handle for NoSQL storage (mapping need)
Row Data Gateway	Yes	Fits scale out
Table Data Gateway	No or implementation dependent	Big data sets problematic unless partitioned
System Transaction	Depends on cloud storage capabilities (NoSQL?)	Larger discussion required (e.g., consistency model and CAP properties, BASE vs. ACID [14])
Business Transaction	Yes	If cloud design best practices are adhered to (statelessness etc.)

This evaluation leads to the identification of required cloud refactorings, which we introduce and outline in the next subsection. For instance, Server Session State is no longer recommended when deploying an application to the cloud; it prevents the presentation layer to be scaled out properly (which corresponds to the "I" and "E" properties in IDEAL). In response, an AR called "Move State to Database" is introduced (see Table 6 in the following subsection).

3.3 Towards an AR catalog for cloud application development

Table 6 identifies cloud ARs in various categories. The names of the ARs again are designed to be self-explanatory (just like the names of code refactorings in books and development tools). The categories use service and deployment models as defined in [9], but also quality attributes (which typically appear in architectural smells) to foster user orientation.

An additional AR on the business level would be to "switch from flat rate to usage-based service billing (to support elasticity and cost-efficiency)". All of these ARs can be represented as instances of the task-centric template introduced in the previous section; e.g., the tasks to introduce a cache include deciding on a lookup key and clean up strategy, distribution. We refer the reader to [3] for 20 additional examples.

4 Related work

Architectural refactoring (AR) has been suggested almost a decade ago, but regrettably not been adopted much in research and practice since then.

Stal was first to blog and present on architecture refactoring [27], providing motivation and a pattern-oriented view as well as a discussion. In his OOPSLA 2007 tutorial, for instance, he presented the first catalog of architectural refactorings, which he recently updated in a book chapter [28]. He uses a standard pattern format originating from [5] to document his ARs, which include *Breaking Dependency Cycles* and *Splitting Subsystems*. In [28], he also clarifies the difference between reengineering and refactoring, and lists twelve architectural smells, including *Unclear Roles of Entities* and *Dependency Cycles*.

In 2009, Garlan introduced the concept of architectural mismatch that compares to our notion of architectural smells [13]. Fairbanks drew a connection between architecture design and risk management and connects software architecture design with agile practices. He used the evolution of the Netflix software as an example to motivate the need for architectural refactoring [8].

Since 2004, the AKM community has established a decision-centric view on software architecture. AKM contributions include metamodels, templates, methods, and tools as well as knowledge bases that compile recurring decisions, e.g., in SOA design [1,34]. In the context of ARs, the AKM notion of a *decision backlog* is particularly interesting. Such decision backlog can inform architects about the ADs that still have to be made or have been made. If *legacy decisions* that are associated with architectural smells (and therefore should be revisited) are included in the decision backlog, it can help to keep track of technical debt and plan reengineering work.

Table 6 Cloud architectural refactorings

Category	Architectural refactorings (ARs)		
IaaS	Virtualize Server	Virtualize Storage	Virtualize Network
IaaS, PaaS	Swap Cloud Provider	Change Operating System	Open Port
PaaS	"De-SQL"	"BASEify" (remove "ACID")	Replace DBMS
PaaS	Change Messaging QoS	Upgrade Queue Endpoint (s)	Swap Messaging Provider
SaaS/application	Increase Concurrency	Add Cache	Precompute Results
SaaS/application	See [9] and [31]	See [27]	See [11]
Scalability	Change Strategy (Scale Up vs. Scale Out)	Replace Own Cache With Provider Capability	Add Cloud Resource (xaaS)
Performance	Add Lazy Loading	Move State to Database	
Communication	Change Message Exchange Pattern	Replace Transport Protocol	Change Protocol Provider
User management	Swap Identity and Access Management (IAM) Provider	Replicate Credential Store	Federate Identities
Service/deployment model changes	Move Workload to Cloud (use XaaS)	Privatize Deployment, Publicize Deployment	Merge Deployments (Use Hybrid Cloud)

In service-oriented computing, a number of methods and patterns have been proposed [20,21]; typically, such approaches extend existing general-purpose ones. None of the existing techniques leverages a task-centric refactoring metaphor; in our own previous work, we have investigated the decision-centric forward engineering of SOAs and cloud applications [34,35]. In cloud computing and cloud migration, patterns have been captured, as well as methods and tools to find and use them [9,29]; pattern-centric cloud migration approaches exist as well. However, no notion of architectural refactoring for the cloud exists already (to the best of our knowledge).

Since our architectural refactoring template is based on architectural decisions and also references patterns (e.g., in solution sketches and as conceptual decision options), these approaches are complementary to ours; AR catalogs can reference such existing knowledge if this knowledge is available publicly (e.g., in the form of online pattern catalogs or repositories).

5 Discussion and outlook

Architectural decision making and architectural refactoring are key responsibilities of software architects that are underrepresented in today's methods and tools. While progress has been made in developing methods and tools around architectural decisions, architectural refactoring has not been studied much. Code refactoring is a mainstream agile practice; no architectural pendant has been established yet.

In response to these deficits, this paper picked up the early work on architectural refactoring and gave a task-centric definition of the term architectural refactoring (rather than a structural one based on pattern templates). It then introduced a quality story template that identifies potential architectural smells and an architectural refactoring template that lists the architectural decisions to be revisited as well as the design and development tasks to be conducted when an architectural refactoring is applied. The article used De-SQL, Introduce Cache, and Move Responsibility as three examples of common general-purpose architectural refactorings. The architectural refactoring template has been validated by applying it to 21 additional architectural refactorings (including a comprehensive method to enable multi tenancy in cloud applications) and implementing ART, a tool prototype to expose this AR content to practicing architects on the Web. Existing enterprise application architecture patterns were scored according to their cloud affinity; cloud architectural refactorings were identified (based on two cloud user stories).

With the task-centric representation suggested by the template and the example in this article, ARs provide an opportunity for cross-community collaboration, e.g., (a) architecture and development: AR execution may involve one or more code refactorings (which have to be stitched together), and (b) architecture and project management: AR descriptions that are organized according to the AR template can be used as planning tasks (and the need for architectural refactorings is an expression of technical debt) and (c) architecture and operations/maintenance (DevOps).

We hope for additional domain- and style-specific AR catalogs to appear in the future, e.g. for banking/financial services software, game development or DevOps. The current catalog that was outlined in this paper does not claim to be complete; its

purpose in the context of this paper is to illustrate our concepts (i.e., the AR template, quality stories, pattern scoring), and to establish a vision for domain-specific AR catalogs. We consider to document additional cloud ARs and other domain-specific ARs ourselves and have them reviewed, e.g., through writer's workshops at patterns conferences. For a broader adoption of the presented concepts, principles and practices for AR capturing are desired, similar to pattern languages for patterns authors and decision modelling guidelines [34]. Such documented practices would explain how to come up with good names for ARs, how deeply to document ARs, how to group and link ARs, etc. (when instantiating the AR template from this paper).

An open question that remains is how to execute ARs—are templates and catalogs good enough as knowledge carriers or are tools more appropriate? Our prototypical Web-based delivery of architectural knowledge in ART, which is available as an open source project, has demonstrated the potential of such collaborative knowledge engineering and maintenance tool. Code refactoring also started with a book and formal groundwork; refactoring tools, e.g., in Eclipse, were developed much later after content and theory had been established and experience had been gained. Full-scope AR tool support would need to tie in with modelling tools supporting UML or architecture description languages, as well as other engineering tools commonly used in the agile development community. Supporting tool research is currently underway.

References

1. Ali Babar, M., Dingsøyr, T., Lago, P., van Vliet, H. (eds.) Software architecture knowledge management: theory and practice. Springer-Verlag (2009)
2. Bass, L., Clements, P., Kazman, R.: Software architecture in practice, 2nd edn. Addison Wesley (2003)
3. Bisig, C.: Ein werkzeugunterstütztes Knowledge Repository für Architectural Refactoring. Masters thesis, HSR Hochschule für Technik Rapperswil (2016). https://github.com/bisigc/art
4. Booch. G.: On design. https://www.ibm.com/developerworks/community/blogs/gradybooch/entry/on_design
5. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: Pattern-oriented software architecture—a system of patterns. Wiley (1996)
6. Cohn, M.: User stories applied. Addison Wesley (2004)
7. Doodle Blog.: Doodle's technology landscape. http://en.blog.doodle.com/2011/04/14/doodles-technology-landscape/ and http://en.blog.doodle.com/2013/11/18/doodles-technology-landscape-2
8. Fairbanks, G.: What agilists should know about software architecture. http://georgefairbanks.com/blog/hangout-what-agilists-should-know-about-software-architecture/
9. Fehling, C., Leymann, F., Retter R., Schupeck, W., Arbitter, P.: Cloud computing patterns. Springer (2014)
10. Fowler M.: http://martinfowler.com/bliki/DefinitionOfRefactoring.html
11. Fowler, M.: Patterns of enterprise application architecture. Addison Wesley (2003)
12. Furda, A., Fidge C., Barros A., Zimmermann, O.: Re-engineering data-centric information systems for the cloud—a method and architectural patterns promoting multi-tenancy, book chapter accepted for Mistrik, I., et al, SABDC, Elsevier (2017) (to appear)
13. Garlan, D., Allen, R., Ockerbloom, J.: Architectural mismatch: why reuse is still so hard. IEEE Softw **26**(4) (2009)
14. Gessert F., et al.: NoSQL database systems: a survey and decision guidance. In: Proc. Of SummerSoC 2016, Computer Science—Research and Development, Springer (to appear)
15. Haberle, T., Charissis, L., Fehling, C., Nahm, J., Leymann, F.: The connected car in the cloud: a platform for prototyping telematics services. IEEE Softw **32**(6) (2015)
16. Höllwarth, T. (ed) Migrating to the cloud. http://www.cloud-migration.eu/en.html
17. Hohpe, G., Woolf B.: Enterprise integration patterns. Addison Wesley (2004)

18. ISO/IEC/IEEE, Systems and software engineering—architecture description, ISO/IEC/IEEE 42010:2011(E) (2011)
19. Jamshidi, P.: Cloud migration patterns: a multi-cloud architectural perspective. http://de.slideshare.net/pooyanjamshidi/cloud-migrationpatterns
20. Josuttis, N.: SOA in practice. O'Reilly Media (2007)
21. Julisch, K., Suter, C., Woitalla T., Zimmermann, O.: Compliance by design—bridging the chasm between auditors and IT architects. In: Computers and Security, vol. 30, Issue 6–7. Elsevier (2011)
22. Kelly, F.: AWS migration patterns. http://java.dzone.com/articles/aws-migration-patterns
23. Kerievsky, J.: Refactoring to patterns. Addison Wesley (2014)
24. Kruchten, P.: The 4 + 1 view model of architecture. IEEE Software, vol. 12, number 6 (1995)
25. Kruchten, P.: Contextualizing Agile Software Development. J. Softw. Maintenance Evol: Res. Pract **25**(4):351–361 (2011)
26. Rozanski, N., Woods, E.: Software systems architecture: working with stakeholders using viewpoints and perspectives. Addison Wesley (2005)
27. Stal, M.: Software Architecture Refactoring, OOP and OOPSLA tutorials and blog post, via; http://www.sigs.de/download/oop_08/Stal%20Mi3-4.pdf
28. Stal, M.: Refactoring Software architectures. In: Babar, A., Brown, A.W., Mistrik, I. (eds.) Agile Software Architecture. Morgan Kaufman (2014)
29. Strauch, S., Andrikopoulos, V., Karastoyanova, D., Leymann, F., Nachev, N., Staebler, A.: Migrating enterprise applications to the cloud: methodology and evaluation. In: International Journal of Big Data Intelligence, vol. 1(3). Perpetual Innovation Media Pvt. Ltd. (2014)
30. Widmer, T.: Unleashing the power of refactoring. Eclipse Magazine, July 2006 (2006)
31. Wilkes, L.: Application migration patterns for the service oriented cloud, CBDI. http://everware-cbdi.com/ampsoc
32. Zimmermann O., Architectural refactoring—a task-centric view on software evolution. IEEE Softw **32**(2) (2015)
33. Zimmermann, O.: Architectural refactoring and cloud computing aus der Sicht des Anwendungsarchitekten, OOP Presentations, 2014. English presentation material available from http://www.ifs.hsr.ch/Architectural-Refactoring-for.12044.0.html?&L=4
34. Zimmermann O, Miksovic C, Küster J.: Reference architecture, metamodel and modeling principles for architectural knowledge management in information technology services. J. Syst. Softw., Elsevier. **85**(9):2014–2033 (2012)
35. Zimmermann, O. Wegmann, L., Koziolek, H., Goldschmidt, T.: Architectural decision guidance across projects. In: Proceedings of the 12th Working IEEE/IFIP Conference on Software Architecture (WICSA), pp. 85–92 (2015). IEEE Computer Society