Architecture Practices for Complex Contexts

John R. Klein

2017

SIKS Dissertation Series No. 2017-28

The research reported in this thesis has been carried out under the auspices of SIKS, the Dutch Research School for Information and Knowledge Systems.

Promotiecommissie:

prof. dr. ir. Paris Avgeriou

prof. dr. Jan Bosch

prof. dr. Sjaak Brinkkemper

dr. Remco de Boer

prof. dr. Patricia Lago

ISBN 978-94-6295-682-7 NUR 982

Cover image "Systems of Systems of Systems" by John Klein

Published by ProefschriftMaken | | www.proefschriftmaken.nl

Typeset in LaTeX by the author

VRIJE UNIVERSITEIT

Architecture Practices for Complex Contexts

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van Doctor aan
de Vrije Universiteit Amsterdam,
op gezag van de rector magnificus
prof.dr. V. Subramaniam,
in het openbaar te verdedigen
ten overstaan van de promotiecommissie
van de Faculteit der Exacte Wetenschappen
op dinsdag 26 september 2017 om 9.45 uur
in de aula van de universiteit,
De Boelelaan 1105

door

John Robert Klein

geboren te Elizabeth, New Jersey, Verenigde Staten

promotor: prof.dr. J.C. van Vliet

Contents

Acknowledgements

I am thankful to the many people who have helped me reach this milestone.

My advisor, Hans van Vliet, has shown remarkable patience with this student. In June, 2011, we sat on a hotel patio in Boulder, Colorado, and I asked for his advice about pursuing a PhD degree. Hans immediately replied, "Let's start working together—we'll see how it goes," and so began this journey. Neither of us thought it would take this long, and I am very grateful that Hans decided to continue as my advisor after his retirement.

David Weiss has been a friend and mentor for many years. He taught me that *Software Architect* was not just a title on a business card, but that there were specific skills and knowledge that I needed to develop. David first introduced me to many of the architecture practices that I discuss in this thesis.

David also introduced me to Paul Clements, who brought me to the Software Engineering Institute at Carnegie Mellon University, and began my transition from practitioner to researcher.

I could not have done all this without the support of my managers at the Software Engineering Institute. Thank you, Linda Northrop, Mark Klein, and James Ivers.

I have had the opportunity to collaborate with many outstanding researchers. I especially want to thank Ian Gorton for introducing me to data-intensive systems, which became a focus of the research in this thesis, and Neil Ernst, who has an encyclopedic knowledge of research methods.

My thesis committee provided insightful comments and feedback.

Thanks to my paranymph, Eltjo Poort, for guiding me through the many formalities of my defense, and for translating my Thesis Summary from English to Dutch.

Finally, I thank my wife, Liese Elerin. She has been beside me through all of the ups and downs of this long adventure, and will stand beside me at my defense as paranymph. Pack your bags, Liese, we're going to Amsterdam!

Gloucester, Massachusetts and Mount Desert, Maine
July 2017

1 Introduction

Software architects are trained to apply certain unquestioned principles, such as "actively engage stakeholders" and "never depend on a particular product" [10]. As a community, we have built up practices on top of these these principles. For example, we employ stakeholder engagement methods such as the Quality Attribute Workshop [9] to engage with stakeholders and we apply architecture patterns such as Facade [57] and Layers [28] to reduce dependencies on specific products.

However, in my practical experience in several complex development contexts, I encountered cases where some long-held architecture principles do not apply. This thesis reports my research on new practices to respond to the forces that cause these principles to fail us.

This thesis focuses on software architecture practices for systems of systems. The scope includes data-intensive systems, which we consider as a type of system of systems.

This Introduction begins by introducing three main concepts: Systems of systems, data-intensive systems, and architecture practices. This is followed by a statement of my research objectives and research questions. I next present my research methods, followed by the *Thesis at a Glance*, which maps the research methods to the research questions and provides a roadmap for my work. I finish the introduction with a discussion of the origin of each chapter and my contribution to each chapter's original publication.

1.1 Background

This section provides background in the three areas that underly the research in this thesis: Systems of systems, data-intensive systems (framed as a type of system of systems), and software architecture practices.

1.1.1 Systems of Systems

A *system* is a collection of elements that together produce some result that cannot be obtained by the elements operating individually [77]. These elements of a system may themselves be large and complex, and composed of sub-elements acting in concert. We use the term *system of systems (SoS)* for the case where the constituent elements are collaborating systems that exhibit two properties [110]:

- **Operational Independence**—each constituent system is operated to achieve a useful purpose independent of its participation in the SoS, and
- **Managerial Independence**—each constituent system is managed and evolved, at least in part, to achieve its own goals rather than the SoS goals.

Maier classifies SoS into three categories, based on the type and degree of managerial control [110]:

- Directed SoS: The SoS is centrally managed. Constituent systems are built primarily to fulfill SoS purposes, with independent operation as a secondary goal. For example, during an SoS failure, a constituent system may continue to operate in a stand-alone mode that provides degraded services to its users.
- Collaborative SoS: The SoS has central management, but that management it lacks authority over the constituent systems. Constituent systems voluntarily choose to collaborate to fulfill the SoS purposes. Maier gives the example of the Internet as a collaborative SoS, with the Internet Engineering Task Force (IETF) setting standards but having no enforcement authority. Participants choose to comply with the standards if they want to be part of the Internet SoS.

- Virtual SoS: The SoS has no central management and no centrally agreed-upon purpose. Maier's example here is the World Wide Web, where there is no central governance. There are incentives for cooperation and compliance to core standards, which emerge and evolve based on market forces.

Later, Dahmann and colleagues identify a fourth SoS category, which they labeled *Acknowledged SoS* [40]. This category falls between Directed SoS and Collaborative SoS: Like a Collaborative SoS, there is central management but it lacks authority over the constituent systems. However, like a Directed SoS, constituent systems are managed to balance achieving SoS purposes and independent purposes.

There are examples of SoS in many domains. Maier's seminal paper, published in 1998, cites diverse examples such as integrated air defense, intelligent transportation, and the Internet [110]. These examples have endured and evolved, with defense systems still being a major area of interest for SoS, intelligent transportation now manifesting as self-driving interconnected vehicles built by multiple manufacturers [112], and the Internet of Things creating systems of devices interconnected for purposes that range from personal entertainment and convenience to energy conservation. Recent research investments by the European Union attest to the continuing importance of SoS [51].

We also encounter SoS in the enterprise information technology domain. Ross and colleagues identify four enterprise operating models [136]. Their *Coordination* and *Diversification* models have independent managerial and operational authority over systems, as might occur when systems from different business units or lines of business are integrated, or as the result of a merger or acquisition. Although the primary function of these systems is to support a particular business function (e.g., sales, manufacturing, or customer service), there is a need to create an SoS to support some top-level organization functions such as finance, or to exchange information such as sales orders feeding into a manufacturing forecast. Within the taxonomy of SoS types, these are Collaborative or Acknowledged SoS.

Collaborative and virtual SoS are related to industry platforms and software ecosystems. An industry platform provides the core technology that allows systems constructed by different organizations to interact to produce some value [39]. In both Collaborative and Virtual SoS, an industry platform can broker interactions between participating systems and provide incentives

to join the SoS and to behave in particular ways in the SoS. The relationships among the systems using an industry platform and among the organizations constructing those systems create an ecosystem with cooperation and competition among participants [117]. We see these today in "virtual" businesses that outsource product design, manufacturing, marketing, and order fulfillment, with business operations accomplished by orchestrating APIs on external systems that independently operate and evolve in response to market forces.

Researchers [22, 113] and practitioners [125] agree that these properties necessitate treating an SoS as something different from a large, complex system. From a software architecture perspective, these properties influence our architecture drivers, for example, emphasizing quality attributes such as interoperability while introducing constraints on evolution.

1.1.2 Data-Intensive Systems

Data-intensive systems, popularly referred to as "big data" systems, are a type of SoS. Fig. 1.1 shows a typical data-intensive system. Inputs from sensors, devices, humans, and other systems are delivered to the system, where they are ingested through a pipeline that translates and transforms the data and writes it to a persistent store. The persistent store is frequently *polygot*, employing a heterogeneous mix of SQL, NoSQL, and NewSQL technologies [139]. Users query and analyze the stored data through many types of devices and applications. The system executes in a cloud infrastructure, shared with other systems.

In most cases, the input sources are under independent operational and managerial control, often acting as an input source for several overlapping data-intensive systems. Users of the data-intensive system are autonomous, presenting a variable and evolving workload to the system. Finally, the underlying cloud infrastructure is managed and operated independently, serving the needs of many SoS.

Considered as an SoS, a data-intensive system presents the same types of architecture challenges as other SoS, discussed above. In addition, these data-intensive systems present another set of challenges. Data storage, whether durable or in-memory, is an essential function of a data-intensive system. While internet giants like Google and Amazon have developed custom storage packages such as Bigtable [32] and Dynamo [45], most developers use off-the-shelf open source packages for this function. Today, these data storage packages are usually some type of NoSQL or NewSQL [139], which reduce the

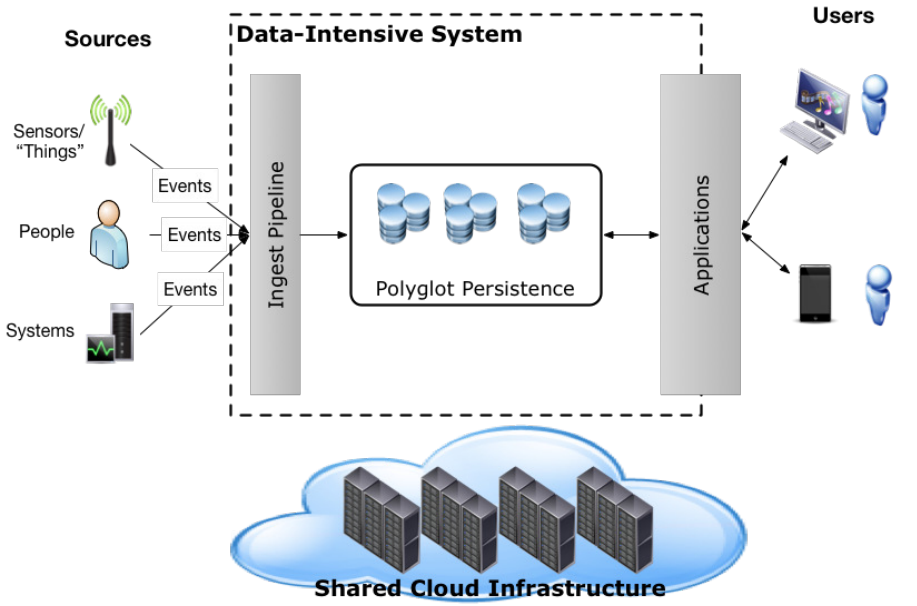


Figure 1.1: Typical Data-Intensive System

mismatch between programming language data structures (e.g., hash maps or serialized objects) and the underlying storage mechanisms. These packages also usually provide fine-grained control—on a request-by-request basis—of the tradeoff between replica consistency, availability, and network partition tolerance, which can significantly improve performance for some applications [1].

These NoSQL data storage packages bundle decisions related to several architecture concerns, as shown in Fig. 1.2. Each data storage package has an intrinsic data model (e.g., *key-value* or *structured document*), which constrains the types of data that can be efficiently stored and queried, and thus affects the complexity and performance of the system. The data storage package has built-in data distribution (partitioning and replication) that affects the replica consistency properties and availability of the system. Finally, the physical deployment of the storage elements in a distributed system at runtime determines the failure modes that the system must handle, and the ultimate availability of the system. As we discuss below in §1.1.3, this *convergence*

of concerns within the data storage package creates a significant challenge for architects: After a data storage package is selected, the decisions that it embodies about data model, distribution, and deployment ripple through much of the system architecture.

1.1.3 Software Architecture Practices

Software architecture development has four main practice areas [31]:

1. Identify architecture drivers (for example, quality attribute requirements and constraints);
2. Design the architecture;
3. Document the architecture;
4. Evaluate the architecture.

In this section, I use the term *traditional practices* to refer to the practices codified in textbooks by authors such as Bass and colleagues [10], Bosch [23], and Gorton [64]. The following sections discuss how the forces present in the SoS and data-intensive system contexts make these traditional practices ineffective in each of the architecture practice areas.

Identify Architecture Drivers Practice Area

In identifying architecture drivers, traditional practices assume a hierarchy of architecture scope and authority, visualized by Malan and Bredemeyer in Fig. 1.3. An enterprise architecture has the broadest scope, and places constraints on the software domain architectures, which in turn constrain the application architectures. This flow down of decisions implies that the architects at one level have technical authority over the architects at the level below.

Using traditional software architecture practices, architects reason about the rationale for architecture decisions as follows [10]: Business or mission goals are expressed as quality attributes of the system, which are satisfied by architecture decisions. The hierarchy of scope and decisions in Fig. 1.3 implies that goals are consistent as we flow down from enterprise to component. However, in an SoS, the managerial independence of constituent systems means that SoS goals may not flow down to system goals, and that system

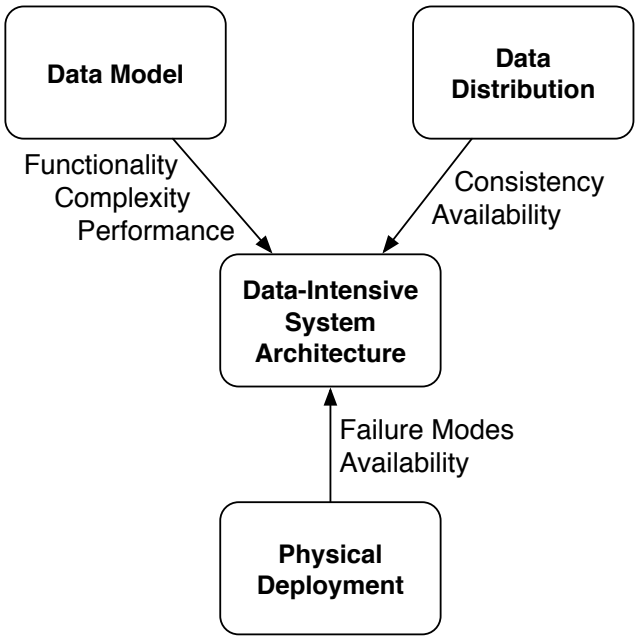


Figure 1.2: Convergence of concerns in data-intensive system architecture

goals may actually conflict with SoS goals. This can produce gaps, where SoS goals are not covered by any composition of constituent system goals, and may also create opportunities, when the composition of constituent system capabilities produces a previously unidentified benefit for the SoS. Both top-down and bottom-up analysis is needed.

The *Identify Architecture Drivers* practice area includes identification of system purpose, key functions, and constraints, which overlaps with requirements engineering practices of systems engineering. The concept of SoS was originally introduced by systems engineers (e.g., Maier [110] and Carlock [29]), and the systems engineering community has paid significant attention to requirements engineering practices for SoS, typified by the work of Gorod and colleagues [63] and Dahmann and colleagues [40]. More recently, work by Gagliardi and colleagues on the Mission Thread Workshop focuses on software architecture drivers in an SoS [56]. Compared to the other practice areas, this practice area is relatively mature.

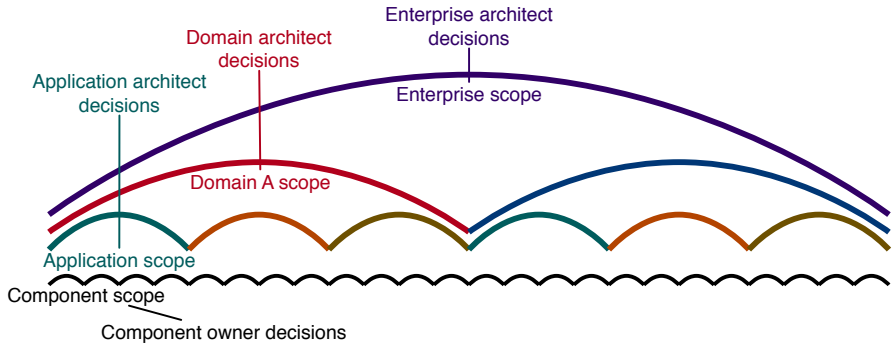


Figure 1.3: Hierarchical architecture scope and authority (after Malan and Bredemeyer [114])

Architecture Design Practice Area

Architecture design is a decision-making process [47]. The body of knowledge that is applied to make architecture decisions comprises a number of elements [31]:

- Design principles
- Reference architectures
- Architectural design and deployment patterns
- Tactics (fine-grained decision to refine patterns and reference architectures in order to promote a particular quality attribute)
- Externally-developed components

SoS architecture design is compositional, making decisions that involve selecting systems and then composing those systems. This makes the *externally-developed components* knowledge element notable in the SoS context, because all constituent systems are generally developed (and evolved) external to the SoS.

When designing with externally-developed components, architects often apply the fundamental design principle of information hiding [127]. This principle is used, for example, in the Facade pattern [57] or Layers pattern [28], to decouple a system from a specific underlying off-the-shelf platform.

Furthermore, this approach supports the agile practice to defer decisions until the last responsible moment [131]. The decision about which off-the-shelf platform to use does not affect layers above the abstraction layer, and so we can defer that selection decision until late in the design.

However, in the SoS architecture context, introducing an abstraction for externally-developed technology can be difficult. An example of such an abstraction is the Structured Query Language [78] used for data storage packages in data-intensive systems. This abstraction is large and complex, and makes tradeoffs regarding system qualities (for example replica consistency versus availability [139]) that may not satisfy all designs. More generally, in SoS architectures, an abstraction layer can incur high development cost and force undesirable quality tradeoffs, and the principle of information hiding must be set aside in favor of designing the system around the features and qualities of a particular selected technology. In this case, the design practice shifts from creating abstractions to making the best selection early in the architecture design process.

This shift is shown in Fig. 1.4. On the left side, we see the traditional practice, where the architecture drivers shape the architecture design, and technology is selected later based on the desired architectural qualities. In contrast, the right side shows the flow in an SoS, where the technology must be selected as one of the first architecture decisions, and here the technology selection influences the architecture design.

Although technology selection influences architecture design in many systems, this is a dominant and unavoidable characteristic of SoS architecture design.

Architecture Documentation Practice Area

Architecture documentation has three main uses: Communication, education, and analysis [34]. SoS architecture documentation approaches such as the Department of Defense Architecture Framework (DoDAF) [48] structure the information to satisfy these uses for an SoS. As noted above, designing an SoS architecture is a compositional activity, and so the SoS architecture documentation depends on information about the constituent systems. However, the constituent systems necessarily precede an SoS, and so the scope of system architecture documentation is generally limited to the independent stand-alone operation of each constituent system.

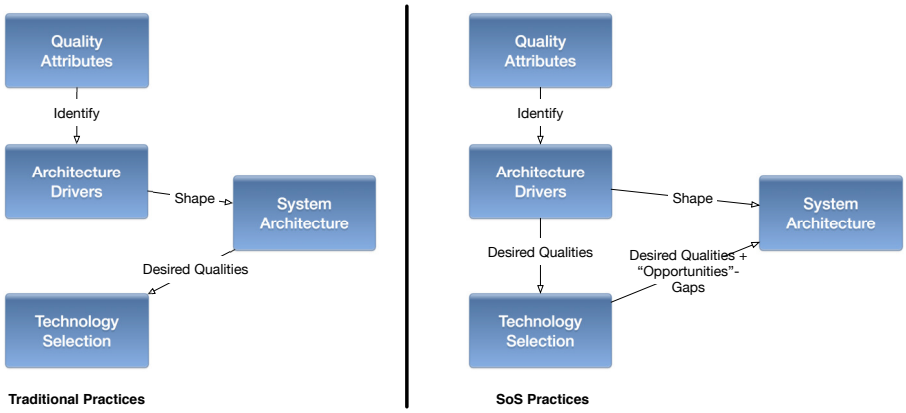


Figure 1.4: Technology Selection Flow

The SoS architect has both analysis concerns and design concerns. Analysis of the composed SoS considers functions and qualities such as performance, security, and availability. In design, the SoS architect focuses on the interfaces provided and required by the constituent systems, with particular attention to architecture mismatch [59]. The SoS architect does not focus on the architecture structures within a constituent system. Constituent system architecture documentation, scoped to address stand-alone operation, may need to be enhanced to satisfy the needs of the SoS architect. Interface-related behavior that might be considered private or insignificant when operating stand-alone can contribute to architecture mismatch in an SoS. For example:

- When servicing a request on a system's interface, the system locks records in an external database. During stand-alone operation, this is considered a private implementation detail, however in the SoS, the database is shared among several systems and this behavior is visible to those systems as increased database latency.
- Two constituent systems use the same registry for service discovery, however each system has different behavior if a request to the registry times out: One system falls back to cached information, while the other falls back to a static default configuration. Again, each constituent system treats this behavior as a private implementation detail, but in an SoS this behavior can prevent these systems from coordinating.

Architecture documentation practices would not usually include these types of behavior in the system's architecture documentation. The behavior may be considered private and is explicitly excluded, or it is not considered to be architecturally significant, in which case the decision is deferred to implementation and so it is not included in the architecture documentation. For example, in the second case above, either approach allows a system to satisfy its stand-alone functional and quality attribute requirements.

In order to fill these gaps, constituent system architecture documentation must be enhanced with additional information to meet the needs of the SoS architect.

Traditional architecture documentation practices are based on collaboration with stakeholders to explicitly identify and satisfy stakeholder needs (e.g., ISO-42010 [79], the Views and Beyond approach [34], and the Viewpoints and Perspectives approach [137]). These practices would identify the SoS architect as a stakeholder in the constituent system, and create documentation that satisfies those needs. However, these practices fail because the SoS does not usually exist when the documentation for the system architecture is being created, and so the SoS architect's concerns cannot be identified. We can adapt traditional practices to produce additional or enhanced system architecture documentation to satisfy the SoS stakeholder needs, but this cannot occur until the SoS is identified.

The managerial independence of the constituent systems can create additional stress on stakeholder collaboration. Architecture documentation stakeholder collaborations include those among architects at different levels of the hierarchy in Fig. 1.3. However, in an SoS, the constituent systems are often developed by separate organizations that have competing or conflicting interests that interfere with collaboration among the architects. I was an SoS architect during several corporate acquisitions, where we were acquiring a company that developed one of our constituent systems. During the acquisition, communication was restricted: I had my requests for architecture information reviewed by lawyers and I had to justify the need for each request to high level corporate executives. As we discuss below in Chapter 6, the interactions among the architects can become one-time high ceremony events, with no opportunity for follow-up or clarification.

Even in the case where there is a mutual desire to collaborate, it may not be feasible because of lifecycle phase differences among the constituent systems. For example, a constituent system may have no ongoing development activities, and hence no responsible architect available to represent the sys-

tem. In cases like this, the SoS architect must articulate a precise request for information, and someone must estimate the cost for a representative of the the constituent system to respond. The SoS owner and the constituent system owner negotiate to fund the constituent system architect's work to respond, and eventually the constituent system representative is directed to supply the requested information to the SoS architect. Again, there is often little or no ability to iterate the requests or seek elaboration of the responses.

Architecture Evaluation Practice Area

The purpose of architecture evaluation is to assess whether desired functions and qualities are likely to be achieved in a system constructed from a particular architecture [35]. Architecture evaluation practices have evolved from a single event that occurs prior to beginning implementation [35] to iterative events that are interwoven with prototyping and development [121]. Practices have also scaled up from software architecture evaluation, with methods emerging for system and SoS architecture evaluation [85]. However, the SoS context creates challenges that are not addressed by these extensions of traditional practices.

An architecture is produced by a series of decisions (as discussed above in §1.1.3), with some decisions justified by requirements and assumptions about the system's operational environment. In an SoS, one constituent system's operational environment is comprised of the other constituent systems. These other constituent systems will evolve independently and original requirements and assumptions that were used to evaluate the architecture may no longer hold true. Garlan labels this mutability of requirements and assumptions as "uncertainty", and identifies sources that include humans in the loop, physical mobility of platforms, and rapid evolution [62], and he notes that traditional architecture practices that rely on design time analysis to predict and assure the properties of the system are insufficient in the "uncertain world". As a specific example, consider the reference architecture shown in Fig. 1.1. The inputs, the user workloads, and the shared infrastructure are usually each under separate operational and management authority, and so each is likely to evolve independently from the core data-intensive system.

In this context, Garlan calls for *closed loop approaches*, in which a system monitors itself to continuously compare the current operating environment to the design-time assumptions, and reacts by adapting or raising an alarm. Architecture practices to accomplish this must consider approaches such as

runtime observability to assess system properties. At the scale of many SoS and data-intensive systems (i.e. 1000s of computer nodes), these approaches point to architecture design practices that automate the generation of observability monitoring components, and automate aggregation of telemetry streams to assess whether the system is achieving the needed properties.

1.2 Objectives and Research Questions

The research reported in this thesis was performed in a consulting context. The objective was always to improve system development outcomes for my client, and more specifically, to provide validated architecture practices that were systematic and would produce successful and repeatable results for the client's organization.

My experiences with a number of failed projects in the SoS context inspired this research. These included:

- The United States Army's Future Combat System, which was based on the System of Systems Common Operating Environment platform. This SoS was to provide battlefield command and control, involving many system owners and development contractors, all with conflicting goals. This project was cancelled in 2009.
- The United States Nationwide Health Information Network, which started in 2004. This system was a highly federated approach to allow healthcare providers at any location to securely access a patient's healthcare records and to provide aggregation of patient records for public health monitoring and research. This architecture went through many iterations and the system is still not widely used.
- My industrial experience, as a chief software architect leading the architecture integration of large-scale business systems after corporate mergers and acquisitions. In these projects, the acquired organizations often would not cooperate with us or collaborate on the architecture integration, and these projects always took longer and delivered less capability than planned.

On these and other projects, I saw clients struggle to use traditional software architecture practices, such as those described above in §1.1.3, to develop large and complex SoS. These practices were not producing satisfactory outcomes: Projects took much longer and delivered less value than planned. This produces our main research question:

> **RQ:** How should traditional software architecture practices be changed to address the challenges of large scale, complex SoS contexts?

An initial survey of research and practice literature found little on the subject, and so I began my research by considering two exploratory sub-questions. First, B

> **RQ-0A** What is the state of research in SoS architecture?

In order to triangulate my client's experience with the broader state of the practice, I asked a second sub-question:

> **RQ-0B** What is the state of the practice in SoS architecture development and analysis?

These studies indicate that the *Identify Architecture Drivers* practice area is relatively mature (as discussed above in §1.1.3); however, there are a number of open issues in research and practice in the remaining areas. At this point, I created a research plan that addressed one issue in each of the remaining practice areas.

In the Architecture Design practice area, there are a number of elements in the architecture design body of knowledge (discussed above in §1.1.3). As networked distributed systems, SoS software architecture draws from the related domains of service-oriented architecture, enterprise information technology architecture, and distributed system architecture. In these areas, knowledge such as principles, reference architectures, and patterns have been well studied (e.g., Erl [50] and Rischbeck & Erl [135] in service-oriented architecture, Fowler [55] and Hohpe [74] in enterprise systems, and Dean [44] in large-scale distributed systems). The remaining design knowledge element, externally-developed software, is directly relevant to SoS architecture design, because all constituent systems are externally developed. My consulting work at this point turned

to data-intensive systems. (As discussed in §1.1.3, data-intensive systems are a type of SoS.) In designing these architectures, technology selection is a fundamental architecture decision, and this leads to a sub-question about how to help less experienced architects make these decisions:

> **RQ-1** What decision support is needed to improve the efficiency and quality of technology selection designs of scalable data-intensive systems?

Turning to the Architecture Documentation practice area, the managerial independence of the constituent systems can limit the stakeholder collaboration that traditional documentation practices depend on (discussed above in §1.1.3). Without stakeholder collaboration, we can still address concerns such as performance and availability by observing the constituent system and making measurements. However, addressing architecture mismatch concerns can require access to expert knowledge and documentation about the constituent systems. In the SoS context, where requests for information can be highly constrained and scrutinized, an SoS architect needs a validated approach to scope an information request. This leads to the sub-question:

> **RQ-2** When an existing system will be introduced into an SoS, what additional architecture documentation is needed?

The final sub-question addresses the Architecture Evaluation practice area. I considered focusing on emerging SoS architecture evaluation methods, investigating issues such as how to assess or improve the coverage of an evaluation method, or how to adapt evaluation methods when stakeholder participation is limited by constituent system managerial independence. However, I was unable to identify data sources to support retrospective analysis or to recruit participants for an experiment or case study. I decided to focus on runtime observability as an extension of traditional architecture evaluation practices, as discussed above in §1.1.3. Runtime observability can provide initial and ongoing assurance that quality properties of the system are satisfied, which produces the sub-question:

> **RQ-3** What approaches can be used to improve the runtime observability of a large-scale data-intensive system?

1.3 Research Methods

My research uses a number of qualitative methods that are common in software engineering research.

- Systematic Literature Review—This method is an objective and repeatable process to critically identify, evaluate, and synthesize findings from published studies, in order to answer pre-defined research questions [49, 88]. I use this method to characterize the state of SoS architecture research and to begin to understand how this context is different from mainstream software architecture (Chapter 2).

- Semi-structured Interviews—This method mixes open-ended and specific questions to elicit both foreseen information and unanticipated information [140, 105]. I use this method to understand the state of the practice of SoS architecture (Chapter 3) and to understand the state of the practice of runtime assurance (Chapter 7). In both cases, I use the *constant comparison method* for data analysis [140], where each interview is coded and analyzed, and the results are checked in the next interview.

- Single Case Mechanism Experiment—This method tests a single case of a treatment (an interaction between a designed artifact and the problem context that the artifact addresses) [156]. I use this method to evaluate the documentation viewpoint (Chapter 6), in the mode of an expert panel solving a simulated problem, and also use this method to evaluate model-driven observability (Chapter 8) with a real-world development team.

- Technical Action Research—This method uses an experimental artifact (an artifact that is still under development) to help a client, and simultaneously gain knowledge about the artifact in use [156]. I use this method to evaluate the QUality At Scale Knowledge Base for Big Data (QuABase-BD) decision support tool (Chapter 4) and to evaluate the Lightweight Evaluation and Architecture Prototyping for Big Data (LEAP4BD) technology selection method (Chapter 5).

1.4 Thesis at a Glance

The research context is illustrated in Fig. 1.5, which shows how the research questions and research methods are related to the goals of the thesis. The main research question is *How should traditional software architecture practices be changed to address the challenges of large scale, complex SoS contexts?* The systematic literature review (SLR) and semi-structured interviews produce descriptions of the state of the research and state of the practice in SoS architecture, to answer RQ-0.

The research then continues in three threads, one for each of the practice areas where the initial study found a need to change architecture practices.

The first thread considers the architecture design practice area. The convergence of concerns shown in Fig. 1.2 drives architects to make early technology selection decisions. I apply technical action research to develop and evaluate first a decision support tool, and then a technology selection method. Together, these answer RQ-1.

The second thread focuses on the architecture documentation practice area. Here, I develop a viewpoint to enhance documentation for systems to be used in an SoS, and then perform a single case mechanism experiment to evaluate that viewpoint to answer RQ-2.

The third thread considers the architecture evaluation practice area. We create a treatment that used model-driven approaches to implement runtime observability at scale. Semi-structured interviews support this framing, and a single case mechanism experiment is used to evaluate the approach.

Together, these three threads produce a set of new architecture practices to address the challenges of the large scale, complex SoS context, answering the main research question.

1.5 Outline and Origin of Chapters

The research presented in this thesis has been previously published, or has been submitted for review for publication. The chapters that follow are based on the publications listed below. For each, I also describe my contribution to the research underlying each publication.

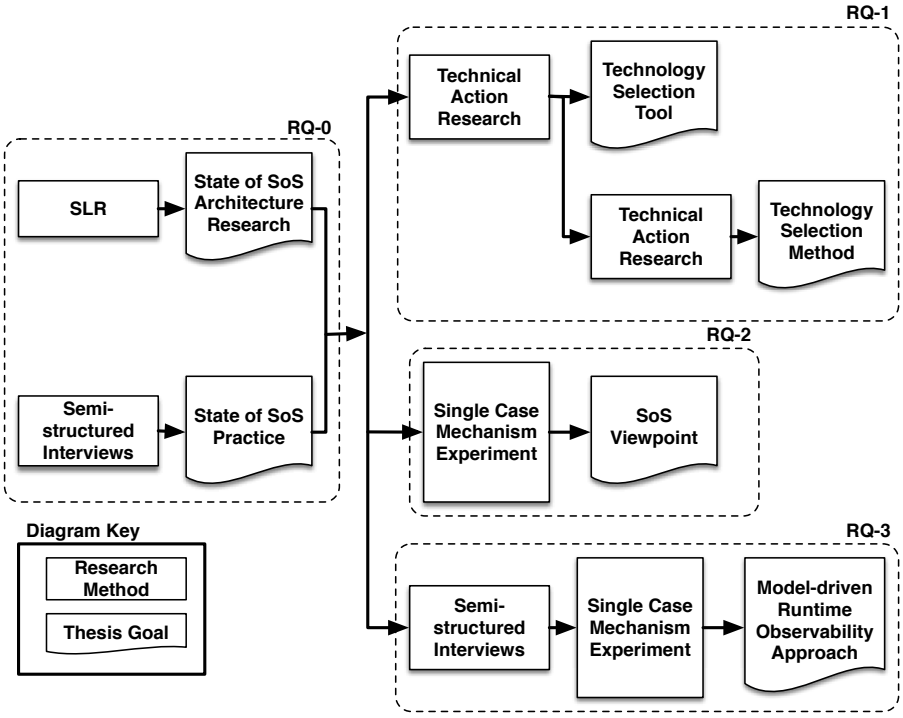


Figure 1.5: Research Context

- Chapter 2: This chapter addresses research question RQ-0A and reports the results of a systematic literature review to characterize the state of SoS architecture research. Parts of this chapter are based on:
  J. Klein, H. van Vliet, A systematic review of system-of-systems architecture research, in: Proc. 9th Int. ACM SIGSOFT Conf. on the Quality of Software Architectures, QoSA'13, Vancouver, BC, Canada, 2013.

  Contribution: Planned and conducted the systematic review, and wrote the paper.

- Chapter 3: This chapter addresses research question RQ-0B and reports the results of a series of semi-structured interviews to characterize the state of the practice of SoS architecture practice. Parts of this chapter are based on:
  J. Klein, S. Cohen, R. Kazman, Common platforms in system-of-systems architectures: Results from an exploratory study, in: Proc. IEEE/SMC Int. Conf. on System of Systems Eng., SoSE, 2013.

  Contribution: Planned the interview protocol and developed the questions, was the lead interviewer for all interviews, conducted the data analysis, wrote the paper.

- Chapter 4: This chapter addresses part of research question RQ-1. The chapter presents the design of a decision support tool for big data storage technology selection, and reports on the evaluation of the tool using technical action research. Parts of this chapter are based on:
  I. Gorton, J. Klein, A. Nurgaliev, Architecture knowledge for evaluating scalable databases, in: Proc. 12th Working IEEE/IFIP Conf. on Software Architecture, WICSA 2015, 2015.

  J. Klein, I. Gorton, Design assistant for NoSQL technology selection, in: Proc. 1st Int. Workshop on the Future of Software Architecture Design Assistants, FoSADA'15, 2015.

  Contribution: Identified and described the coupling of concerns that motivated the research. Developed and implemented the knowledge model that links quality attributes, quality attribute scenarios, architecture tactics, and product implementations. Contributed 80% of the knowledge base content in the areas of quality attributes, general scenarios, and tactics. Planned, conducted, and analyzed the results of the evaluation presented in the second paper. Wrote 60% of the first paper and 100% of the second paper.

- Chapter 5: This chapter addresses part of research question RQ-1. The chapter presents the LEAP4BD method for evidence-based selection of big data storage technology, and reports on the evaluation of the method using technical action research. Parts of this chapter are based on:
  J. Klein, I. Gorton, N. Ernst, P. Donohoe, K. Pham, C. Matser, Application-specific evaluation of NoSQL databases, in: Proc. IEEE Big Data Congress, 2015, pp. 526-534.

J. Klein, I. Gorton, N. Ernst, P. Donohoe, K. Pham, C. Matser, Performance evaluation of NoSQL databases: A case study, in: Proc. of 1st Workshop on Performance Analysis of Big Data Systems, PABS 2015.

Contribution: Developed 80% of the LEAP4BD method, led the technical action research that applied the method, conducted the analysis and interpretation of the performance analysis measurements, and wrote both papers.

- Chapter 6: This chapter addresses research question RQ-2. The chapter presents an architecture documentation viewpoint for a system to be used in an SoS, and reports on the evaluation of the viewpoint using a single case mechanism experiment on a simulation with an expert panel. Parts of this chapter were submitted as:
  J. Klein, H. van Vliet, System-of-systems viewpoint for system architecture documentation, Submitted to Journal of Systems and Software.

  Contribution: Developed the viewpoint, planned and conducted the expert panel review, analyzed the review results, and wrote the paper.

- Chapter 7: This chapter addresses (in part) research question RQ-3. This chapter presents an approach to assuring the runtime properties of a data-intensive SoS, using model-driven engineering to insert runtime observability at scale. Parts of this chapter are based on:
  J. Klein, I. Gorton, Runtime performance challenges in big data systems, in: Proc. Workshop on Challenges in Performance Methods for Software Development, WOSP-C'15, 2015.

  Contribution: Defined the solution approach presented in this paper, and wrote the paper.

- Chapter 8: This chapter addresses (in part) research question RQ-3. The chapter reports on the results of applying the model-driven engineering approach in a single case mechanism experiment. Parts of this chapter are based on:
  J. Klein, I. Gorton, L. Alhmoud, J. Gao, C. Gemici, R. Kapoor, P. Nair, V. Saravagi, Model-driven observability for big data storage, in: Proc. 13th Working IEEE/IFIP Conf. on Software Architecture, WICSA 2016, 2016.

  Contribution: Planned and led the experiment, analyzed the results, wrote the paper.

2

Systematic Review of Systems-of-Systems Architecture Research

Summary

A system of systems is an assemblage of components which individually may be regarded as systems, and which possesses the additional properties that the constituent systems are operationally independent, and are managerially independent. Much has been published about the field of systems of systems by researchers and practitioners, often with the assertion that the system-of-systems design context necessitates the use of architecture approaches that are somewhat different from system-level architecture. However, no systematic review has been conducted to provide an extensive overview of system-of-systems architecture research. This chapter presents such a systematic review. The objective of this review is to classify and provide a thematic analysis of the reported results in system-of-systems architecture. The primary studies for the systematic review were identified using a predefined search strategy followed by an extensive manual selection process. We found the primary studies published in a large number of venues, mostly domain-oriented, with no obvious center of a research community of practice. The field seems to be maturing more slowly than other software technologies: Most reported results described individuals or teams working in apparent isolation to develop solutions to particular system-of-systems architecture problems, with no techniques gaining widespread adoption. A comprehensive research agenda

for this field should be developed, and further studies should be performed to determine whether the information system-related problems of system-of-systems architecture are covered by existing software architecture knowledge, and if not, to develop general methods for system-of-systems architecture.

2.1 Introduction

A system is a collection of elements that together produce some result that cannot be obtained by the elements operating individually [77]. These elements of a system may themselves be large and complex, and comprised of sub-elements acting in concert. The term system of systems designates the case where the constituent elements are collaborating systems that exhibit the properties of operational independence (each constituent system operates to achieve a useful purpose independent of its participation in the system of systems) and managerial independence (each constituent system is managed and evolved, at least in part, to achieve its own goals rather than the system of systems goals) [110]. There is consensus among researchers [22, 113] and practitioners [125] that these properties necessitate treating a system of systems as something different from a large, complex system. While fields such as enterprise architecture and service-oriented architecture address systems that include the distinguishing characteristics noted above, "systems of systems" is treated as a distinct field by many researchers and practitioners, with its own conferences (e.g., IEEE International Conference on System of Systems Engineering) and journals (e.g., International Journal of System of Systems Engineering).

Architecture plays a vital role in a system's ability to meet stakeholders' business and mission goals [10], hence we decided to perform a Systematic Review [88] of the published literature to characterize the state of research on system-of-systems architecture. We define the architecture of a system as the set of structures needed to reason about the system, which comprise elements, relations among them, and properties of both [10]. In the context of a system of systems, some structures may be comprised of elements and relations that are purely physical. For example, in structures where the elements are radar systems, the relationship is their arrangement and orientation to detect targets in a particular geographic area, and a property is the transmission frequency of

each radar so as to avoid electronic interference. While such physical structures are obviously important to achieving business and mission goals, we confined our review to research in the information system aspects of system-of-systems architectures.

The specific research questions that motivated our study are:

1. What research has been published on the subject of system-of-systems architecture?
2. What is the impact of these studies to the research and practice of system-of-systems architecture?

Previous literature surveys on systems of systems have focused on the definition and distinguishing characteristics of systems of systems [22, 63]. Our research has different goals, as noted above, and we have used a systematic and rigorous approach to identifying and selecting the reviewed primary studies. Our study performed a systematic search for publications in multiple data sources and followed a pre-defined protocol for study selection and data extraction.

This chapter is organized as follows: Section 2 of this chapter discusses the research method used for the study. Section 3 presents and discusses the results of the review. Section 4 discusses threats to validity of these results. Section 5 presents conclusions and identifies opportunities for additional research.

2.2 Research Method

As noted above, this study was conducted according to the systematic review methodology described by Kitchenham and Charters [88], following all steps and guidelines. There are many recent publications that describe the methodology, mechanics, and advantages and limitations of systematic reviews, so we discuss here only those aspects of the methodology relevant to the results reported here.

The specific process that we followed to create the set of primary sources is shown in Fig. 2.1 and is described in the following sections.

2.2.1 Search Strategy and Data Sources

The search strings used in this review were constructed using the following strategy:

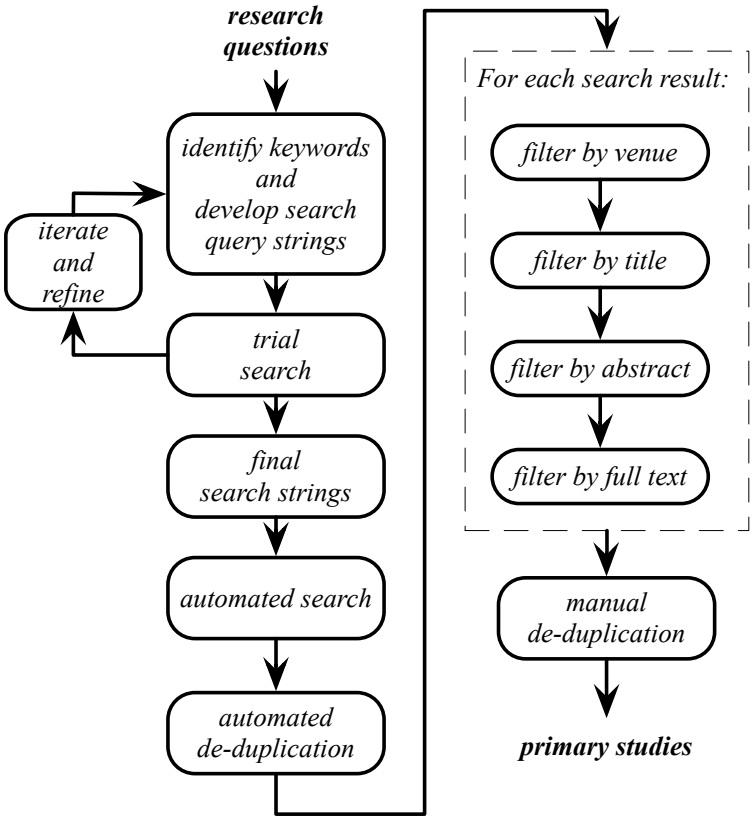


Figure 2.1: Primary studies selection process

- Identify the main terms based on the research questions and topics being researched;
- Determine and add synonyms, related terms, and alternative spellings as appropriate, and incorporate into the string using Boolean "or";
- Link the main terms using Boolean "and";
- Pilot the search string and iterate the steps until sufficient sensitivity [164] was achieved, using a standard constructed of selected references from a recent textbook [81] surveying the field of systems of systems.

We took the position that systems of systems are an independent field of research and practice (as discussed above), and focused on studies that were explicitly in that field.

Our main terms were "system of systems" and "architecture".

We considered several synonyms for "system of systems". "Service-oriented architecture" is concerned with interoperation between independent systems, but was rejected as too narrow in scope since it represents just one possible architecture style for system integration.

We also considered "enterprise architecture", but rejected it as being too broad. An enterprise architecture is usually realized as a set of interoperating systems; however, these systems may or may not satisfy the criteria of independence of purpose and evolution that characterize a system of systems. Furthermore, in practice, an enterprise architecture frequently manifests as principles and governance processes, rather than the structures (elements and relationships) that comprise a system-of-systems architecture or a system architecture [103].

We added alternate spellings, and extended the keywords based on pilot search results.

This strategy produced the following ideal search string:

```
("system of systems" OR "systems of systems" OR
 "system-of-systems" OR "systems-of-systems") AND
 (architecture OR design OR implementation OR model
 OR interoperability OR interoperation)
```

Variabilities in the search features provided by the digital literature collections (e.g., ACM Digital Library, IEEEXplore, and Elsevier Science Direct) required adapting the ideal search string to the capabilities of each particular search engine, as has been done in other recent systematic reviews (e.g., [33]). We made every effort to ensure that the adapted search strings were logically and semantically equivalent. Both authors were involved in this phase of the review, refining and reviewing the capability of the adapted search strings.

The digital sources searched for this review were:

- IEEExplore (`http://www.ieeexplore.org`)—IEEE publications only
- ACM Digital Library (`http://dl.acm.org`)—ACM publications only
- Compendex (`http://engineeringvillage2.org`)
- Inspec (`http://engineeringvillage2.org`)
- Wiley (`http://onlinelibrary.wiley.com`)
- Elsevier Science Direct (`http://sciencedirect.com`)
- Inderscience (`http://inderscience.com`)
- SpringerLink (`http://springerlink.com`)
- IOS Information, Knowledge, Systems Management (`http://iospress.metapress.com/content/105654/`)

As noted above, the sensitivity (or inclusiveness) of the search was checked using a standard of 22 primary studies. These studies were selected from the references cited in Jamshidi's recent textbook [81], based on their relevance to this study's research questions.

The search results used for this study were generated on 12 July 2012, and covered published results up to that date.

2.2.2 Search Results

The search strategy used for this study resulted in 1,865 initial candidate papers. Semantic ambiguity of several of the main search terms resulted in a strategy that traded off higher sensitivity against lower precision [164]. For example, the terms "system of systems" and "architecture" are often used in ways that do not relate to the subject of this systematic review, but we were

unable to automatically detect these semantic differences. Also, some of the search engines included full text of the paper in the search, so any paper citing a reference published in the IEEE System of Systems Engineering Conference was selected by this search strategy.

We performed an automated de-duplication based on first author and title, which reduced the set to 1,617 papers. Next, one researcher looked at publication venue, title, and abstract, and applied the following selection criteria:

1. Include only papers from peer-reviewed journals, conferences, and workshops.

2. Include only papers written in English, with full text available.

3. Exclude papers that do not explicitly or implicitly define "system of systems" as an assemblage of constituent systems that are independently operated and independently managed (i.e. as defined by Maier [110]).

4. Exclude papers that focus on defining the term "system of systems," or on the general implications of a particular definition (unless those implications explicitly address architecture). There is a rich body of publications that focus on defining and distinguishing a system of systems, and several previously published surveys collect and analyze this body of literature [22, 63]. In the case where a paper discussed definitions and also explicitly addressed architecture implications of a particular definition (e.g., the heuristics that Maier presents [110]), we included the paper.

5. Exclude papers that address architecture concerns unrelated to information systems (e.g., papers focused only on physical architecture structures, or modeling social or political systems as a system of systems.).

6. Exclude papers that are primarily concerned with general distributed system issues, e.g., agent-based coordination, service-oriented architecture and web services, or system interoperability. From an information systems architecture perspective, systems of systems are almost always distributed systems, comprising multiple autonomous computers executing software that communicates through a computer network. The converse is not true—many distributed systems are not systems of systems, because the constituent systems do not exhibit operational and

managerial independence. Many papers addressing issues related to distributed systems (e.g., coordination or interoperation) make reference to the system-of-systems context and were selected by the automated search. We excluded papers that discussed general distributed system issues, unless the paper focused specifically on those issues within the system-of-systems context.

7. Exclude papers that deal with domain-specific algorithms, not generalizable architecture approaches (e.g., reformulating a particular algorithm to operate in a particular system-of-systems architecture).

8. Exclude papers that deal primarily with system-of-systems requirements, acquisition, test, integration, or certification processes, unless there is also discussion of how architecture impacts those other lifecycle processes.

9. Exclude papers that deal with general system architecture concerns or approaches, with only reference to scaling up to a system-of-systems context.

Applying these inclusion and exclusion criteria looking only at the publication venue, title, and abstract resulted in 234 papers. At this point, additional filtering was performed to remove papers that presented the same results. Where we found the same authors publishing several papers that presented similar results, we retained only the most recent or most comprehensive presentation. This excluded an additional 34 papers, resulting in 200 papers.

A single researcher performed the initial exclusion screening, so a "test-retest" protocol was used to verify the exclusion decisions [88]. At the conclusion of the initial exclusion process, 50 of the excluded papers were selected at random and re-evaluated. None of the re-evaluated papers was incorrectly excluded.

Two researchers then read the full text of each of the 200 papers during the data extraction process. During this step, an additional 6 papers were found to violate one of the inclusion/exclusion criteria, and so were excluded, leaving 194 primary studies in the review. The full list of primary studies is available at `http://www.andrew.cmu.edu/user/jklein2/primary-sources.pdf`.

2.2.3 Data Extraction and Synthesis

We read the full text of each of the 194 primary studies, and used a predefined spreadsheet form to extract and store the following data related to the research questions:

- Type of research result reported by the study, categorized using Shaw's scheme [143];
- The architecture task(s) that were the focus of the primary study, based on the categorization developed by Bass and colleagues [11];
- The system application domain (if any) that was the focus of the primary study or was used in any examples presented in the study. This was an emergent classification, with no pre-defined categories.
- The quality attributes (if any) that were the focus of the primary study. This was also an emergent classification, with no pre-defined categories. We limited the data extraction to no more than 3 discrete quality attributes for any of the studies.
- The technology maturity level indicated by the results presented in the primary study, using the classification scheme of Redwine and Riddle [133].

Each researcher initially performed data extraction independently on a set of 20 studies, and these results were discussed in detail. This led to the creation of decision trees for the Result Type and Technology Maturity data extraction, to assist the two researchers in making more consistent classification. The researchers then independently performed data extraction for the entire set of 194 primary studies. Results were compared, and disagreements were discussed and resolved.

2.3 Results and Discussion

2.3.1 Demographic Data

The 194 primary studies were published in 95 different venues. However, just 12 venues account for 104 of the published papers, as shown in Table 2.1, with each of the other 83 venues having just 1 or 2 published papers. Most of the primary studies were published in conference proceedings (143), with 35 published in journals and 16 published in workshop proceedings.

Table 2.1: Most frequent publication venues

Venue	**# of studies**
Int. Conf. on System of Systems Eng. (SoSE)	27
IEEE Systems Conf.	13
Systems Engineering	10
IEEE Aerospace Conf.	9
Int. Conf. on Systems, Man and Cybernetics (SMC)	11
IEEE Military Communications Conf. (MILCOM)	8
IEEE Systems Journal	8
Winter Simulation Conf. (WSC)	5
Proc. SPIE	4
Digital Avionics Systems Conf.	3
Information, Knowledge, Systems Management	3
MTS/IEEE Biloxi Conference (OCEANS)	3
Subtotal	**104**

The year of publication is shown in Table 2.2. Our systematic review reveals that there was a sharp increase in the number of publications in 2008, and the number of publications has held steady each year since then.

The first International Conference on System of Systems Engineering (SoSE) was held in 2005. Although no papers from that conference were selected as primary studies for this systematic review, the creation of a new conference indicates that the community of researchers in the field had reached a tipping point and was sufficient to support a stand-alone event. Another milestone event occurred in 2008, when the United States Department of Defense (US DOD) published its System Engineering Guide for Systems of Systems, clearly

Table 2.2: Number of primary studies published each year

Year	#
1993	2
1994	2
1995	0
1996	0
1997	0
1998	3
1999	0
2000	2
2001	4
2002	3
2003	4
2004	5
2006	12
2007	10
2008	13
2009	28
2010	34
2011	30
2012	13*

*This study included papers published through 12 July 2012, approximately $\frac{1}{2}$ of the year 2012.

distinguishing and highlighting the significance of systems of systems. Also in 2008, the US DOD funded and launched the Systems Engineering Research Center. This focus of attention and infusion of funding may have contributed to the higher publication rates since 2008.

2.3.2 Type of Research Result Reported

The research results reported in each primary study were classified by selecting one of the categories defined by Shaw [143]. Shaw's categorization was originally created to explain how software engineering research strategies—questions, type of results, and validations—shift as the field matures. We apply the categorization to understand where researchers are focusing their effort.

We used the following decision tree to perform the classification: The *specific solution* category included studies that presented an architecture in some detail, with no generalization of the results as method, model, or notation. Taxonomies, frameworks, or well-argued generalizations were classified as *a qualitative or descriptive model*. Studies presenting a repeatable way to perform an architecture task were classified as *procedure or technique*. A model supporting formal or automatic analysis or code generation was classified as an *analytic model*; a graphical or textual notation for an analytic model, or a tool supporting such a notation, was classified as *notation or tool*. The *report* category included experience reports and guidelines for applying a technique or procedure. Benchmarks and trade studies were classified as *answer or judgment*. Finally, predictive models based on observed data were classified as *empirical model*. In cases where a study reported multiple result types, we selected a single one that was the focus of the paper.

Although these categories appear to be distinct, we found some difficulty in applying this scheme to the primary studies in this systematic review. We observed that researchers publishing their results tended to clearly define the result type in a paper's abstract, and tended to organize a paper around a single result type. On the other hand, practitioners were often less precise in defining the type of result reported in a paper, and published papers covering several result types. We did not attempt to classify authors as "researcher" or "practitioner" (this would have to be based on the affiliation reported in the publication, which itself does not necessarily distinguish researchers from practitioners, and may not be representative of the organization that the author belonged to when the reported work was conducted). In the initial data extraction from practitioner studies, there was frequent disagreement between the authors, which was resolved by independently re-reading the study and discussing the categorization until agreement was reached.

As shown in Table 2.3, the most frequently reported result type was a *procedure or technique* (61 studies). These procedures were presented at multiple task levels, ranging from specific single tasks in the system development process like analyzing a particular quality attribute, up to comprehensive approaches for performing architecture design and evaluation for a system of systems. We found that most of these procedures seemed to be created in isolation—there was almost no reference to, or building on, the work of others.

Table 2.3: Type of research result reported

Result Type	**#**
Procedure or technique	61
Qualitative or descriptive model	43
Specific solution	36
Analytic model	29
Notation or tool	11
Report	10
Answer or judgment	4
Empirical model	0

The second most frequently reported result type was a *qualitative or descriptive model* (43 studies). Many of these models were aimed at dealing with the scale and complexity of the system-of-systems architecture context. Examples include informal taxonomies of element types and identification of significant element properties related to a particular type of analysis. Applying the ISO-42010 metamodel [79], many of these qualitative or descriptive models were framed in terms of stakeholder concerns, and partially define a viewpoint.

Specific solutions were frequently published (36 studies). These primary studies presented the architecture of a particular system of systems, usually with some accompanying discussion of drivers, notable quality attribute achievements, or particular challenges. While many primary studies presented a system-of-system architecture to demonstrate or validate another result type, we applied this category to studies where the primary focus was presentation of a completed architecture, with only incidental discussion of other types of results.

There were a number of *analytic models* published (29 studies), focused mostly on automatic manipulation of the architecture related to a particular quality attribute. We applied this category to papers that reported both an analytic model and an accompanying tool.

Finally, there were a smaller number of results in the *notation or tool* category (11 studies), *report* category (10 studies), and *answer or judgment* category (4 studies). Our systematic review found no papers reporting an *empirical model*.

2.3.3 Architecture Task Focus

We grouped the architecture tasks defined by Bass and colleagues [11] into a smaller number of composite categories, reflecting that in practice, the tasks of design, analysis, modeling, and documentation are often performed concurrently. Each primary study was classified into a single category, as shown in Table 2.4.

Table 2.4: Architecture task focus

Architecture Task	**#**
Design/Analysis/Modeling/Documentation	137
Model-driven architecture	19
Evaluation/Analysis	12
All tasks (not model-driven architecture)	18
No architecture task	5
Test (design for test, testability analysis of an architecture)	3

Most of the primary studies focused on the design/analysis/modeling/-documentation tasks that are core to creating an architecture. This category included primary studies that reported specific solution results (as described in the previous section) in which the authors discussed issues related to these architecture tasks.

There were 19 primary studies that presented results related to model-driven architecture (MDA) methods. The models created in applying MDA methods are used in many architecture tasks, but we chose to distinguish MDA results from other studies that focused on all architecture tasks, but did not apply an MDA approach. There were 18 studies in this category.

Architecture evaluation and the analysis related to evaluation were the focus of 12 primary studies.

A small number of studies (3) focused specifically on architecture tasks related to test and integration, including architecture design for testability and analysis of testability.

Finally, there were five studies where no focus on an architecture task could be identified. These studies presented specific solution results, with no accompanying discussion that contributed to architecture knowledge for any of the architecture task categories.

2.3.4 Application Domain

Many of the primary studies framed their results in the context of a particular application domain. These are summarized in Table 2.5.

There were 74 primary studies that did not frame their results in a particular application domain. Of those that did discuss a particular application domain, the most frequently discussed was defense and national security (58 studies). The Global Earth Observation System of Systems was discussed by 20 of the primary studies. The remaining 42 primary studies discussed a variety of other application domains.

Table 2.5: Application domain

Application Domain	**#**
No specific application domain	74
Defense and national security	58
Earth observation system	20
Space system	8
Modeling and simulation	6
Sensor network	5
Healthcare, electric power grid	4
Business information system	3
Transportation system	3
Astronomy	2
Cloud computing, crisis management system, enterprise architecture, home automation, human tracking, SCADA, or social computing	1 each

2.3.5 Quality Attribute Focus

Architecture allows us to reason about a system's ability to satisfy both functional and quality requirements. There were 128 primary studies that focused on one or more quality attributes.

Quality attributes are notoriously difficult to define when only the name is given [9], and so we did not attempt to infer an author's meaning. We performed an emergent categorization, extracting the specific terminology used in each study, recognizing that there may be some overlaps in this categorization. For example, an author referring to Quality of Service (QoS), may have intended to include availability, performance, and other qualities under that label.. We extracted up to three quality attributes from each study (there were nine studies that focused on more than three qualities). The results are shown in Table 2.6.

The most frequently discussed quality attribute was interoperability (45 studies). Since a system of systems is a collection of collaborating systems, interoperability is a necessary concern. Evolution was also frequently addressed (13 studies), and since a system of systems combines existing independent systems, evolution is a natural concern. Security and safety were also frequently discussed (14 and 8 studies, respectively), and these concerns often arise in the defense and national security application domain, which was the most frequently discussed application domain.

2.3.6 Technology Maturity

The result reported by each of the primary studies was classified using Redwine and Riddle's technology maturation model [133]. This model traces the evolution of software technology from initial concept definition through six phases that culminate in popularization, as demonstrated by production quality versions of the technology and broad commercialization. The classification results are shown in Table 2.7.

There were no basic research results, and a small number of concept development results. These may be attributed to our decision to exclude studies that focused only on defining and distinguishing the basic concepts related to systems of systems.

Table 2.6: Quality attribute focus

Quality Attribute Discussed	**#***
No specific quality attributes discussed	66
Interoperability	45
Security	14
Evolution	13
Performance	9
Safety	8
Testability	6
QoS, reusability, risk	5
Adaptability, complexity, correctness, coupling, flexibility, reliability	3
Availability, compliance, composability, cost, efficiency	2
Assurance, consistency, dependability, feasibility, manageability, monitorability, privacy, reconfigurability, robustness, self-healing, self-configuration, supportability, survivability	1
More than three qualities discussed	9

*Multiple classification allowed—up to three distinct quality attributes per study

A majority of the results were in the development and extension category (124 studies), reflecting results that have not yet been applied to develop a system. A prototypical example of a study in this category is the work of Dimarogonas [46], which reports on a set of design tenets and rules for architecture development, with no evidence presented that the reported approach was applied in the design of a system-of-systems architecture. Also, the framework reported in this study was independently developed, not building on any previous system-of-systems architecture research.

There was some internal development and extension (30 studies), mostly reflecting authors' extensions of their own previous work. Finally, 35 studies were classified as external extension and development, explicitly applying and extending the work of other researchers. Li and Yang's study [106] is a prototypical example of this category, reporting a system-of-systems architecture design process that extends system architecture, system-of-systems architecture, and software product line research and technology.

Redwine and Riddle originally noted that the maturation process takes 15-20 years, based on data through the mid-1980s. Shaw performed a similar analysis, confirming the maturation timeline using data through the 1990s [142]. Many researchers cite Maier's 1998 publication [110] as marking the start of the system-of-systems technology development, which puts the field approximately 14 years into the maturation process at the time of our systematic review. Compared to other software technologies, system-of-systems architecture technology appears to be maturing relatively slowly.

Table 2.7: Technology maturity phase

Maturity Phase	**#**
Basic research	0
Concept formulation	5
Development and extension	124
Internal enhancement and exploration	30
External enhancement and exploration	35
Popularization	0

2.3.7 Impacts on research and practice

This systematic review has a number of implications for research and practice.

Relationship to Adjacent and Overlapping Fields

Among the primary studies that were framed in a particular application domain, the most frequently discussed domains deal with systems that are typically government-funded and government-acquired (defense and national defense, earth observation system, space system), and we see very little reference to the term "systems of systems" in other domains. We also note that most of the primary studies were published in venues focused on a specific application domain, rather than in venues focused on more general software engineering or information systems. This could imply that systems of systems appear more frequently in certain application domains, or that the designation of a large, complex system as a "system of systems" provides a benefit only in certain application domains, and these types of systems are simply not distinguished in other application domains. For example, in many large

corporations, the information technology infrastructure satisfies the definition of system of systems (operational and managerial independence of the constituent systems, which are managed and operated by different business units and functional units), but this type of system is not typically referred to as a "system of systems".

Additional research is needed to determine if the information system aspect of system-of-systems architecture constitutes a distinct field of research and practice, or if other fields such as distributed systems, service-oriented architecture and interoperating systems, and enterprise architecture already cover it.

Relationship to Industry Platforms and Software Ecosystems

Maier classified systems of systems into three categories, based on the type of managerial control [110]:

- Directed: The system of systems is centrally managed. Constituent systems are built primarily to fulfill system of systems purposes, with independent operation as a secondary goal (for example, stand-alone system operation may provide degraded services during a system of systems failure).
- Collaborative: The system of systems has central management, but it lacks authority over the constituent systems. Constituent systems voluntarily choose to collaborate to fulfill the system of systems purposes. Maier gives the example of the Internet as a collaborative system of systems, with the IETF setting standards but having no enforcement authority. Participants choose to comply with the standards if they want to be part of the Internet system of systems.
- Virtual: The system of systems has no central management and no centrally agreed-upon purpose. Maier's example here is the World Wide Web, where there is no central governance. There are incentives for cooperation and compliance to core standards, which emerge and evolve based on market forces.

Collaborative and virtual systems of systems are related to industry platforms and software ecosystems. An industry platform provides the core technology that allows systems constructed by different organizations to interact to produce some value [39]. In both collaborative and virtual systems of

systems, an industry platform can broker interactions between participating systems and provide incentives to join the system of systems and to behave in particular ways in the system of systems. The relationships among the systems using an industry platform and among the organizations constructing those systems create an ecosystem with cooperation and competition among participants [117].

Our systematic review uncovered little research in industry platforms as part of system-of-system architectures, or in links between systems of systems and software ecosystems. Research in this area is needed to address how an industry platform for a system of systems is scoped and defined, addressing issues such as which features and variabilities might be included in a platform, what architecture approaches (e.g., patterns, tactics, and heuristics) are useful in this design context, and how to assess the cost and value of these design alternatives in order to make design decisions.

Pace of Maturation

Our classification using the Redwine and Riddle model of technology maturation shows that a majority of the studies fall into a middle maturity stage of development and extension. As noted above, system-of-systems architecture technology is maturing at a relatively slow rate, compared to other software engineering fields. This is supported by our finding that the majority of the studies reported results that were procedure or technique, and we see many studies reporting results of researchers and practitioners working in apparent isolation to create new approaches to solve system-of-systems architecture problems, with no particular approaches gaining widespread adoption.

This leads to a set of questions that impacts both research and practice:

- What types of architecture knowledge are needed to design, analyze, evaluate, and evolve system-of-system architectures? What design and organizational pattern, tactics, and heuristics apply, given a particular technical problem and technical and non-technical constraints? How should this knowledge be organized to support the tasks and workflows used in working on system-of-systems architectures?

- Are there general methods for designing and evolving system-of-systems architectures, or does the scale, complexity, and non-technical constraints of each system of systems require a unique solution approach?

- Are there general methods for analyzing and evaluating system-of-systems architectures? Given the scale and complexity of a system of systems, how is the coverage or completeness of an analysis or evaluation method determined?

Finally, most of the primary studies were published in conference venues, with just 35 studies published in journals. This is consistent with our finding of a relatively low level of technology maturity, since journal publication is usually indicative of more mature research results. Furthermore, most of the primary studies were published in domain-oriented venues, with no de facto home for research in general system-of-systems architecture technology. Notably, few of the primary studies were published in leading software engineering or software architecture venues. This calls for the creation of a venue to nurture and disseminate research about the questions identified above.

2.3.8 Study Limitations and Threats to Validity

This study is limited to reviewing studies reporting research results about system-of-systems architecture, published in peer-reviewed venues through 12 July 2012. We did not include any gray literature (technical reports, white papers, web blog postings, etc.).

The main threats to the validity of this research are bias in the selection of studies to include, and bias in the data extraction.

Selection bias was controlled by developing a research protocol based on the research questions. The research protocol included a search strategy and inclusion/exclusion criteria. The research protocol was developed by the first author, and reviewed by the second author to ensure correct formulation of the research questions and whether the search strategy and the inclusion/exclusion criteria followed from the research questions. The first author is an experienced consultant in the field of systems of systems, while the second author is an academic experienced in the conduct of systematic reviews.

As noted in Section 2.2.2 above, the semantic ambiguity in the primary search terms "system of systems" and "architecture" produced a large number of automated search results. The automated search results were manually filtered using a multi-step process, as established a priori in the research protocol. The first author performed most of the manual filtering. A "test-retest" protocol [88] was used to check for any bias in the manual filtering—50 papers were randomly selected from the set of excluded papers and rechecked to ensure that the inclusion/exclusion criteria were consistently applied.

Bias in data extraction was controlled by establishing a research protocol based on the research questions. Both authors independently performed the data extraction on all primary studies, using the same spreadsheet form. Categories for research result type and technology maturity phase were established before the data collection started, and were supported by a decision tree to guide classification decisions. Architecture task classification was initially attempted using a fine-grained categorization, but was repeated using the categorization discussed in Section 2.3.3 above. In all cases, where the authors' independent data extraction results disagreed, there was an independent reassessment by each author and a discussion of the updated result.

2.4 Conclusions

A system of systems is a design context where scale, complexity, and certain non-technical constraints necessitate the use of architecture methods and approaches that are different from those used for system architectures. This chapter reports the results of a systematic review of the research in the information system aspects of system-of-systems architecture. We found that this field is maturing more slowly than other software engineering fields, and there is a need for additional research to understand and address this slow maturation.

We found that the primary studies were published in a large number of diverse and mostly domain-oriented venues, and conclude that a publication venue focused on system-of-system architecture could contribute to the formation of a research community of practice. The SHARK workshops in the field of architecture knowledge are one such successful example.

Our systematic review found that most reported research reflects individuals and teams working in apparent isolation to develop techniques to solve particular system-of-system architecture problems. Research is needed to develop more general procedures and techniques for design, analysis, evaluation, and evolution of system-of-systems architectures. The discussion above outlines several specific research opportunities, but the creation of a cohesive research agenda for the field is needed.

Finally, we noted above that there are domains, such as corporate information technology infrastructures, which are creating systems-of-systems architectures but not using the "system of systems" label or directly using system-of-systems technology. Further research is needed to understand how fields such as enterprise architecture relate to system-of-systems architecture.

3

Common Software Platforms in System-of-Systems Architectures: The State of the Practice

Summary

System-of-systems (SoS) architectures based on common software platforms have been commercially successful. Common platforms are a goal of several DoD initiatives (US Army Common Operating Environment, US Navy Open Architecture, multi-service Future Avionics Capability Environment), but progress on creating and adopting such platforms has been slow. We conducted a study to understand the technical issues related to SoS common platform development and adoption, and the non-technical constraints that must be satisfied. We interviewed 12 experts, collecting and analyzing mostly qualitative data. Although there were significant differences in approaches between developers of commercial SoS platforms, military SoS platforms, and command and control SoS, all reported that non-technical constraints dominate intrinsic technical issues. We recommend further research to create systematic architecture design and analysis methods for SoS, to study agile development methods in the SoS context, and to develop approaches for documentation of constituent systems within an SoS.

3.1 Introduction

3.1.1 System-of-Systems Context

A system is a collection of elements that together produce some result that cannot be obtained by the elements operating individually [77]. The elements of a system may themselves be large and complex, and comprised of sub-elements. The term *system of systems* (SoS) designates the case where the constituent elements of a system are collaborating systems that exhibit two properties: (1) operational independence (each constituent system operates to achieve a useful purpose independent of its participation in the SoS); and (2) managerial independence (each constituent system is managed and evolved, at least in part, to achieve its own goals rather than the SoS goals) [110]. There is a consensus among researchers [63, 113] and practitioners [125] that these properties necessitate developing and operating an SoS in different ways than a large, complex (single) system.

In the SoS context, interoperability among the constituent systems is a primary architecture concern [94]. An SoS platform that provides services and functions to all constituent systems within an SoS is one strategy to promote interoperability [90].

3.1.2 Platforms, Product Platforms, & System-of-Systems Platforms

The term "platform" is used to refer to several different concepts. In the military domain, a platform is a vehicle (ship, aircraft, tank, etc.) that transports systems and provides physical services such as power and cooling [66]. In other domains, "platform" is used to refer to the common elements reused across a product line or product family. Cusumano calls this a "product platform" [39], while Madni uses the term "platform-based engineering" [109]. In both cases, the primary concern of this type of platform is reuse of hardware, software, and related assets.

A third use of the term "platform" is what Cusumano refers to as an "industry platform" and we call a "system-of-systems platform" [90]. This type of platform provides services to an open set of systems that interact to form an SoS. The services provided can be general-purpose, such as directory and authentication services, or domain-specific, such as geospatial information

processing for a command and control SoS. The primary concerns of an SoS platform are: (1) Support interoperation among the systems using the platform; (2) reduce the cost and time needed to develop or modify systems for use in the SoS; and (3) enable modular substitution of constituent systems in the SoS.

These three concerns are related to each other. First, an SoS platform supports interoperation by providing common information models (semantic interoperation), and common communication mechanisms such as those provided by frameworks and middleware (syntactic interoperation). The platform may also prescribe patterns or sequences of interaction for certain system-of-system functions.

Second, the SoS platform also provides implementations of services needed by constituent systems. As services are relocated within the SoS architecture from constituent systems into the SoS platform, system-to-system dependencies are replaced by system-to-platform dependencies, which typically reduces the time and effort required to develop, integrate, and test systems to create the SoS. The availability of an SoS platform also reduces the barrier to entry for an organization to create a new or replacement system, since less effort and expertise are needed, and the risk is lower.

Finally, the ability to substitute one implementation of a system for a different implementation is necessary to create an *ecosystem*, where organizations "have a strategy to open their technology to complementors and create economic incentives (such as free or low licensing fees, or financial subsidies) for other firms to join the same 'ecosystem' and adopt the platform technology as their own" [39]. The reduced barrier to entry described above contributes to the incentives to join or participate in the ecosystem.

Examples of successful commercial ecosystem-enabling SoS platforms include Facebook, Apple (iOS, OS X, iCloud, App Store, etc.), and Salesforce.com. In other domains, Future Avionics Capability Environment (FACE) and the US Army Common Operating Environment (COE) are examples of emerging SoS platforms, with their eventual success yet to be determined.

3.1.3 Goals of this Study

As noted above, there are commercially successful SoS platforms, but success in military systems and other domains has been elusive. As part of a multi-year research project, we are developing systematic approaches to support SoS platform development. Our research began with this exploratory study of the state of the practice of SoS architecture development, with a focus on architectures that include SoS platforms. Our goals for this study included answering the following questions:

1. What processes are used to develop SoS architectures, and how are software elements of the architecture treated in the processes used?

2. What challenges do SoS programs face in developing architectures; performing test, integration, and assurance; managing runtime configuration and operation; and evolving the SoS? What approaches have been used in successful programs to overcome these challenges?

3. What are the constraints on new approaches to developing, using, and evolving these SoS architectures?

4. What are the important differences between practices used to create commercial SoS architectures and military SoS architectures?

The following section presents our research approach, including our interview protocol and participant demographics. We next present the results of our interviews, followed by analysis and discussion of the results. We conclude by identifying additional research needed to address the issues raised in this study.

3.2 Research Method

Our first attempt to answer the questions outlined above was to convene a workshop, bringing invited participants together to answer these questions in a group setting. Participants were recruited from the professional networks of the research team members. Our inclusion criterion for participation was direct experience as an architect or systems engineering leader on the development of at least one SoS. Each invitee was also requested to forward the invitation to other appropriately qualified members of his network.

Only one invitee agreed to participate in the workshop/focus group, and several responses implied a reluctance to share relevant experience in a group setting. This led us to develop an interview protocol that reported responses anonymously. The recruitment process was repeated, yielding 14 qualified participants. Two of these participants later withdrew from the study, leaving the 12 participant interviews that are reported here.

Study participants had between 10 and 25 years of professional experience. Two of the participants had experience working on one SoS project, and eight participants had experience working on four or more SoS projects. Table 3.1 describes the types of organizations represented by the participants.

Table 3.1: Organization Types Represented

Organization Type	**# of Participants**
Commercial software development (non-military)	4
Military system development - Industry	5
Military system development - Government	3

For each interview, one researcher acted as the lead interviewer and at least one other researcher participated. All interviewers recorded responses that were later combined into a single interview record. We prepared a script to guide the lead interviewer and act as a checklist to ensure that all topics were covered, as an interviewee's response to one question often covered several of our topics. Nine of the interviews were conducted with the lead interviewer meeting the interviewee in-person and other interviewers participating in the interview by telephone, and the other three interviews were conducted solely by telephone. All of the researchers have experience conducting interviews as part of exploratory research.

3.3 Interview Questions

Each interview began with the lead interviewer reading a prepared statement that stated that all reported results would be anonymized to protect the privacy of the interviewee and their organization, and that the interviewee should not disclose any protected proprietary information. We then collected the

demographic information presented above. Interview questions were divided into three sections, corresponding to the study goals outlined above. The complete interview instrument is available online at `http://www.andrew.cmu.edu/user/jklein2/SoS-Study-Interview-Questions.pdf`.

The first set of questions focused on the processes used to develop SoS architectures. The questions in this set allowed us to understand how the participant defined the term "system of systems", and their general approach to the architecture process. Maier identified conflict between the classical systems engineering "is part of" decomposition hierarchy and the layered software approach based on the "is used by" relation [111], and so we asked questions to understand how the participant addressed this conflict. Finally, constituent systems in an SoS are independently developed and evolved, and so we asked questions to understand how architecture trade offs are framed and how decisions are made, balancing the concerns of the SoS with the concerns of each constituent system. The objective of this set of questions was to understand the scope of activities and concerns of SoS architecture, and to understand how software concerns interact with other SoS concerns.

The second set focused on challenges in various system lifecycle phases, and on how successful projects addressed those challenges. We focused on lifecycle activities that are related to architecture: development of constituent systems for the SoS; test, integration, and assurance of the SoS; runtime configuration and management of the SoS; and sustainment and evolution of the SoS. For each of these activities, we asked participants to discuss technical and non-technical challenges, and to provide examples of projects that successfully addressed the challenges. The objective of this set of questions was to identify specific gaps in current practice where new methods would have the greatest impact, and to identify specific solutions employed by the interviewees that would be candidates for generalization.

The final set of questions focused on the constraints that a solution (e.g., a new method for design or analysis) must satisfy. We focused on the same four activity areas used in the previous question set. The objective of this set of questions was to identify factors necessary for any new approach to be successfully translated from research into practice.

3.4 Results and Discussion

Participants' responses broadly separated into three groups, based on the their group context, experience, and responsibilities. These groups are (1) commercial SoS platforms, (2) command and control SoS, and (3) military SoS platforms. In the discussion that follows, we organize our findings using these groups.

3.4.1 Architecture Framing and Processes

Participants framed and defined the SoS platform in markedly different ways. Command and control SoS architects and military SoS platform architects described the platform in terms of "what it is". They focused on technology characteristics, such as APIs and programming language bindings, and on the services provided by the platform. Commercial platform developers, on the other hand, framed the platform in terms of "what it does". They focused on the platform's ability to create network effects that support an ecosystem. Military SoS platform architects also recognized the importance of an ecosystem to the success of the platform, but they were less focused on the role of the platform in enabling the ecosystem.

None of the participants reported the use of particular methods or approaches for SoS architecture development or analysis or for SoS platform definition.

Most participants identified two development scenarios: Creation of a new SoS comprising primarily new constituent systems, and integration of existing systems to create an SoS. The first scenario applies primarily to directed systems of systems [110], where constituent systems goals and governance are aligned well with those of the SoS. In this scenario, architecture design can begin either top-down, based on requirements with a platform emerging as the design matures, or bottom-up, creating a platform first and then defining systems that use the platform. In the second scenario, architecture is much more constrained, and consistency or conceptual integrity across the SoS may not be achievable without substantial rework (and hence cost).

When making architecture decisions, no participant reported performing economic modeling of design alternatives. For commercial platform developers, time-to-market was a primary decision driver and after a viable solution was identified, they did little additional solution space exploration. On the other hand, architects involved with military systems reported that extensive

trade studies were performed, with architecture decisions frequently driven by development constraints. Software was not an early concern for them - it was initially treated like any other element of the SoS architecture, but as the architecture design matured, concerns such as maximizing software development efficiency, minimizing development cost, and meeting development schedules were high priorities that were balanced against overall SoS measures of performance.

Command and control SoS architects reported that downstream lifecycle costs and sustainment costs were less important than SoS operational performance. In contrast, both commercial and military platform architects reported that success depended on proper consideration of future needs. In the commercial organizations, these future needs were defined through market analysis, and in development of military platforms, future needs were informed by science and technology investment roadmaps.

All participants noted that deep domain knowledge was necessary to design a successful architecture. Domain knowledge enabled architects to identify the most important tradeoffs, eliminate ineffective parts of the solution space, and make timely decisions.

The command and control SoS architects reported that the way software architecture concerns are framed has changed over time. Earlier projects framed decisions only in terms of functional requirements, while more recent projects are framing decisions in terms of both functional and quality attribute requirements. Commercial SoS platform architects framed decisions in a context that included both functional and quality attribute requirements, and they did not explicitly distinguish between the two types of requirements.

3.4.2 Challenges and Patterns of Success

All participants reported that the primary challenges in developing and evolving SoS architectures are not rooted in technology, but are due to non-technical factors. These non-technical factors include misalignment of development organization and authority with the architecture, misalignment of system and SoS goals, reluctance to introduce dependence on the SoS platform into the constituent system architectures, and regulatory and policy constraints (for systems acquired by the United States government) that diminish the potential value of an SoS platform approach.

Another challenge reported across all projects is a challenge in migrating existing constituent systems in the SoS to use the platform: New platform features frequently duplicated existing features in the constituent systems. Modifying a constituent system to use the platform version of a feature incurs a short-term cost, but produces long-term value from reduced integration and sustainment costs. Commercial platforms use the value of modification as an incentive, whereas military organizations relied on top-down mandates.

These reported challenges are similar to challenges of developing, adopting, and sustaining software product lines [123]. Experience from software product lines and platform-based engineering provide insight into some of the challenges of platform-based systems of systems. Practices that are successful for single products need to change to achieve success in a product line context. Similarly, practices focused on developing single systems must change to be successful in the context of a platform-based SoS. Practices used for software product lines consider the relationships among development, organizational, and management concerns, and recognize that architecture and technology is just one contributor to overall product line success.

Software product line and platform-based engineering practices also promote the reuse of assets other than software, such as tools, plans, templates, test equipment, test cases, and personnel training and skills. Architects of military SoS platforms included assets such as documentation, training materials, and user community collaboration repositories as part of their SoS platform.

When developing or evolving systems to use the SoS platform, many participants reported challenges related to documentation of the constituent systems within the SoS. Although extensive architecture and design documentation may exist for a constituent system, it is often focused on the independent operation of the constituent system, and does not adequately address concerns related to the constituent system's operation in the SoS. Examples included resource scheduling approaches and handling of interface errors or exceptions.

The large scale and complexity of the SoS architecture context created several challenges in creating the initial instantiation of the SoS platform. Architects of the command and control SoS used the "V-model" [54] to develop their SoS. Significantly, the relatively long time between architecture definition and system integration allowed some architecture errors to remain undiscovered until late in the development process. On one project, they addressed this challenge by shifting to an iterative agile approach during later phases of the development cycle. This enabled faster feedback on the correctness of design

decisions, but there were unresolved questions that remained: Is it practical to use an iterative approach from the beginning of the project, or is there an initial base of functionality that should be in place before starting an iterative approach? What is the best way to plan iteration contents and duration?

The commercial SoS platform architects reported several approaches to creating and delivering the initial instantiation of the platform. From these, we have identified two "proto patterns" [157]. The first proto pattern is a sequence for evolving the architecture of a new platform. This begins by first defining and implementing atomic message types and message schemas, with no concept of workflow (i.e. sequences of messages related to a business task or process). Initially, all workflow is organically built into the systems and applications using the platform. Later, workflow orchestration is added to the platform, with the platform providing versioned workflow definitions that include endpoint roles (endpoint cardinality, supported message sets, and other workflows that the endpoint can participate in), workflow sequence definitions, and transaction support. This proto pattern allows an initial version of the platform to be deployed quickly, and then allows incremental definition of workflows based on actual platform use.

A second proto pattern is related to the evolution proto pattern described above. Workflow execution scalability and availability is achieved by maintaining workflow state only in the participating endpoints, not in the platform infrastructure. This "stateless platform" approach is a refinement of stateless services in service-oriented architectures.

Maintaining backward compatibility for systems using the SoS platform was reported as a challenge in architecture evolution. The commercial platform architects addressed this challenge through extensive test automation. Nearly all testing was automated, with one organization reporting that they have "tens of thousands" of automated tests, which allow them to maintain full compatibility back to systems developed for the first versions of the platform (the platform is now almost 10 years old and is updated three times per year). Commercial SoS platform architects also reported a proto pattern for deploying new platform features. This three-step pattern begins by piloting a new feature with selected customers, and special IT operations processes are used to carefully monitor usage and quality attributes such as performance. In a second release, the feature is stabilized with those customers, and IT

operations processes are similar to standard production processes. Finally, in a third release, the feature is generally available to all customers in production. (In the organization using this proto pattern, the time from pilot to general availability of a feature was 4-8 months.)

Command and control SoS architects reported similar issues maintaining compatibility as the architecture underwent evolution throughout the initial SoS development iterations. They also used a test automation strategy. This strategy used tests that covered both syntax and semantics of interfaces, and incorporated modeling and simulation systems into the test environment to extend test coverage beyond just platform interfaces.

3.4.3 Solution constraints

Participants identified two general constraints that must be satisfied by any new approaches or methods to address the challenges discussed in the previous section.

The first constraint is that any new approach or method should be integrated with existing tools, including tools used for architecture modeling, analysis, and documentation, and tools used for project/program management. The need for integration with architecture tools was expected - adoption of new approaches and methods is facilitated if there is no need for acquiring or learning new tools. The need for integration with project/program management tools is indicative of the strategic importance of architecture decisions, and the necessity of efficiently translating decisions about technical approaches into cost, schedule, and other metrics relevant to program executives.

The second constraint was that any new approach or method must align with assurance and certification processes. Much of the potential value of an SoS platform is reducing the cost and time to perform assurance and certification of the SoS, but this value can be accrued only if the features included in the platform, the analysis of the platform architecture, and documentation provided for the platform are aligned with the assurance and certification requirements of the SoS.

Finally, several participants involved in military SoS discussed specific constraints that their environment imposes on creating an ecosystem based on an SoS platform. These participants expect that such an ecosystem could reduce SoS acquisition costs, since the modular substitution of SoS elements should promote competition among suppliers of the elements. The ecosystem is also expected to increase innovation by enabling a broader community of

contributors to new and improved SoS capabilities. Creation of such an SoS platform-based ecosystem in this environment is currently constrained by United States government acquisition policies and by a limited ability to create effective incentives for both acquirers and suppliers to join or participate in the ecosystem.

3.5 Conclusions

This study interviewed 12 experts to characterize the state of the practice of system-of-system architecture development, with a focus on architectures that include SoS platforms. Our goal was to inform further research in systematic approaches to support the development and evolution of architectures for SoS platforms. This study identified several areas where additional research is needed.

The first area is selection of features for an SoS platform. The study identified a number of critical stakeholder concerns, including time-to-market, ease of adoption, support for future capabilities, and alignment with SoS assurance and certification processes. Feature selection requires consideration of both the problem space, to identify candidate platform features and assess their value, and the solution space, to assess costs to implement and maintain each feature. Systematic approaches to analyze the problem space might combine techniques such as mission thread analysis [85] with domain analysis [123]. Solution space analysis might include economic models and models that consider alignment of the architecture with constraints such as acquisition strategy, organizational structures, and other socio-technical factors. These analyses would be facilitated by catalogs of architecture knowledge, such as pattern handbooks, to provide a repertoire of solutions that exhibit particular functional and quality attribute properties. Finally, a systematic approach, such as economic modeling, is needed to prioritize and select features for inclusion in the platform from a set of candidates.

The second area for additional research is in agile development methods for platform-based systems of systems. Approaches for architecture-led incremental development have primarily focused on the software and system level [7, 17]. Further work is needed to model the more complicated dependencies in an SoS architecture, and to develop iteration planning strategies that accommodate the managerial independence of the constituent systems.

A final area for additional research is to create approaches to characterize and document constituent systems to support their use in systems of systems. Systematic approaches are needed to identify the relevant concerns, and collect and present the information to efficiently satisfy those concerns. An approach such as the creation of an ISO 42010-style architecture description viewpoint may be appropriate.

4

Design Assistant for NoSQL Technology Selection

Summary

Designing massively scalable, highly available big data systems is an immense challenge for software architects. Big data applications require distributed systems design principles to create scalable solutions, and the selection and adoption of open source and commercial technologies that can provide the required quality attributes. In big data systems, the data management layer presents unique engineering problems, arising from the proliferation of new data models and distributed technologies for building scalable, available data stores. Architects must consequently compare candidate database technology features and select platforms that can satisfy application quality and cost requirements. In practice, the inevitable absence of up-to-date, reliable technology evaluation sources makes this comparison exercise a highly exploratory, unstructured task. To address these problems, we have created a detailed feature taxonomy that enables rigorous comparison and evaluation of distributed database platforms. The taxonomy captures the major architectural characteristics of distributed databases, including data model and query capabilities. In this chapter we present the major elements of the feature taxonomy, and demonstrate its utility by populating the taxonomy for nine different database technologies. We also describe the QUality At Scale Knowledge Base for Big Data (QuABaseBD) knowledge base, which we have built to support the pop-

ulation and querying of database features by software architects. QuABaseBD links the taxonomy to general quality attribute scenarios and design tactics for big data systems. This creates a unique, dynamic knowledge resource for architects building big data systems.

4.1 Introduction

No industry in the history of engineering exhibits the rate of change we see in software technologies. By their very nature, complex software products can be created and evolved much more quickly than physical products, which require redesign, retooling, and manufacturing [104]. In contrast, the barriers to software product evolution are no greater than incorporating new functionality into code, testing, and releasing a new build for download on the Internet.

For software engineers building modern applications, there exists a dizzying number of potential off-the-shelf components that can be used as building blocks for substantial parts of a solution [24]. This makes component selection, composition, and validation a complex software engineering task that has received considerable attention in the literature (e.g. [14, 80, 107, 163]). While there is rarely a single 'right' answer when selecting a complex component for use in an application, selection of inappropriate components can be costly, reduce downstream productivity due to extensive rework, and even lead to project cancellation [137].

A contemporary application domain where there is particular difficulty in component selection is that of massively scalable, big data systems [2]. The exponential growth of data in the last decade has fueled rapid innovation in a range of components, including distributed caches, middleware and databases. Internet-born organizations such as Google and Amazon are at the cutting edge of this revolution, collecting, storing, and analyzing the largest data repositories ever constructed. Their pioneering efforts, for example [45] and [32], along with those of numerous other big data innovators, have created a variety of open source and commercial technologies for organizations to exploit in constructing massively scalable, highly available data repositories.

This technological revolution has instigated a major shift in database platforms for building scalable systems. No longer are relational databases the *de facto* standard for building data repositories. Highly distributed, scalable "NoSQL" databases [139] have emerged, which eschew strictly-defined normalized data models, strong data consistency guarantees, and SQL queries.

These features are replaced with schema-less data models, weak consistency guarantees, and proprietary APIs that expose the underlying data management mechanisms to the application programmer. Prominent examples of NoSQL databases include Cassandra, Riak, neo4j and MongoDB.

NoSQL databases achieve scalability through horizontally distributing data. In this context, distributed databases have fundamental quality constraints, as defined by Brewer's CAP Theorem [26]. When a network partition occurs ("P"—arbitrary message loss between nodes in the cluster), a system must trade consistency ("C"—all readers see the same data) against availability ("A"—every request receives a success/failure response).

The implications of the CAP theorem are profound for architects. To achieve high levels of scalability and availability, distribution must be introduced in all system layers. Application designs must then be aware of data replicas, handle inconsistencies from conflicting replica updates, and continue degraded operation in spite of inevitable failures of processors, networks, and software. This leads to new and emerging design principles, patterns and tactics based on established distributed systems theory, which must be adopted to successfully build scalable, big data systems [65].

This confluence of rapidly evolving technologies and (re-) emerging design principles and patterns makes designing big data systems immensely challenging. Application architectures and design approaches must exploit the strengths of available components and compose these into deployable, extensible and scalable solutions.

This is especially challenging at the data storage layer. The multitude of competing NoSQL database technologies creates a complex and rapidly evolving design space for an architect to navigate. Architects must carefully compare candidate database technologies and features and select platforms that can satisfy application quality and cost requirements. In the inevitable absence of up-to-date, unbiased technology evaluations, this comparison exercise is in practice a highly exploratory, unstructured task that uses an Internet search engine as the primary information gathering and assessment tool.

In this chapter we introduce a detailed feature taxonomy that can be used to systematically compare the capabilities of distributed database technologies. This taxonomy was derived from our experiences in evaluating databases for big data systems in a number of application domains (e.g., [93]). The feature taxonomy describes both the core architectural mechanisms of distributed databases, and the major data access characteristics that pertain to

the data architecture a given database supports. We describe our experience populating this taxonomy with the features of nine different databases to demonstrate its efficacy. We also describe a dynamic knowledge base we have built to semantically encode the feature taxonomy so that it can be queried and visualized.

The major contributions of this chapter are:

- The first presentation of a detailed, software and data architecture-driven feature taxonomy for distributed database systems.
- A demonstration of the efficacy of the taxonomy through its population with the features from nine different database technologies.
- A description of the semantic encoding and representation of the feature taxonomy to support efficient knowledge capture, query and visualization.
- An implementation of the knowledge base on the Semantic MediaWiki platform, using forms to add knowledge that conforms to the models, query-driven templates dynamically that render content as the knowledge base grows, and hyperlinked text, tables, and graphics for presentation and navigation.

4.2 Related Work

Our work builds upon and extends established work in software architecture knowledge management [4]. Early research in this area includes Kruchten [100], which introduced an ontology describing architectural design decisions for software systems. The ontology can be used to capture project-specific design decisions, their attributes, and relationships to create a graph of design decisions and their interdependencies. Our feature taxonomy also describes a graph of related design alternatives and relationships, but the knowledge relates to the class of distributed databases as opposed to a specific project or system. Hence, the knowledge has applicability to the broad class of big data software systems.

Other research has focused on using knowledge models to capture project-specific architectural decisions [3] and annotate design artifacts using ontologies [52, 72, 95]. Ontologies for describing general architecture knowledge have also been proposed. These include defining limited vocabularies [5], formal definitions of architecture styles [126], and supporting reuse of architecture documentation [155]. However, the inherent complexity of these approaches has severely limited adoption in practice.

Formal knowledge models for capturing architecture-related decisions also exist, for example [43, 82, 99, 98, 151], and [83]. Shahin describes a conceptual framework for these approaches that demonstrates significant overlap between the proposed concepts [141]. Our representation of the feature taxonomy takes a conceptually similar approach in that it semantically codifies a collection of general capabilities of big data systems, allowing an architect to explore the conceptual design space.

A primary use case for our knowledge base that semantically encodes the feature taxonomy is to provide decision support for evaluating alternative database technologies. Earlier work has demonstrated how a similar approach based on feature categorization of technology platforms can be effective in practice for middleware technology evaluation [107]. Our work extends this approach by reifying technology-specific knowledge as semantic relationships that enable querying of the knowledge to rapidly answer project-specific evaluation questions.

Architecture design assistants, such as ArchE [6], support reasoning from quality attributes to patterns, but do not directly support reasoning all the way through to COTS selection. Other assistants, such as AREL, capture design rationale for reuse [148]. Both types of assistants take a broad, general approach to architecture decisions. At the other extreme, curated COTS comparisons consolidate product feature lists but do not relate features to qualities or provide insight into tradeoffs, and are not easily queried or filtered [96, 162].

4.3 Feature Taxonomy

Scalability in big data systems requires carefully harmonized data, software and deployment architectures [65]. In the data layer, scalability requires partitioning the data sets and their processing across multiple computing and storage nodes. This inherently creates a distributed software architecture in the data tier. Contemporary database technologies adopt a variety of approaches

to achieve scalability. These approaches are primarily distinguished by the data model that a database supports and by the data distribution architecture it implements. Therefore, the selection of a specific database technology has a direct impact on the data and software architecture of an application.

Our feature taxonomy for distributed databases reflects these influences directly. It represents features in three categories related directly to the data architecture—namely *Data Model*, *Query Languages*, and *Consistency*—and four categories related directly to the software architecture—namely *Scalability*, *Data Distribution*, *Data Replication*, and *Security*. We decomposed each of these feature categories into a collection of specific features. Each feature has a set of allowed values representing the spectrum of design decisions that are taken in distributed databases. Depending on the database, some features may be assigned one or more of these values to fully characterize the database's capabilities.

In the following subsections we describe these feature categories and the spectrum of design decisions that are represented in our feature taxonomy. Space precludes a detailed description of each feature. Instead we briefly describe the major classes of features and their implications on both the data and software architecture.

4.3.1 Data Model

The data model supported by a distributed database dictates both how application data can be organized, and to a large extent, how it can be queried. Our taxonomy, shown in Table 4.1, captures these data architecture issues, organizing features in three broad groups:

1. **Data Organization**: These features capture how a database platform enables data to be modeled, whether fixed schemas are required, and support for hierarchical data objects.

2. **Keys and Indexes**: Flexibility in data object key definition, including support for secondary and composite keys, can greatly influence application performance, scalability and modifiability. This collection of features describes how any given database supports key definition.

3. **Query Approaches**: This collection of features describes the options available for querying a database, including key-based searching, text searching, and support for Map-Reduce based aggregation queries.

Table 4.1: Data Model Features

Feature	Allowed Values
Data Model	Column, Key-Value, Graph, Document, Object, Relational
Fixed Schema	Required, optional, none
Opaque Data Objects	Required, not required
Hierarchical Data Objects	Supported, not supported
Automatic Primary Key Allocation	Supported, not supported
Composite Keys	Supported, not supported
Secondary Indexes	Supported, not supported
Query by Key Range	Supported, not supported
Query by Partial Key	Supported, not supported
Query by Non-Key Value (Scan)	Supported, not supported
Map Reduce API	Builtin, integration with external framework, not supported
Indexed Text Search	Support in plugin (e.g., Solr), built-in proprietary, not supported

4.3.2 Query Languages

The query language features of a database directly affect application performance and scalability. For example, if a database does not return sorted result sets, the application itself must retrieve data from the database and perform the sort. For big data applications in which large results sets are common, this places a significant performance burden on an application, and uses resources (memory/CPU/network) that may be scarce under high loads.

Our feature taxonomy captures the broad query language characteristics, such declarative or imperative styles and languages supported, and the major detailed features that impart quality concerns. Table 4.2 illustrates these features. Note that for some features, for example *Languages Supported* and *Triggers*, multiple values may be assigned to the same feature for a database. This is a common characteristic that is seen across all categories in feature taxonomy.

Table 4.2: Query Language Features

Feature	**Allowed Values**
API-Based	Supported, Not Supported
Declarative	Supported, Not Supported
REST/HTTP-based	Supported, Not Supported
Languages supported	Java, C#, Python, C/C++, Perl, Ruby, Scala, Erlang, Javascript
Cursor-based queries	Supported, Not Supported
JOIN queries	Supported, Not Supported
Complex data types	Lists, maps, sets, nested structures, arrays, geospatial, none
Key matching options	Exact, partial match, wildcards, regular expressions
Sorting of query results	Ascending, descending, none
Triggers	Pre-commit, post-commit, none
Expire data values	Supported, Not Supported

4.3.3 Consistency

With the emergence of scalable database platforms, consistency has become a prominent quality of an application's data architecture. Transactional consistency properties that are standard in relational databases are rarely supported in NoSQL databases. Instead, a variety of approaches are supported for both transactional and replica consistency. This inevitably places a burden on the application to adopt designs that maintain strong data consistency or op-

erate correctly with weaker consistency. A common design denormalizes data records so that a set of dependent updates can be performed in a single database operation. While this approach ensures consistency in the absence of ACID transactional semantics, denormalization also leads to increased data sizes due to duplication, and increased processing in order to keep duplicates consistent.

The features in Table 4.3 are grouped into those that support strong consistency, and those that support eventual (replica) consistency. Strong consistency features such as ACID and distributed transactions reduce application complexity at the cost of reduced scalability. Eventual consistency is a common alternative approach in scalable databases. Eventual consistency relies on storing replicas of every data object to distribute processing loads and provide high availability in the face of the database node failures. The taxonomy describes a range of features that constitute eventually consistent mechanisms, including conflict detection and resolution approaches.

4.3.4 Scalability

Evaluating qualities like performance and scalability in absolute terms requires benchmarks and prototypes to establish empirical measures. However, the core architectural design decisions that underpin a database implementation greatly affect the scalability that an application can achieve. In our taxonomy, we capture some of these core scalability features, shown in Table 4.4.

Horizontal scaling spreads a data set across multiple nodes. In some databases, it is only possible to replicate complete copies of a database onto multiple nodes, which restricts scalability to the capacity of a single node—this is scaling up. Other databases support horizontal partitions, or sharding, to scale data onto multiple nodes.

Another key determinant of scalability is the approach to distributing client requests across database nodes. Bottlenecks in the request processing path for reads and writes can rapidly become inhibitors of scalability in a big data system. These bottlenecks are typically request or transaction coordinators that cannot be distributed and replicated, or processes that store configuration state that must be accessed frequently during request processing.

Table 4.3: Consistency Features

Feature	**Allowed Values**
Object-level atomic updates	Supported, Multi-Value Concurrency Control, conflicts allowed
ACID transactions in a single database	Supported, lightweight transactions (e.g., test and set), not supported
Distributed ACID transactions	Supported, not supported
Durable writes	Supported, not supported
Quorum Reads/Writes (replica consistency)	In client API, in database configuration, in the datacenter configuration, not supported
Specify number of replicas to write to	In client API, in database configuration, not supported, not applicable - master-slave
Behavior when specified number or replica writes fails	Rollback at all replicas, No rollback, error returned, hinted handoffs, not supported
Writes configured to never fail	Supported, not supported
Specify number of replicas to read from	In client API, in database configuration, not supported, not applicable - master-slave
Read from master replica only	Not supported, in the client API, not applicable - peer-to-peer
Object level timestamps to detect conflicts	Supported, not applicable (single threaded), not applicable (master slave), not supported

Table 4.4: Scalability Features

Feature	Allowed Values
Scalable distribution architecture	Replicate entire database only; horizontal data partitioning; horizontal data partitioning and replication
Scaling out—adding data storage capacity	Automatic data rebalancing; manual database rebalancing; not applicable (single server only)
Request load balancing	HTTP-based load balancer required; client requests balanced across any coordinator; fixed connection to a request coordinator
Granularity of write locks	Locks on data object only; Table level locks; database level locks; no locks (single threaded); no locks (optimistic concurrency control); no locks (conflicts allowed)
Scalable request processing architecture	Fully distributed - any node can act as a coordinator; centralized coordinator but can be replicated; centralized coordinator (no replication); requires external load balancer

4.3.5 Data Distribution

There are a number of software architecture alternatives that a database can adopt to achieve data distribution. These alternatives can greatly affect the quality attributes of the resulting system. To this end, the features in this category capture how a given database coordinates access to data that is distributed over deployment configurations ranging from single clusters to multiple geographically distributed data centers, shown in Table 4.5. The mechanisms used to locate data and return results to requesting clients are an important aspect of distributed data access that affects performance, availabil-

ity and scalability. Some databases provide a central coordinator that handles all requests and passes them on to other nodes where the data is located for processing. A more scalable solution is provided by databases that allow any database node to accept a request and act as the request coordinator.

Table 4.5: Data Distribution Features

Feature	**Allowed Values**
Data distribution architecture	Single database only; master-single slave; master-multiple slaves; multimaster
Data distribution method	User specified shard key; assigned key ranges to nodes; consistent hashing; not applicable (single server only)
Automatic data rebalancing	Failure triggered; new storage triggered; scheduled rebalancing; manual rebalancing; not applicable (single server only)
Physical data distribution	Single cluster; rack-aware on single cluster; multiple co-located clusters; multiple data centers
Distributed query architecture	Centralized process for key lookup; distributed process for key lookup; Direct replica connection only
Queries using non-shard key values	Secondary indexes; non-indexed (scan); not supported
Merging results from multiple shards	Random order; sorted order; paged from server; not supported

4.3.6 Data Replication

Data replication is necessary to achieve high availability in big data systems. This feature category is shown in Table 4.6. Replication can also enhance performance and scalability by distributing database read and write requests across replicas, with the inevitable trade-off of maintaining replica consistency. All databases that support replication adopt either a master-slave or peer-to-peer (multi-master) architecture, and typically allow a configurable number of replicas that can be geographically distributed across data centers.

Replication introduces the requirement on a database to handle replica failures. Various mechanisms, ranging from fully automated to administrative, are seen across databases for replica failure and recovery. Recovery is complex, as it requires a replica to 'catch up' from its failed state and become a true replica of the current database state. This can be done by replaying operation logs, or by simply copying the current state to the recovered replica.

4.3.7 Security

Security is necessary in the data tier of an application to ensure data integrity and prevent unauthorized access. This feature category is shown in Table 4.7. Our taxonomy captures the approaches supported by a database for authentication, which is often a key factor in determining how a data platform can be integrated into an existing security domain. We also capture features such as roles that greatly ease the overheads and complexity of administering database security, and support for encryption - an important feature for applications requiring the highest levels of data security.

Table 4.6: Data Replication Features

Feature	Allowed Values
Replication Architecture	Master-slave; peer-to-peer
Replication for backup	Supported; not supported
Replication across data centers	Supported by data center aware features; Supported by standard replication mechanisms; Enterprise edition only
Replica writes	To master replica only; to any replica; to multiple replicas; to specified replica (configurable)
Replica reads	From master replica only; from any replica; from multiple replicas; from specified replica (configurable)
Read repair	Per query; background; not applicable
Automatic Replica Failure Detection	Supported; not supported
Automatic Failover	Supported; not supported
Automatic new master election after failure	Supported; not supported; not applicable
Replica recovery and synchronization	Performed by administrator; supported automatically; not supported

Table 4.7: Security Features

Feature	**Allowed Values**
Client authentication	Custom user/password; X509; LDAP; Kerberos; HTTPS
Server authentication	Shared keyfile; server credentials
Credential store	In database; external file
Role-based security	Supported; not supported
Security role options	Multiple roles per user; role inheritance; default roles; custom roles; not supported
Scope of rules	Cluster; database; collection; object; field
Database encryption	Supported; not supported
Logging	Configurable event logging; configurable log flush conditions; default logging only

4.4 Knowledge Base Overview

This section describes how we instantiated and populated the feature taxonomy in a knowledge base that we call Quality Attributes at Scale Knowledge Base for Big Data, or QuABaseBD (pronounced *kay-base-bee-dee*).

QuABaseBD is a linked collection of computer science and software engineering knowledge created specifically for designing big data systems with NoSQL databases. As depicted in Fig. 4.1, QuABaseBD is presented to a user through a Web-based wiki interface. QuABaseBD is built upon the Semantic MediaWiki (SMW) platform (`https://semantic-mediawiki.org/`), which adds dynamic, semantic capabilities to the base MediaWiki implementation (as used, for example, for Wikipedia).

In contrast to a typical wiki such as Wikipedia, the pages in QuABaseBD are dynamically generated from information that users enter into a variety of structured forms. This significantly simplifies content authoring for QuABaseBD and ensures internal consistency, as newly-added content is automatically included in summary pages and query results without needing to manually add links. Form-based data entry structures knowledge capture when populating the knowledge base, which ensures that the new content adheres to the underlying knowledge model. Hence the dynamic, structured nature of the QuABaseBD ensures it can consistently capture and render knowledge useful for software architects exploring the design space for big data systems.

QuABaseBD exploits these dynamic, semantic capabilities to implement a model that represents fundamental software architecture design knowledge for building big data systems. The initial version of QuABaseBD populates this knowledge model specifically for designing the data layer of an application.

QuABaseBD links two distinct areas of knowledge through an underlying semantic model. These areas are:

1. **Software design principles for big data systems**—Knowledge pertaining to specific quality attribute scenarios and design tactics for big data systems.

2. **Database feature taxonomy**—Knowledge pertaining to the specific capabilities of NoSQL and NewSQL databases to support database evaluation and comparison, as described in the previous section.

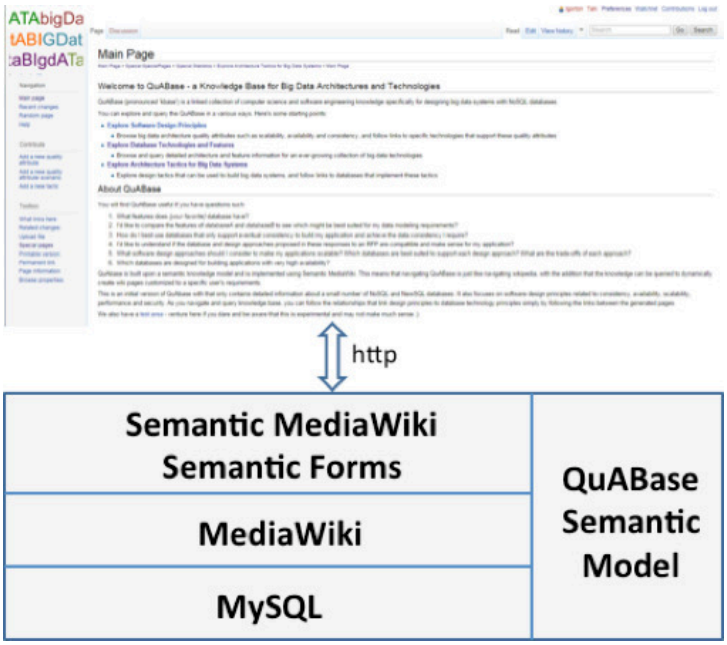


Figure 4.1: Conceptual Architecture of QuABaseBD

In the following, we describe how we link these two areas in the QuABase-BD knowledge model, and how we use the features of the SMW platform to populate and query the feature taxonomy.

4.4.1 Semantic Knowledge Model

The SMW platform supports semantic annotation of information as *Categories* and *Properties*. These annotations can be applied in an *ad hoc* manner as markup in the wikitext allowing the semantic structure to emerge from the contributed content. In contrast, QuABaseBD takes a structured approach to knowledge representation, as discussed above. All content is created using a form and rendered using a template, and the set of forms and templates embodies the structure of the semantic knowledge model. The knowledge model is split into two main sections. One section represents software architecture concepts related to quality attributes, quality attribute scenarios,

and architecture tactics. This section of the knowledge model is not intended to be a complete representation of general software design knowledge, but instead, it represents a growing collection of concepts and properties needed to reason about big data systems design and database technology selection. The purposes of this section of the model are to support the definition of architecturally significant requirements, to identify the quality attribute tradeoffs that are inherent in distributed data-intensive systems, and to describe design tactics to achieve particular architecture requirements. The second section of the knowledge model represents the feature taxonomy described above. The novelty of the QuABaseBD knowledge model is the linkage between the two sections through the relationship of an instance of a tactic to the instances of the features of a particular database that implement that tactic. This is shown in the extract of the knowledge model shown in Fig. 4.2.

Here we see that a *Quality Attribute* is represented using a *General Scenario*. The general scenario includes only those stimuli, responses, and response measures that are relevant in big data systems, in contrast to the abstract general scenarios presented by Bass and colleagues [10]. An example of the QuABaseBD General Scenario for Scalability is shown in Table 4.8.

Table 4.8: QuABaseBD General Scenario for Scalability

Stimulus:	Increase in load (demand) on a system resource such as processing (OR) I/O (OR) storage.
Environment:	Increase in load is transient (OR) Increase in load is permanent
Response:	System provides new resources to satisfy the load
Response Measure:	Ratio of increase in cost to provide new resources to value of increased load Time to provide additional resources when load increases

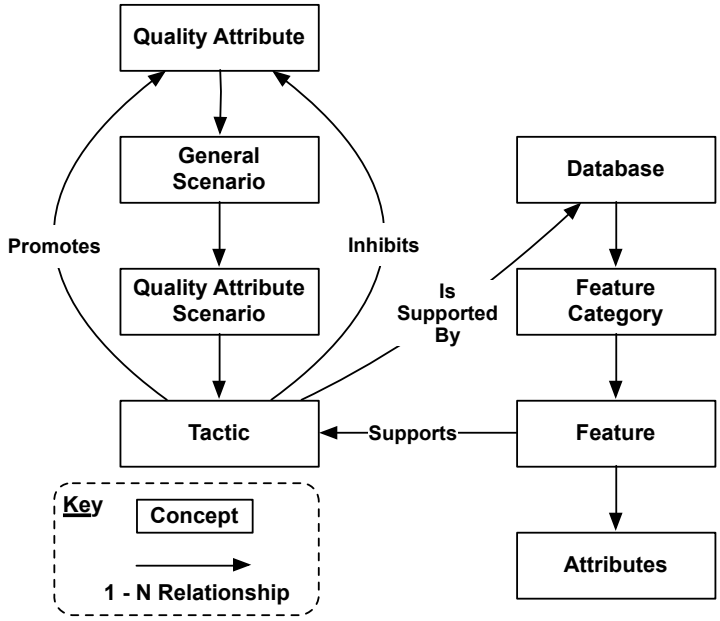


Figure 4.2: Extract from QuABaseBD Knowledge Model

A general scenario is a prototype that generates many *Quality Attribute Scenarios*, each of which combines a stimulus and response in the context of a big data system. A Quality Attribute Scenario covers a specific situation, and so we can identify the *Tactics* that can be employed to achieve the desired scenario response. Table 4.9 shows a quality attribute scenario derived from the Scalability General Scenario. Note that the stimulus, response, and response measure specialize the general scenario shown above.

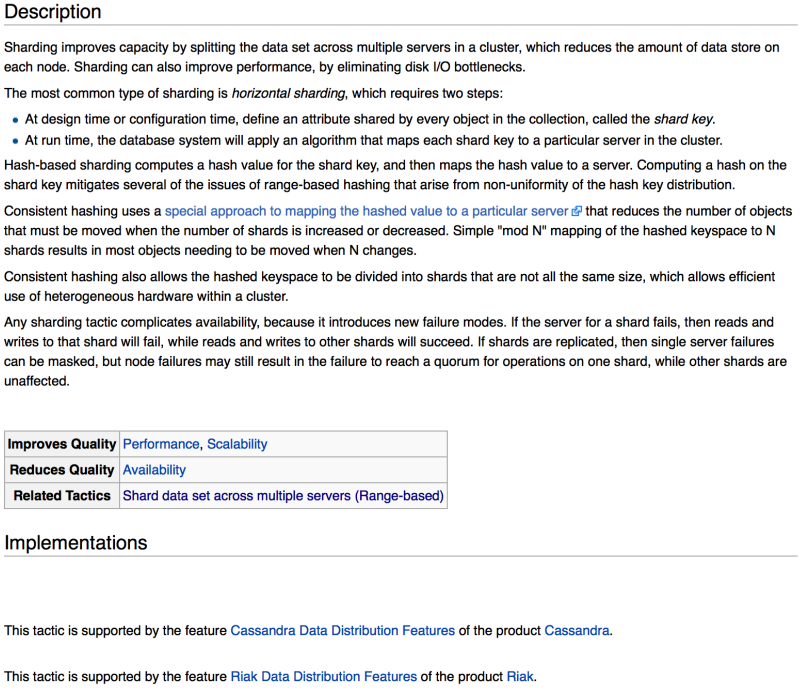
Tactics represent tradeoffs—each tactic promotes at least one quality attribute, and may inhibit other quality attributes. Although not represented in Fig. 4.2, the knowledge model also includes "anti-tactics", representing design approaches that prevent the desired response from being achieved. In Fig. 4.3, we show a screenshot from QuABaseBD for the Consistent Hashing tactic, referenced in the Scalability Scenario. The tactic definition includes a description that outlines the approach, followed by a table that summarizes the tradeoffs inherent in applying the tactic along with references to related tactics.

Table 4.9: Scalability Scenario

Scale to handle increased read or write request load	
Quality Attribute:	Scalability
Stimulus:	An increase in read requests is experienced by the system for a finite period of time (typically minutes to days)
Environment:	The system has been operating in production.
Response:	Additional nodes can be added to the cluster and the data set can be repartitioned to use the new resources.
Response Measure:	Downtime during repartitioning, Amount of manual intervention needed.
Tactics:	Automatically maintain cluster membership list (gossip), Shard data set across multiple servers (Consistent Hashing), Shard data set across multiple servers (Range-based), Load balance across replicas (one data center), Load balance across replicas (multiple data centers)
AntiTactics:	None

Tactics also represent specific design decisions that can be realized by a *Database* implementation, so we can say that a database supports a collection of tactics. This support is provided by one or more Features, which are grouped into *Feature Categories*. Finally, a feature has one or more *Attributes*, which represent the allowable values for the feature. At the bottom of the screenshot in Fig. 4.3, we see links to the specific NoSQL products that support the Consistent Hashing tactic.

This relationship between features and tactics allows an architect to reason about architecture qualities. For example, an architect may reason about the need for certain database features in order to achieve a particular system quality, or how different implementations of a feature in different databases will affect system qualities.



Description

Sharding improves capacity by splitting the data set across multiple servers in a cluster, which reduces the amount of data store on each node. Sharding can also improve performance, by eliminating disk I/O bottlenecks.

The most common type of sharding is *horizontal sharding*, which requires two steps:

- At design time or configuration time, define an attribute shared by every object in the collection, called the *shard key*.
- At run time, the database system will apply an algorithm that maps each shard key to a particular server in the cluster.

Hash-based sharding computes a hash value for the shard key, and then maps the hash value to a server. Computing a hash on the shard key mitigates several of the issues of range-based hashing that arise from non-uniformity of the hash key distribution.

Consistent hashing uses a special approach to mapping the hashed value to a particular server that reduces the number of objects that must be moved when the number of shards is increased or decreased. Simple "mod N" mapping of the hashed keyspace to N shards results in most objects needing to be moved when N changes.

Consistent hashing also allows the hashed keyspace to be divided into shards that are not all the same size, which allows efficient use of heterogeneous hardware within a cluster.

Any sharding tactic complicates availability, because it introduces new failure modes. If the server for a shard fails, then reads and writes to that shard will fail, while reads and writes to other shards will succeed. If shards are replicated, then single server failures can be masked, but node failures may still result in the failure to reach a quorum for operations on one shard, while other shards are unaffected.

Improves Quality	Performance, Scalability
Reduces Quality	Availability
Related Tactics	Shard data set across multiple servers (Range-based)

Implementations

This tactic is supported by the feature Cassandra Data Distribution Features of the product Cassandra.

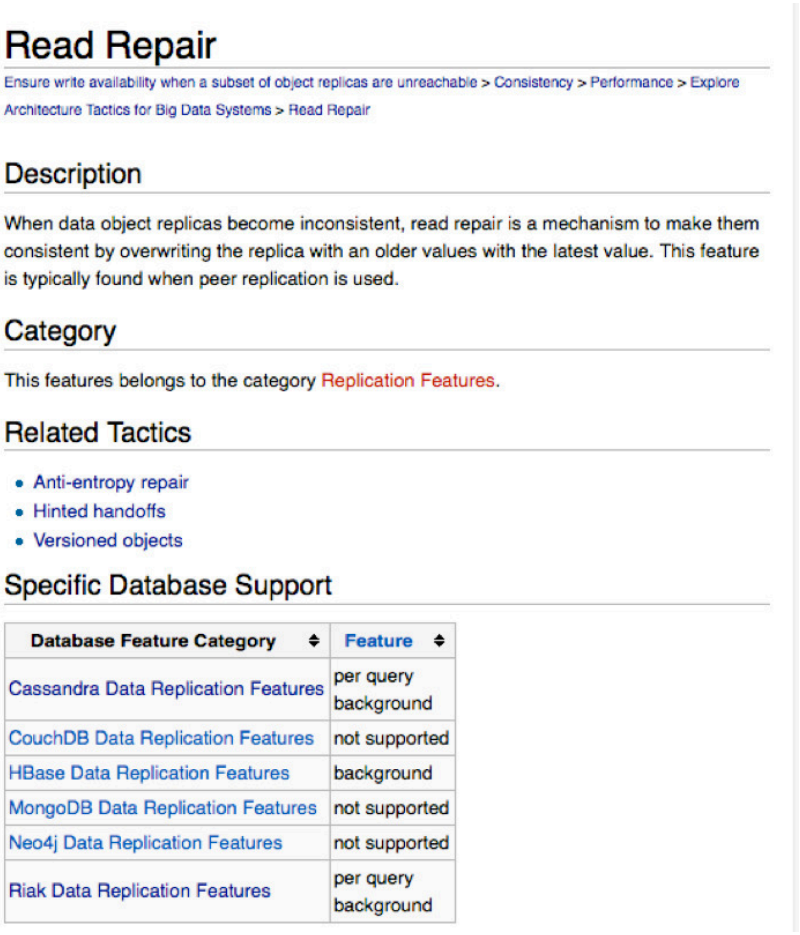
This tactic is supported by the feature Riak Data Distribution Features of the product Riak.

Figure 4.3: Screenshot of a Tactic Page in QuABaseBD

An example of how QuABaseBD uses a template to link tactics to database implementations is shown in Fig. 4.4. This illustrates the feature page for the *Read Repair* feature, which is in the feature category *Replication Features*. The related tactics that are supported by this feature are selected during the content creation process, and the table showing which databases implement the feature is generated dynamically by querying the knowledge base content.

4.4.2 QuABaseBD Implementation of Feature Taxonomy

From the main QuABaseBD page, users can choose to explore the knowledge base content for specific database technologies. Fig. 4.5 shows an extract from the main database page. This table is a dynamically generated list (the result of the SMW query shown in Fig. 4.6) of the databases for which QuABaseBD currently provides information.



Read Repair

Ensure write availability when a subset of object replicas are unreachable > Consistency > Performance > Explore Architecture Tactics for Big Data Systems > Read Repair

Description

When data object replicas become inconsistent, read repair is a mechanism to make them consistent by overwriting the replica with an older values with the latest value. This feature is typically found when peer replication is used.

Category

This features belongs to the category Replication Features.

Related Tactics

- Anti-entropy repair
- Hinted handoffs
- Versioned objects

Specific Database Support

Database Feature Category	Feature
Cassandra Data Replication Features	per query background
CouchDB Data Replication Features	not supported
HBase Data Replication Features	background
MongoDB Data Replication Features	not supported
Neo4j Data Replication Features	not supported
Riak Data Replication Features	per query background

Figure 4.4: Linking from Tactics to Implementations and Features

When a user navigates to the content for a specific database, for example Riak, they see a page that gives a brief overview of the database and a table listing the feature categories from the feature taxonomy, as shown in Fig. 4.7.

Knowledge creators can edit the values for the feature taxonomy for each database. As discussed above, all content creation is done using SMW forms. Clicking on an *Edit* link displays a form for the associated feature category, as shown in Fig. 4.8 for the *Data Replication* category. The forms render the

Database	Data Model
HBase	Column
Cassandra	Column
MongoDB	Document
CouchDB	Document
Neo4j	Graph
Riak	Key-Value
FoundationDB	NewSQL
VoltDB	NewSQL

Figure 4.5: QuABaseBD Database Knowledge

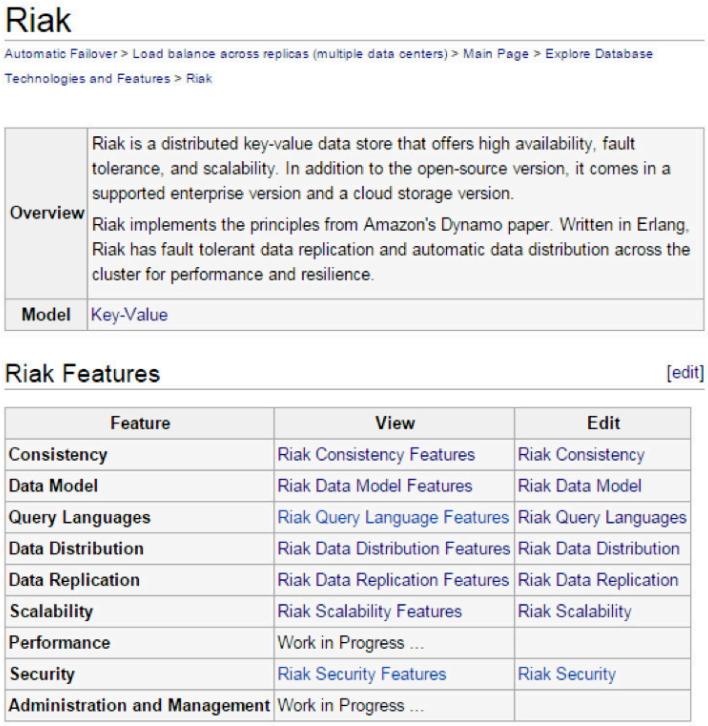
```
{{#ask: [[Category:Database]]
|intro=Select any of the database below to \
get information on their features and \
the tactics they support
| mainlabel=Database
| ?Has DB Model=Data Model
| sort=Has DB Model
| order=asc
}}
```

Figure 4.6: Query to List Databases and Data Model Types

elements of the feature taxonomy as fields organized in tabs, along with a valid set of values that a knowledge creator can select from for each feature. Using forms in this manner, the QuABaseBD implementation ensures a consistent set of values for each feature, thus greatly facilitating ease of comparison across databases.

When a form is saved by a knowledge creator, an associated SMW template performs two actions:

1. It associates the selected feature values with a semantic property for each feature, creating a collection of RDF triples with the general form Feature, Has Value, Value to populate the semantic feature taxonomy.



Riak

Automatic Failover > Load balance across replicas (multiple data centers) > Main Page > Explore Database Technologies and Features > Riak

Overview	Riak is a distributed key-value data store that offers high availability, fault tolerance, and scalability. In addition to the open-source version, it comes in a supported enterprise version and a cloud storage version. Riak implements the principles from Amazon's Dynamo paper. Written in Erlang, Riak has fault tolerant data replication and automatic data distribution across the cluster for performance and resilience.
Model	Key-Value

Riak Features [edit]

Feature	View	Edit
Consistency	Riak Consistency Features	Riak Consistency
Data Model	Riak Data Model Features	Riak Data Model
Query Languages	Riak Query Language Features	Riak Query Languages
Data Distribution	Riak Data Distribution Features	Riak Data Distribution
Data Replication	Riak Data Replication Features	Riak Data Replication
Scalability	Riak Scalability Features	Riak Scalability
Performance	Work in Progress ...	
Security	Riak Security Features	Riak Security
Administration and Management	Work in Progress ...	

Figure 4.7: QuABaseBD Knowledge for Riak

2. It generates a wiki page for knowledge consumers by substituting the selected feature values into a text template that describes each feature and the associated value for each database.

An example of the resulting generated wiki page is shown in Fig. 4.9. To a knowledge consumer, the mechanics of selecting alternate feature values is completely hidden. They simply see a description of each feature and how it is realized in the associated database.

A major advantage of associating each feature with a semantic property is that it facilitates fine grain searching across the different features for each database. Fig. 4.10 illustrates the search facilities available to users for each feature category. By default, if no specific feature values are selected in the query form (top left), then all the databases in the QuABaseBD that have a

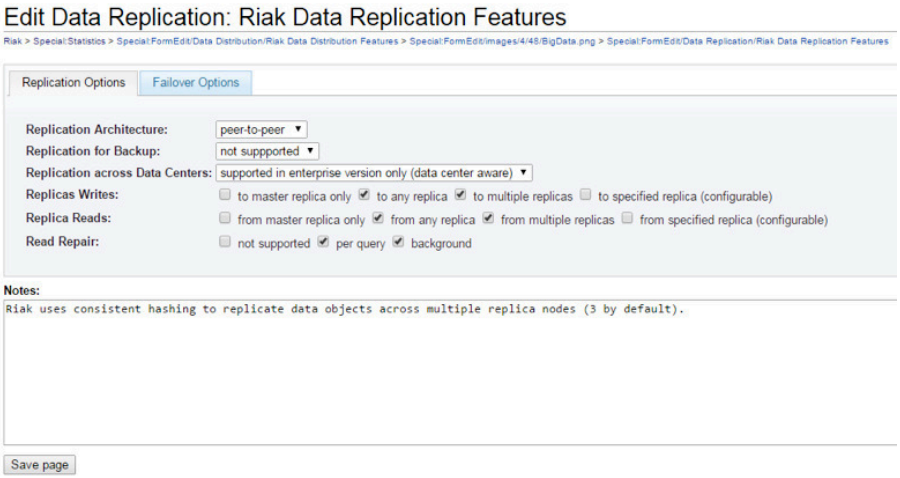


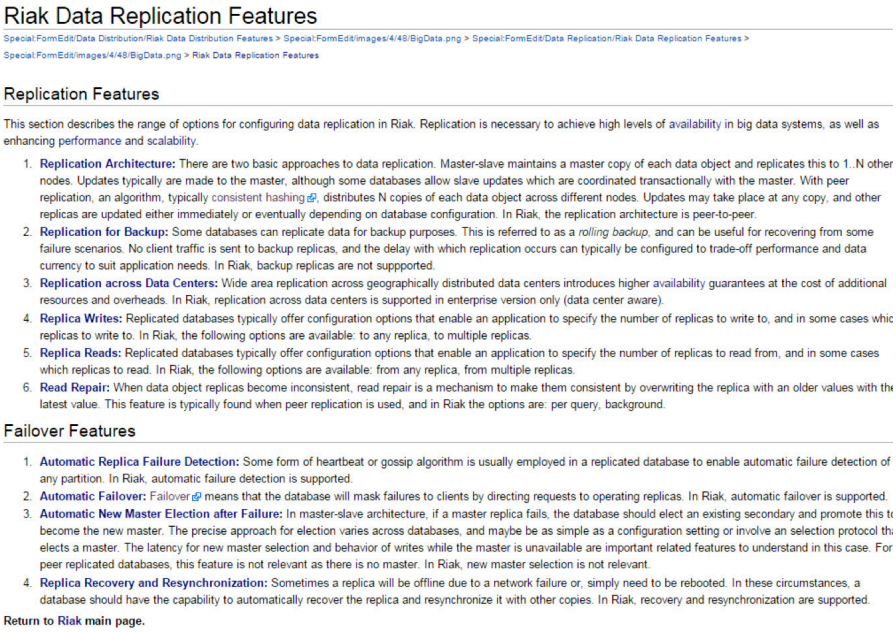
Figure 4.8: Populating the QuABaseBD Feature Taxonomy for Riak Replication

completed feature page for that category are displayed (bottom right). The generated table facilitates a direct comparison of the databases in the QuABaseBD . The user can then choose specific values in the query form for the various features they are interested in and generate customized result sets to answer their specific questions.

4.5 QuABaseBD Use Cases

Design and implementation decisions for QuABaseBD were driven by two primary use cases:

- Architecture Design: The target user is an architect who has little experience with big data systems and NoSQL technology. He or she uses the General Scenarios and Quality Attribute Scenarios in QuABaseBD to support definition of architecturally significant requirements. The Quality Attribute Scenarios are used to select appropriate Tactics, and finally, one or more candidate NoSQL products are selected that implement the tactics.



Riak Data Replication Features

Special:FormEdit/Data Distribution/Riak Data Distribution Features > Special:FormEdit/images/4/48/BigData.png > Special:FormEdit/Data Replication/Riak Data Replication Features > Special:FormEdit/images/4/48/BigData.png > Riak Data Replication Features

Replication Features

This section describes the range of options for configuring data replication in Riak. Replication is necessary to achieve high levels of availability in big data systems, as well as enhancing performance and scalability.

1. **Replication Architecture:** There are two basic approaches to data replication. Master-slave maintains a master copy of each data object and replicates this to 1..N other nodes. Updates typically are made to the master, although some databases allow slave updates which are coordinated transactionally with the master. With peer replication, an algorithm, typically consistent hashing, distributes N copies of each data object across different nodes. Updates may take place at any copy, and other replicas are updated either immediately or eventually depending on database configuration. In Riak, the replication architecture is peer-to-peer.
2. **Replication for Backup:** Some databases can replicate data for backup purposes. This is referred to as a *rolling backup*, and can be useful for recovering from some failure scenarios. No client traffic is sent to backup replicas, and the delay with which replication occurs can typically be configured to trade-off performance and data currency to suit application needs. In Riak, backup replicas are not suppported.
3. **Replication across Data Centers:** Wide area replication across geographically distributed data centers introduces higher availability guarantees at the cost of additional resources and overheads. In Riak, replication across data centers is supported in enterprise version only (data center aware).
4. **Replica Writes:** Replicated databases typically offer configuration options that enable an application to specify the number of replicas to write to, and in some cases whic replicas to write to. In Riak, the following options are available: to any replica, to multiple replicas.
5. **Replica Reads:** Replicated databases typically offer configuration options that enable an application to specify the number of replicas to read from, and in some cases which replicas to read. In Riak, the following options are available: from any replica, from multiple replicas.
6. **Read Repair:** When data object replicas become inconsistent, read repair is a mechanism to make them consistent by overwriting the replica with an older values with the latest value. This feature is typically found when peer replication is used, and in Riak the options are: per query, background.

Failover Features

1. **Automatic Replica Failure Detection:** Some form of heartbeat or gossip algorithm is usually employed in a replicated database to enable automatic failure detection of any partition. In Riak, automatic failure detection is supported.
2. **Automatic Failover:** Failover means that the database will mask failures to clients by directing requests to operating replicas. In Riak, automatic failover is supported.
3. **Automatic New Master Election after Failure:** In master-slave architecture, if a master replica fails, the database should elect an existing secondary and promote this t become the new master. The precise approach for election varies across databases, and maybe be as simple as a configuration setting or involve an selection protocol th elects a master. The latency for new master selection and behavior of writes while the master is unavailable are important related features to understand in this case. For peer replicated databases, this feature is not relevant as there is no master. In Riak, new master selection is not relevant.
4. **Replica Recovery and Resynchronization:** Sometimes a replica will be offline due to a network failure or, simply need to be rebooted. In these circumstances, a database should have the capability to automatically recover the replica and resynchronize it with other copies. In Riak, recovery and resynchronization are supported.

Return to Riak main page.

Figure 4.9: Template-generated Wiki Page for Riak Replication

- Identification of Alternatives: The target user is an architect who has some experience with big data systems and NoSQL technology. The architect knows some or all of the product features needed for his or her system, and uses the QuABaseBD feature queries to identify one or more suitable candidate NoSQL products.

Both of these cases might be followed by a use case in which the architect "works backwards" from a candidate NoSQL product. Large COTS products, like these NoSQL databases, implement many tactics, with each tactic embodying a set of quality attribute tradeoffs. An architect uses QuABaseBD to identify a product because it supports tactics he or she desires, but then the architect must ensure that other tactics supported by the product do not embody tradeoffs that would be detrimental to system qualities. QuABaseBD

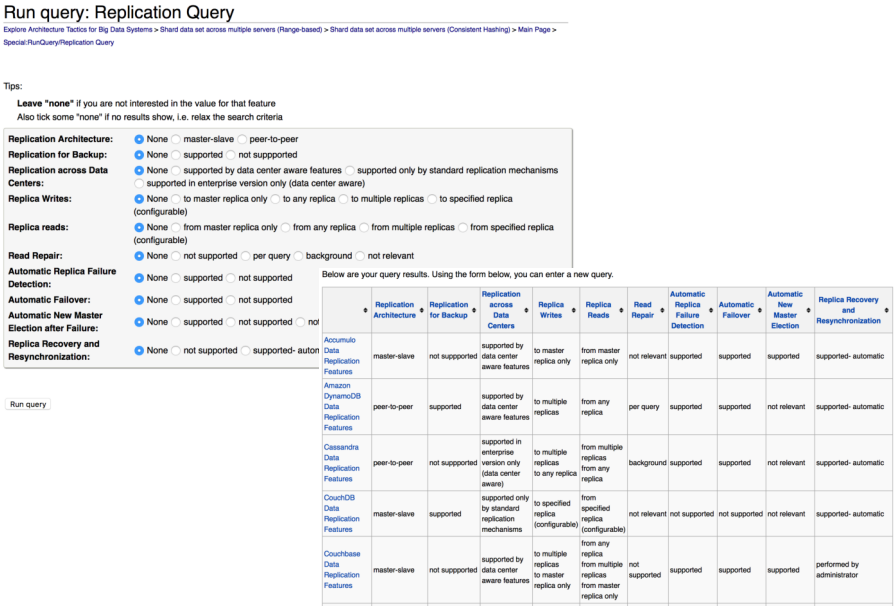


Figure 4.10: Querying the Database Features

supports tracing from the Tactics that are implemented by a candidate product, identifying the quality attribute tradeoffs that those tactics embody, and using Quality Attribute Scenarios to provide concrete examples of the implications of each tradeoff.

4.6 Demonstrating Feature Taxonomy Efficacy

The feature taxonomy and the initial implementation of QuABaseBD were co-developed during an evaluation of four NoSQL databases, namely MongoDB, Cassandra, Neo4j and Riak [93]. During this project, QuABaseBD was populated with information about those databases by the authors of this paper. The feature taxonomy and the allowed values were based on the capabilities of the

four databases. As each of these databases represented a different NoSQL data model, we expected that the feature taxonomy we developed would be sufficiently general to facilitate differentiation amongst a much larger collection of technologies.

To assess the efficacy of the feature taxonomy, we further populated QuABaseBD with information about five other databases: Accumulo, Hbase, CouchDB, FoundationDB, and VoltDB. Graduate students helped perform this *content curation*[1]. Each contributor located and reviewed the product documentation for the database published on the Internet by the vendor or open source developer, and used the QuABaseBD SMW forms to enter the relevant information to populate the feature taxonomy.

Throughout this curation process, we observed no additional features were needed to describe the functionality delivered by any of these databases, and curators were able to map all product features to one or more of the taxonomy features. For several features, we incorporated new *allowed values* to more precisely capture a particular database's capabilities. For example, CouchDB eschews object locking on updates in favor of Multi-Version Concurrency Control (MVCC). This approach was not supported in our initial set of databases, and hence we added MVCC as an allowed value for the *Object Level Atomic Updates* feature. Additional allowed values for a number of features were also necessary to describe the capabilities of two so-called *NewSQL* databases that were incorporated into QuABaseBD , namely VoltDB and FoundationDB. These databases are distributed, scalable implementations of relational databases, supporting large subsets of the standard SQL query language. Encouragingly, the feature taxonomy was sufficiently rich to enable us to capture the capabilities of these NewSQL databases in QuABaseBD .

Fig. 4.11 is a simple visualization of the populated feature taxonomy that we can generate by querying QuABaseBD . It shows how each of the nine databases in the knowledge base relate to a value for the *Load Balancing* feature in the *Scalability* category. This visualization clearly shows how the databases cluster around the different feature values. Being able to convey such information visually is important as it makes it straightforward for architects to focus on features of interest and see how databases compare. As the number of databases in QuABaseBD grows, we will develop further visualizations to help convey the information in a concise and informative manner.

[1] `http://en.wikipedia.org/wiki/Content_curation`

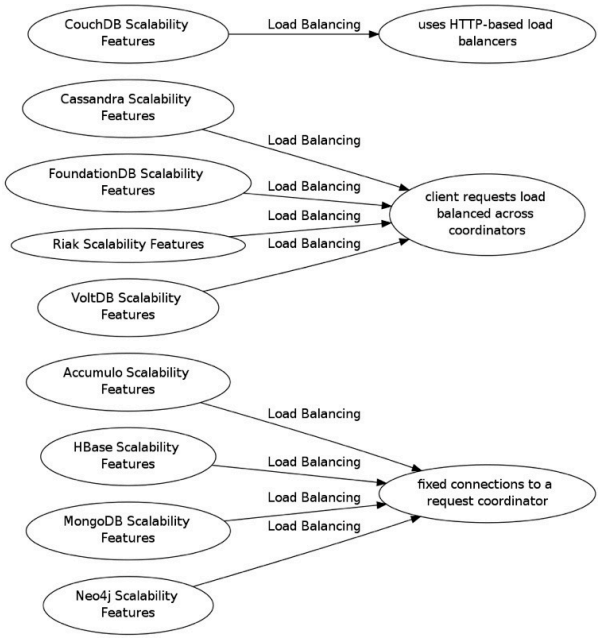


Figure 4.11: Clustering of Implementations for the Load Balancing Feature

4.7 User Trials

In order to prepare for public deployment of the QuABaseBD , we have performed usability and utility testing of the resulting Web site. We publicized QuABaseBD through the authors' professional networks, and opened QuABaseBD to volunteers to perform testing. Volunteers were requested to purposefully explore the knowledge base: when access credentials were issued, they were encouraged to use the knowledge base to solve a particular design problem that we provided, rather than simply browsing the QuABaseBD content. These open access sessions were limited to one hour in duration. Users were given a worksheet to record their impressions from the testing. There was no additional training, guidance, or instruction provided.

All of the 20 users who provided feedback on their experience were software architects with between 5 and 23 years of experience. All but two users characterized their expertise in big data systems as "somewhat knowledgeable", but they did not have any specific experience in any of the database technologies currently represented in QuABaseBD .

The "main page", where users enter the knowledge base after authenticating, offers three options: Explore Software Design Principles, Explore Database Technologies and Features, and Explore Architecture Tactics for Big Data Systems. The workflow that testers employed was split nearly evenly between those that started with architecture tactics, and those that started with database features. No testers started with design principles, although some testers eventually explored this section of the knowledge base. These workflows matched our pre-test expectations: The software design principles path is intended to help define architecturally significant requirements. If these are already established (as they were in the test problems), then we expect users to start with tactics (top-down reasoning), or database features (bottom-up reasoning). This gives us confidence that the QuABaseBD design is structured to support this use case.

Testers starting with database features made extensive use of the faceted search capability and the tabular results visualizations (for example, the pages shown in Fig. 4.10). The information in these tables was sufficient to answer the testers' questions. Few of these testers relied on the detailed feature description pages. In contrast, testers who started with tactics relied on following links within QuABaseBD , which led them to the detailed feature description pages. Interestingly, no testers employed the full-text search capability of the SMW platform.

All but one tester said that they were able to answer all of their questions using the content of QuABaseBD . In providing feedback on the utility of the knowledge base, testers were asked to rate their confidence that their answers were complete and correct. The large majority rated their confidence as 3 or 4 on a scale of 1 (no confidence) to 5 (absolutely sure).

Only one tester followed any of the hyperlinks to external resources that are provided in the detailed product feature descriptions. We hypothesize that this may be due to the limited time we allowed for testing, but again this gives us confidence that the current QuABaseBD content is sufficiently extensive to meet many of the anticipated needs of the big data software engineering community.

4.8 Further Work and Conclusions

We are currently working towards finalizing QuABaseBD for public release. To this end we are testing the knowledge base functionality, and validating with experts on each database that our curated values for the database features are valid. We believe that as a curated scientific knowledge base, there are high expectations that the content in QuABaseBD is trustworthy at all times. To this end, we are working to design a systematic curation process where a small cohort of experts will be responsible for changes to the content. We anticipate that visitors to the knowledge base will be able to suggest changes through associated comments pages, and these proposals will be assessed for inclusion by the curators.

The SMW platform has provided a usable interface for these architects to derive answers to questions in a short amount of time. The platform's semantic metamodel, combined with the forms for content entry and templates for content rendering, allowed us to represent a novel domain knowledge model for big data architecture and technology in a form (a wiki) that users are familiar with.

After deployment, we will continue to expand the QuABaseBD content, and have identified several areas for future work. Manual creation and maintenance of content is inefficient, as the scope of the content is expanded to cover more of the database product landscape. Automation of these tasks is therefore needed, using technology such as machine learning to extract content from product documentation.

The terminology used in the database feature taxonomy needs further study. We chose terms that are abstract and general, rather than adopting implementation-specific terms. In some cases, adding alternative terminology for feature values may increase usability. We also anticipate expanding and restructuring the taxonomy as we receive feedback from the community.

We hope that QuABaseBD can become an enduring resource to support the design of big data systems. We also hope it will demonstrate the potential of curated, dynamic knowledge bases as sources of highly reliable scientific knowledge, as well as the basis for a next generation of software engineering decision support tools.

5

Application-Specific Evaluation of NoSQL Databases

Summary

The selection of a particular NoSQL database for use in a big data system imposes a specific distributed software architecture and data model, making the technology selection difficult to defer and expensive to change. This chapter reports on the selection of a NoSQL database for use in an Electronic Healthcare Record system being developed by a large healthcare provider. We performed application-specific prototyping and measurement to identify NoSQL products that fit data model and query use cases, and met performance requirements. We found that database throughput varied by a factor of 10, read operation latency varied by a factor of 5, and write latency by a factor of 4 (with the highest throughput product delivering the highest latency). We also found that the throughput for workloads using strong consistency was 10-25% lower than workloads using eventual consistency. We conclude by reflecting on some of the fundamental difficulties of performing detailed technical evaluations of NoSQL databases specifically, and big data systems in general, that have become apparent during our study.

5.1 Introduction

At the heart of many big data systems are NoSQL database management systems that are simpler than traditional relational databases and provide higher scalability and availability [139]. These databases are typically designed to scale horizontally across clusters of low cost, moderate performance servers. They achieve high performance, elastic storage capacity, and availability by replicating and partitioning data sets across a cluster of servers. Each of these products implements a different data model and query language, as well as specific mechanisms to achieve distributed data consistency and availability.

When a big data system uses a particular database, the data, consistency and distribution models imposed by the database have a pervasive impact on the design of the associated applications [65]. Hence, the selection of a particular NoSQL database must be made early in the design process and is difficult and expensive to change downstream. In other words, NoSQL database selection becomes a critical architectural decision for big data systems.

Commercial off-the-shelf (COTS) software selection has been extensively studied in software engineering [14, 36, 163]. In complex technology landscapes with multiple competing products, developers must balance the cost and speed of the selection process against the fidelity of the analysis [108]. While there is rarely a single right answer in selecting a complex component for an application, selection of inappropriate components can be costly, reduce downstream productivity due to rework, and even lead to project cancellation. This is especially true for large scale, big data systems, due to their complexity and the magnitude of the investment.

There are several unique challenges that make selection of NoSQL databases for use in big data applications a particularly hard problem:

- This is an early architecture decision that must be made with inevitably incomplete requirements.
- Capabilities and features vary widely across NoSQL databases and performance is very sensitive to how a product's data model and query features match application needs, making generalized comparisons difficult.
- Production-scale prototypes, with hundreds of servers, multi-terabyte data sets, and thousands or millions of clients, are usually impractical.

- The solution space is changing rapidly, with new products emerging and existing products releasing several versions per year with ever evolving feature sets.

We faced these challenges during a recent project for a healthcare provider considering the use of NoSQL databases for an Electronic Health Record (EHR) system. The next section provides details of the project context and technology evaluation approach. This is followed in §5.3 by a discussion of the prototype design and configuration. §5.4 presents the performance test results. We conclude with a reflection on lessons learned during this project.

5.2 Electronic Health Record Case Study

5.2.1 Project Context

Our customer was a large healthcare provider developing a new Electronic Health Record (EHR) system. This system supports healthcare delivery for over 9,000,000 patients at more than 100 facilities across the globe. The data growth rate is more than one terabyte per month, and all data must be retained for 99 years.

NoSQL technologies were considered attractive candidates for two specific uses, namely:

- the primary data store for the EHR system
- a local cache at each site to improve request latency and availability

This will replace an existing system that uses thick client applications running at sites around the world, all connected to a centralized relational database, so the customer wanted to characterize performance with hundreds of concurrent database sessions.

The customer was familiar with RDMS technology for these use cases, but had no experience using NoSQL, so we were directed to focus our evaluation only on NoSQL technology.

5.2.2 Evaluation Approach

The approach builds on previous work on middleware evaluation [108, 107] and was customized to address the characteristics of big data systems. The basic main steps are depicted in Fig. 5.1 and outlined below.

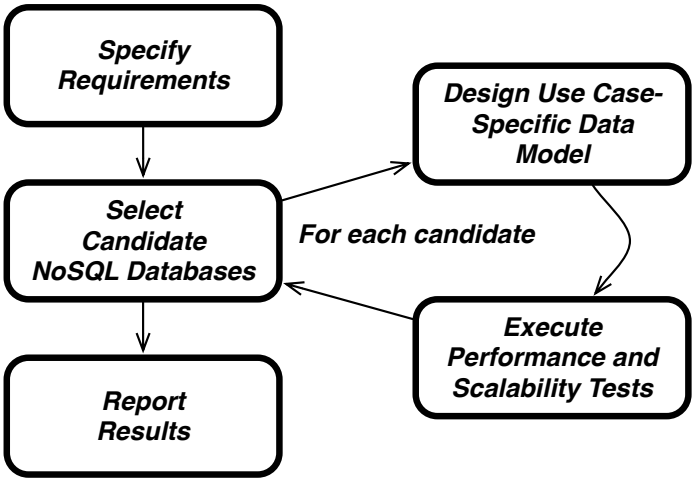


Figure 5.1: Lightweight Evaluation and Architecture Prototyping for Big Data (LEAP4BD)

1. *Specify Requirements*

   We used a stakeholder workshop to elicit key functional and quality attribute requirements to frame the evaluation. These key requirements were:

   Performance/Scalability: The main quantitative requirements were to replicate data across geographically distributed data centers, and to achieve high availability and low latencies under load in distributed database deployments. Hence understanding the inherent performance and scalability that is achievable with each candidate NoSQL database was an essential part of the evaluation.

   Data Model Mapping Complexity: Healthcare systems have common logical data models and query patterns that need to be supported by a NoSQL database. This required us to evaluate the specific data modeling and query features for each product, including capabilities to maintain replica consistency in a distributed deployment.

We next helped the customer define two primary use cases for the EHR system. These drove the evaluation that we performed in subsequent steps in the project. The first use case was to read recent medical test results for a single patient, which is a core function used to populate the user interface whenever a clinician selects a new patient. The second use case was achieving strong replica consistency when a new medical test result is written for a patient, so that all clinicians using the EHR to make patient care decisions will see the same information, whether they are at the same site as the patient, or providing telemedicine support from another location.

2. *Select Candidate NoSQL Databases*

   Our customer was specifically interested in evaluating how different NoSQL data models (key-value, column, document, graph) would support their application domain, and so we selected one NoSQL database from each category to investigate in detail. We subsequently ruled out graph databases, as none provided the horizontal partitioning required for this customer's application. We chose Riak, Cassandra and MongoDB as the three candidates, based on product maturity and availability of enterprise support.

3. *Design and Execute Performance Tests*

   In order to make an apples to apples comparison of the databases that were evaluated, we defined and performed a systematic test procedure. Based on the use cases defined during the requirements step, we:

   - Defined and implemented a consistent test environment, which included server platform, test client platform, and network topology.
   - Mapped the logical model for a patient's medical test history onto each database's data model and loaded the resulting database with a large collection of synthetic test data.
   - Created a load test client that implements the database read and write operations defined for each use case. This client is capable of issuing many simultaneous requests so that we can analyze how each product responds as the request load increases.
   - Defined and executed test scripts that exerted a specified load on the database using the test client.

We executed each test case on several distributed configurations to measure performance and scalability. These test scenarios ranged from baseline testing on a single server to nine server instances that sharded and replicated data.

This enabled us to produce a consistent set of test results that assessed the likely performance and scalability of each database for this customer's EHR system. The details of the environment and test design are presented in the next section.

5.3 Prototype and Evaluation Setup

5.3.1 Test Environment

The three databases we tested were:

1. MongoDB version 2.2, a document store[1];
2. Cassandra version 2.0, a column store[2];
3. Riak version 1.4, a key-value store[3].

In §5.4, we report performance results for two database server configurations: Single node server, and a nine-node configuration that was representative of a production deployment. Executing on a single node allowed us to validate our test environment for each database. The nine-node cluster was configured to represent a geographically distributed deployment across three data centers. The data set was sharded across three nodes, and then replicated to two additional groups of three nodes each (we refer to this configuration as "3x3"). This was achieved using MongoDB's primary/secondary feature, and Cassandra's data center aware distribution feature. Riak did not directly support this 3x3 data distribution, so we used a configuration where the data was sharded across all nine nodes, with three replicas of each shard stored across the nine nodes. Testing was performed using the Amazon EC2 infrastructure-as-a-service cloud environment. (`http://aws.amazon.com/ec2/`). Database servers executed on m1.large

[1] `http://docs.mongodb.org/v2.2/`
[2] `http://www.datastax.com/documentation/cassandra/2.0`
[3] `http://docs.basho.com/riak/1.4.10/`

instances. Database data and log files were stored on separate EBS volumes attached to each server instance. The EBS volumes were not provisioned with the IOPS feature, to minimize the tuning parameters used in each test configuration. The test client was also executed on an m1.large instance. The servers and the test client both used the CentOS operating system (`http://www.centos.org`). All instances were in the same EC2 availability zone (i.e. the same Amazon cloud data center).

5.3.2 Mapping the data model

Most of the prototyping effort was spent mapping the application-specific logical data model onto the particular data model, indexing, and query language capabilities of each database to be tested.

We used the HL7 Fast Healthcare Interoperability Resources (FHIR) (`http://www.hl7.org/implement/standards/fhir/`) as the logical data model for our analysis and prototyping. The set of all test results for a patient were modeled using the *FHIR Patient Resources* (e.g., demographic information such as names, addresses, and telephone numbers) along with *FHIR Observation Resources* (e.g., test type, result quantity, and result units). There was a one-to-many relation from each patient to the associated test results. Although this was a relatively simple model, the internal complexity of the FHIR Patient Resource, with multiple addresses and phone numbers, along with the one-to-many relation from patient to observations, required a number of data modeling design decisions and tradeoffs in the data mapping.

The most significant data modeling challenge was the representation of the one-to-many relation from patient to lab results, coupled with the need to efficiently access the most-recently written lab results for a particular patient. Zola has analyzed the various approaches and tradeoffs of representing the one-to-many relation in MongoDB [165]. We used a composite index of [Patient ID, Observation ID] for lab result records, and also indexed by the lab result date-time stamp. This allowed efficient retrieval of the most recent lab result records for a particular patient.

A similar approach was used for Cassandra. Here we used a composite index of [PatientID, lab result, date-time stamp]. This caused the result set returned by the query to be sorted by the server, making it efficient to retrieve the most recent lab records for a particular patient.

In Riak, representing the one-to-many relation was more complicated. Riak's key-value data model provides the capability to retrieval a value, given a unique key. Riak also provides a secondary index capability to avoid a full scan when the key is not known. However, each node in the cluster stores only the secondary indices for those shards stored by the node. A query to match a secondary index value causes the request coordinator to perform a scatter-gather, asking each node for records with the requested secondary index value, waiting for all nodes to respond, and then sending the list of keys for the matching records back to the requester. The requester must then make a second request with the list of keys, to retrieve the record values.

The latency of the scatter-gather to locate records and the need for two round trips to retrieve the records had a negative impact on Riak's performance for our data model. Since there is no mechanism in Riak for the server to filter and return only the most recent observations for a patient, all matching records must be returned and then sorted and filtered by the client.

MongoDB and Cassandra both provided a relatively straightforward data model mapping and both provided the strong replica consistency needed for this application. The data model mapping for MongoDB seemed more transparent than the use of the Cassandra Query Language (CQL), and the indexing capabilities of MongoDB were a better fit for this application.

5.3.3 Generate and Load Data

A synthetic data set was used for testing. This data set contained one million patient records, and 10 million lab result records. The number of lab results for a patient ranged from 0 to 20, with an average of 7.

5.3.4 Create Load Test Client

We used the YCSB framework [38] as the foundation for the test client, to manage test execution and test measurement. For test execution, we replaced YCSB's very simple default data models, data sets, and queries with implementations specific to our use case data and requests.

YCSB's built-in capabilities allow specification of the total number of operations and the mix of read and write operations in a workload. Our customer specified that the typical workload for the EHR system was 80% read and 20% write operations. For this operation mix, we implemented a read operation to retrieve the five most recent observations for a single patient, and a write operation to insert a single new observation record for a single existing patient.

In order to investigate using the NoSQL technology as a local cache (described in §5.2.2, above), we implemented a write-only workload that represented a daily operation to load a local cache from a centralized primary data store with records for patients with scheduled appointments for that day. We also implemented a read-only workload that represented flushing the cache back to the centralized primary data store.

The YCSB measurement framework measures *operation latency* as the time from when the request is sent to the database until the response is received from the database. The YCSB reporting framework aggregates latency measurements separately for read and write operations. Latency distribution is a key scalability metric for big data systems [45, 44], so we recorded both average and 95^{th} percentile values.

We extended YCSB to report overall throughput, in operations per second. This was the total number of operations performed (reads plus writes) divided by the workload execution time (from the start of the first operation to the completion of the last operation in the workload execution, and not including initial setup and final cleanup times).

5.3.5 Define and Execute Test Scripts

For each database and configuration, every workload was run three times, to minimize the impact of transient events in the cloud infrastructure. The standard deviation of the throughput for any three-run set never exceeded 2% of the average.

YCSB can use multiple execution threads to create concurrent client sessions, so for each of the three test runs, the workload execution was repeated for a defined range of test client threads (1, 2, 5, 10, 25, 50, 100, 200, 500, and 1000), which created a corresponding number of concurrent database connections. Post-processing of test results averaged measurements across the three runs for each thread count.

NoSQL databases are not typically designed to operate with a large number of concurrent database client sessions. Usually, clients connect to a web server tier and/or an application server tier, which aggregates the client operations on the database using a pool of roughly 16-64 concurrent sessions. However, since this customer was modernizing a system that used thick clients with direct database connections, they wanted to understand the feasibility of retaining the thick client architecture.

Since there were multiple concurrent connections to the database, we had to define how these were distributed across the server nodes. MongoDB uses a centralized router node, so all clients connected to that single router node. Cassandra's data center aware distribution feature created three sub-clusters of three nodes each, and client connections were spread uniformly across the three nodes in one sub-clusters. In the case of Riak, the product architecture only allowed client connections to be spread uniformly across the full set of nine nodes.

5.4 Performance and Scalability Test Results

We report here on results for the nine-node configuration that reflected a typical production system (described in §5.3.1, above). Other tested configurations included running on a single server. The single-node configuration's availability and scalability limitations make it unfeasible for production use, and so we do not present performance comparisons across databases for this configuration. However, in the following discussion, we compare the single node configuration for a particular database to its distributed configuration, to provide insights into the efficiency of distributed coordination mechanisms and guide tradeoffs to scale up by adding more nodes versus using faster nodes with more storage.

This EHR application required strong replica consistency. These results are reported first, below. This is followed by a comparison of strong replica consistency to eventual consistency.

5.4.1 Performance Evaluation—Strong Consistency

The database configuration options to achieve strong replica consistency are summarized in Table 5.1. For MongoDB, these settings cause all writes to be committed on the primary server, and all reads are from the primary server. For Cassandra, the effect is that all writes are committed on a quorum formed on each of the three sub-clusters, while a read required a quorum only on the local sub-cluster. For Riak, the effect is to require a quorum on the entire nine-node cluster for both write operations and read operations.

Table 5.1: Settings for representative production configuration

Database	**Write Options**	**Read Options**
MongoDB	Primary Acknowledged	Primary Preferred
Cassandra	EACH_QUORUM	LOCAL_QUORUM
Riak	quorum	quorum

The throughput performance for the representative production configuration for each of the workloads is shown in Fig. 5.2, Fig. 5.3, and Fig. 5.4.

In every case, Cassandra provided the best overall performance (peaking at approximately 3500 operations per second), with read-only workload performance approximately 10% better than the single node configuration, and write-only and read/write workload performance approximately 25% higher than the single node configuration. In moving from single node to a distributed configuration, we gain performance from decreased contention for storage I/O and other per-node resource. We also lose performance, due to the additional work of coordinating write and read quorums across replicas and data centers. For Cassandra, the gains exceeded the losses, resulting in net higher performance in the distributed configuration.

Furthermore, Cassandra's *data center aware* features provide some separation of replication configuration from sharding configuration. In these tests, compared to Riak, this allowed a larger portion of the read operations to be completed without requiring request coordination (i.e. peer-to-peer proxying of the client request).

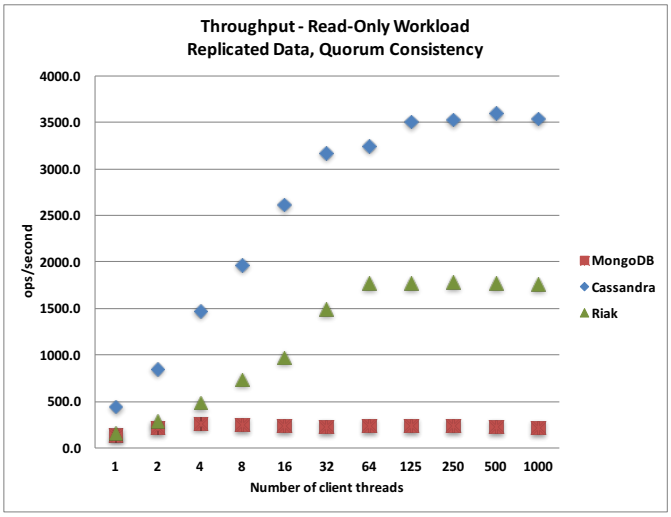


Figure 5.2: Throughput, Representative Production Configuration, Read-Only Workload (higher is better)

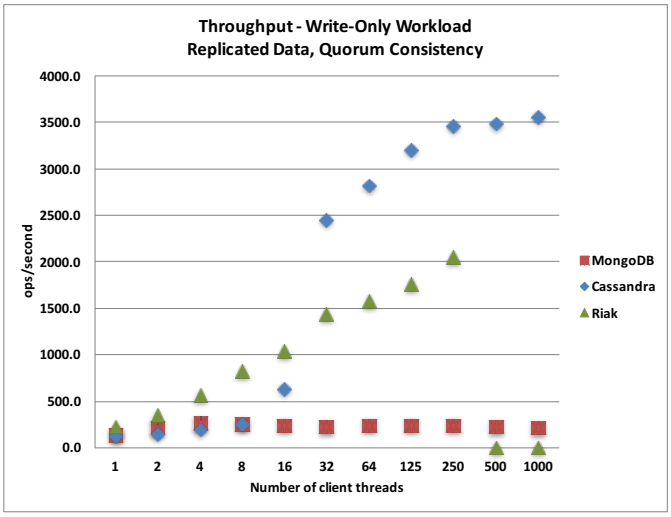


Figure 5.3: Throughput, Representative Production Configuration, Write-Only Workload

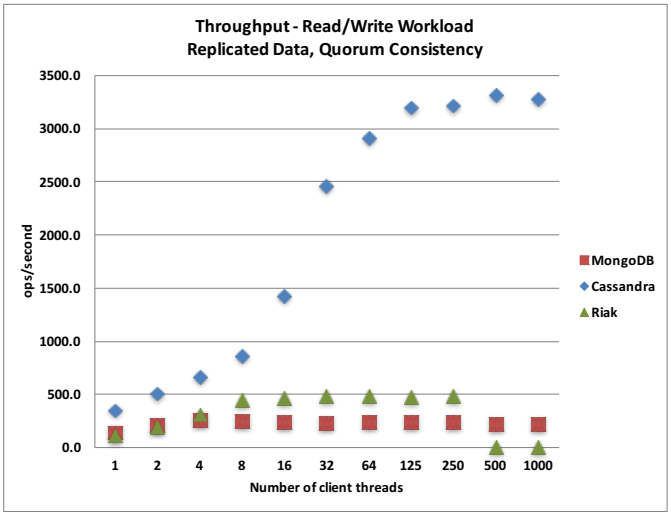


Figure 5.4: Throughput, Representative Production Configuration, Read-/Write Workload

Riak performance in this distributed configuration is approximately 2.5x better than the single node configuration. In test runs using the write-only workload and the read/write workload, our Riak client had insufficient socket resources to execute the workload for 500 and 1000 concurrent sessions. These data points are hence reported as zero values in Fig. 5.3 and Fig. 5.4. We later determined that this resource exhaustion was due to ambiguous documentation of Riak's internal thread pool configuration parameter, which creates a pool for each client session and not a pool shared by all client sessions. After determining that this did not impact the results for one through 250 concurrent sessions, and given that Riak had qualitative capability gaps with respect to our strong consistency requirements (discussed below), we decided not to re-execute the tests for those data points.

MongoDB's single node configuration performance was nearly 8x better than the distributed configuration. We attribute this to two factors: First, the distributed configuration is sharded, which introduces the router and configuration nodes into the MongoDB deployment architecture. The router proxies each request to the appropriate shard, using the key mapping stored in the configuration node. In our tests, the router node became a performance

bottleneck. Fig. 5.5 and Fig. 5.6 show read and write operation latency for the read/write workload, with nearly constant average latency for MongoDB as the number of concurrent sessions is increased, which we attribute to rapid saturation of the single router node.

The second factor affecting MongoDB performance is the interaction between the sharding scheme used by MongoDB and the write-only and read-/write workloads that we used. Both Cassandra and Riak use a hash-based sharding scheme, which provides a uniformly distributed mapping from the range of keys onto the physical nodes. In contrast, MongoDB used a range-based sharding scheme with rebalancing (see `http://docs.mongodb.org/v2.2/core/sharded-clusters/` for a discussion of MongoDB's sharding and rebalancing features).

Our workloads generated a monotonically-increasing key for new records to be written, which caused all write operations to be directed to the same shard, since all of the write keys mapped into the range stored in that shard. This is a typical key generation approach (e.g., the SQL autoincrement key types), but in this case, it focuses the write load for all new records onto a single node and thus negatively impacts performance. A different indexing scheme was not available to us, as it would impact other systems that our customer operates. (We note that MongoDB introduced hash-based sharding in v2.4, after our testing had concluded.)

Our tests also measured latency of read and write operations. While Cassandra achieved the highest overall throughput (approximately 3500 operations per second), it also delivered the highest average latencies (indicative of high internal concurrency in request processing). For example, at 32 client connections, Riak's read operation latency was 20% of Cassandra (5x faster), and MongoDB's write operation latency was 25% of Cassandra's (4x faster). Fig. 5.5 and Fig. 5.6 show average and 95^{th} percentile latencies for each test configuration.

5.4.2 Performance Evaluation—Eventual Consistency

We also performed tests to quantify the performance cost of strong replica consistency, compared to eventual consistency. These tests were limited to the Cassandra and Riak databases—the performance of MongoDB in the representative production configuration was such that no additional characterization of that database was warranted for our application. The selected write and read options to achieve eventual consistency are summarized in Table 5.2.

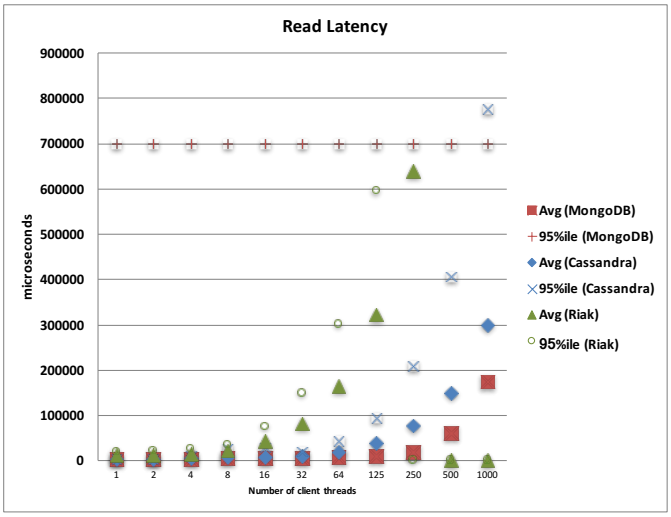


Figure 5.5: Read Latency, Representative Production Configuration, Read-/Write Workload

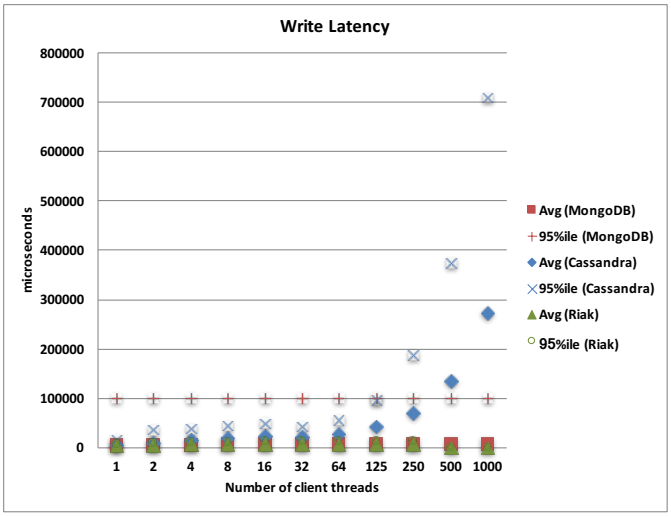


Figure 5.6: Write Latency, Representative Production Configuration, Read-/Write Workload

The effect of these settings for both Cassandra and Riak was that writes were committed on one node (with replication occurring after the operation was acknowledged to the client), and read operations were executed on one replica, which may or may not return the latest value written.

Table 5.2: Settings for eventual consistency configuration

Database	Write Options	Read Options
Cassandra	ONE	ONE
Riak	noquorum	noquorum

For Cassandra, at 32 client sessions, there is a 25% reduction in throughput going from eventual consistency to strong consistency. Fig. 5.7 shows throughput performance for the read/write workload on the Cassandra database, comparing the representative production configuration with the eventual consistency configuration.

The same comparison is shown for Riak in Fig. 5.8. Here, at 32 client sessions, there is only a 10% reduction in throughput. (As discussed above, test client configuration issues resulted in no data recorded for 500 and 1000 concurrent sessions.)

In summary, the Cassandra database provided the best throughput performance, but with the highest latency, for the specific workloads and configurations tested here. We attribute this to several factors. First, hash-based sharding spread the request and storage load better than MongoDB. Second, Cassandra's indexing features allowed efficient retrieval of the most recently written records, particularly compared to Riak. Finally, Cassandra's peer-to-peer architecture and data center aware features provide efficient coordination of both read and write operations across replicas and data centers.

5.5 Lessons Learned

Product evaluation of NoSQL databases presents a number of challenges that we had to address in the course of this project. Our lessons learned fall in two broad categories: The first category includes issues related the essential complexity of evaluating NoSQL products, and the second category includes issues that arose from the accidental complexity of the available tools and technologies.

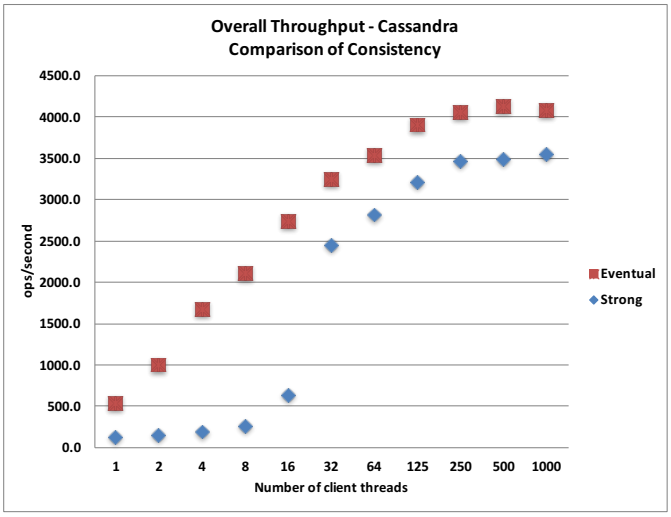


Figure 5.7: Cassandra—Comparison of strong and eventual consistency

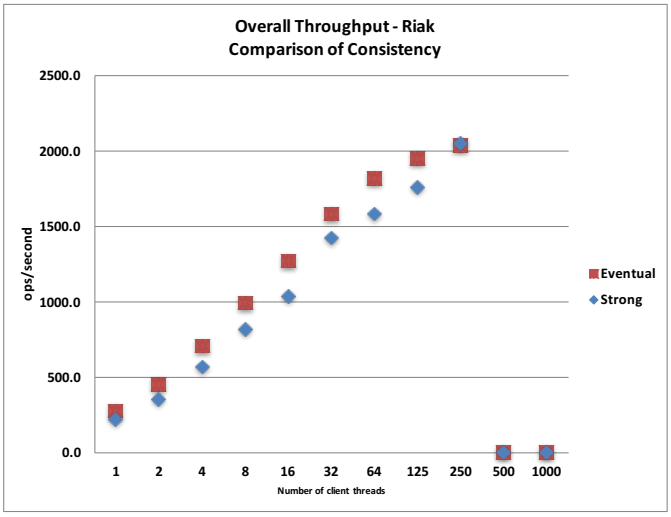


Figure 5.8: Riak—Comparison of strong and eventual consistency

5.5.1 Essential Issues

Defining selection criteria

This technology selection decision must be made early in the design cycle, and may be difficult and expensive to change [65]. The selection must be made in a context where the problem definition may not be complete, and the solution space is large and rapidly changing as the open source landscape continues to evolve.

We found that principal decision drivers were the size and growth rate of the data (number of records and record size), the complexity of the data model including the relations and navigations to support the use cases, the operational environment including system management practices and tools, and user access patterns including operation mix, queries, and number of concurrent users. We used quality attribute scenarios [9] to elicit these requirements, followed by clustering and prioritization. There were diverse stakeholder concerns, and identifying "go/no-go" decision criteria helped to focus the evaluation.

Configuration tuning

There are many configuration tuning parameters available, at the database, operating system, and EC2 level. We minimized changes to default configurations for two reasons. First, tuning can be a lengthy process, balancing interactions between settings within a layer and across layers. Second, our workload and data set were representative of, but not identical to, our production system and so optimization for our test workload would not necessarily apply to the production system. We found significant performance differences between products that would not be eliminated by tuning, and these were sufficient to make a selection.

Quantitative selection criteria

Quantitative selection criteria with hard thresholds were problematic to validate through prototyping. There are many parameters that can be tuned to affect performance, in the infrastructure, operating system, and database product itself. While the final target system architecture must include that tuning, the testing space can quickly explode during selection. We found it useful to frame the performance criteria in terms of the shape of the perfor-

mance curve. For example, is there a linear increase in throughput as the load increases? If not, are there discontinuities or inflection points within the input range of interest? Understanding the sensitivities and trade offs in a product's capabilities can be sufficient to make a selection, and also provides valuable information to make downstream architecture design decisions regarding the selected product.

Screening candidate products to prototype

We used architecturally significant requirements to perform a manual survey of product documentation to identify viable candidates for prototyping. The manual survey process was slow and inefficient; as noted earlier, the solution space is large and rapidly changing. We began to collect and aggregate product feature and capability information into a queryable, reusable knowledge base, which included general quality attribute scenarios as templates for concrete scenarios, and linked the quality attribute scenarios to particular product features. This knowledge base was reused successfully for later projects, and is an area for further research.

Tradeoff between evaluation cost and fidelity

Any COTS selection process must balance cost (in time and resources) against fidelity (along dimensions such as data set size, cluster size, and exact configuration tuning), and the rapid changes in NoSQL technology exacerbate these issues. During the course of our evaluation, each of the candidate products released at least one new version that included changes to relevant features, so a lengthy evaluation process is likely to produce results that are not relevant or valid. Furthermore, if a public cloud infrastructure is used to support the prototyping and measurement, then changes to that environment can impact results. For example, during our testing process, Amazon changed standard instance types offered in EC2. Our recommendation is to limit prototyping and measurement to just two or three products, in order to finish quickly and produce results that are both valid and relevant in this evolving context.

5.5.2 Accidental Issues

Choosing between manual and automated testing

The prototyping and measurement reported here used the Amazon cloud, which enabled efficient management and execution of the tests. Our peak utilization was more than 50 concurrently executing server nodes (supporting several product configurations), which is more than can be efficiently managed in physical hardware environments.

We had constant tension between using manual processes for server deployment and management, and automating some or all of these processes. While repeating manual tasks goes against software engineering best practices such as "don't repeat yourself"[4], in retrospect we think that the decision to make slow, but constant, forward progress, rather than stopping to introduce automation, was appropriate. Organizations that have an automation capability and expertise in place may reach a different conclusion. We did automate test execution and data collection, processing, and visualization. These tasks were performed frequently, had many steps, and had to be repeatable.

Initial database loading

Evaluation of a big data system requires that the database under test contains a large data set. Our read-intensive use cases required populating the database prior to test execution. We found that bulk or batch loading of NoSQL databases requires special attention; each database product had specific recommendations and special APIs for this function. For example, recommendations like pre-splitting the data set significantly improved bulk load performance (e.g., for MongoDB). In other cases, we found that following the recommendations was absolutely necessary to avoid failures due to resource exhaustion in the database server during the load processing. We recommend that if bulk load is not one of your selection criteria, consider taking a brute force approach to load the data once, and then use database backups, or virtual machine or storage volume snapshots to return to the initial state as needed.

[4] `http://c2.com/cgi/wiki?DontRepeatYourself`

Deleting records at completion of a test

All of our tests that performed write operations ended the test by restoring the database to its initial state. We found that deleting records in most NoSQL databases is a very slow operation, taking as much as 10 times longer than a read or write operation. In retrospect, we would consider using snapshots to restore state, rather than cleaning up using delete operations.

Measurement framework

It is necessary to understand the measurement framework used in the test client. Although YCSB is the de facto standard for NoSQL database characterization, the 95^{th} and 99^{th} percentile measurements that it reports are incorrect under certain latency distribution conditions. The YCSB implementation could be modified to extend the validity of those measurements to a broader range of latencies, or alternative metrics could be used for selection criteria.

5.6 Further Work and Conclusions

Ultimately, technical capabilities and performance are just one input to a software technology selection decision. Non-technical factors such as development and operational cost, schedule, risk, and alignment with organizational standards are also considered, and may have more influence on the final decision. However, a rigorous technical evaluation, based on prototyping and measurement, provides important information to assess both technical and non-technical considerations.

We have described a systematic method to perform this technology evaluation for NoSQL database technology, in a context where the solution space is broad and changing fast, and the system requirements may not be fully defined. Our approach was to evaluate the products in the specific context of use, starting with elicitation of key requirements to capture architecture drivers and selection criteria. Next, product documentation is surveyed to identify viable candidate technologies, and finally, rigorous prototyping and measurement is performed on a small number of candidates to collect data to make the final selection.

We described the execution of this method to evaluate NoSQL technologies for an electronic healthcare system, and present the results of our measurements of performance, along with a discussion of alignment of the NoSQL data model with system-specific requirements. We presented lessons learned from our experience on this project.

Our experience identified the benefits of having a trusted knowledge base that can be queried to discover the features and capabilities of particular NoSQL products, and accelerate the initial screening to identify viable candidate products for a particular set of quality attribute scenario requirements. This is an area for further research.

6

System-of-Systems Viewpoint for System Architecture Documentation

Summary

A system of systems (SoS) is formed by integrating existing systems, each independently operated and managed. Frequently, the architecture documentation of these existing systems addresses concerns only from a stand-alone system perspective, and must be augmented to address concerns that arise in the SoS. We evaluated the ability of an architecture documentation viewpoint to address the concerns of a SoS architect about a constituent system within the SoS, in order to support SoS design and analysis involving that constituent system. An expert panel reviewed documentation produced by applying the viewpoint to a system, using the active review method. The expert panel was able to use the documentation to answer questions related to all SoS architect concerns, except for questions related to concerns about the interaction of the constituent system with the platform and network infrastructure. The expert panel was unable to answer certain questions using the supplied artifacts because the baseline version of the viewpoint had a gap in concern coverage related to relationship of software units of execution (e.g., processes or services) to computers and networks. The viewpoint was revised to add a Deployment Model to address these concerns, and that revised viewpoint is included here in an appendix.

6.1 Introduction

A system of systems (SoS) is created by composing constituent systems. Each constituent system retains operational independence (it operates to achieve a useful purpose independent of its participation in the SoS) and managerial independence (it is managed and evolved, at least in part, to achieve its own goals rather than the SoS goals) [110]. In order to assess suitability of the system for use in the SoS and to reason about SoS functionality and quality attributes, the architect of a SoS relies on documentation about the constituent system. In an ideal world, constituent system documentation would be available and address all SoS concerns. Our previous research (discussed in Related Work below) reports that this is not usually the case [91]. The challenge of documenting architectures whose parts are designed by separate organizations is a fundamental challenge of SoS and ultra-large scale systems [122].

Pragmatically, the SoS architect seeking information about a constituent system has limited options. If there is documentation and/or source code available, the SoS architect can attempt to learn enough about the constituent system design to address concerns about how that system will operate in the SoS. However, constituent systems are developed independently, and often there is little or no documentation or code available. In that case, the architect of the SoS would seek to collaborate with the architect of each constituent system to augment the constituent system architecture documentation with the information needed to address the SoS concerns. However, the managerial independence of the development and evolution of constituent systems within a SoS [110] often creates barriers to collaboration. Consider three examples of these barriers:

1. Each constituent system owner retains independent management of funding and objectives, and the constituent system architect's responsibilities for delivering system-oriented capabilities may not provide slack time to allow collaboration with the SoS architect.

2. There is no ongoing development on a particular constituent system, and so there is no architect assigned who could collaborate with the SoS architect.

3. Firms are integrating IT systems after a merger or acquisition, and the architects of particular acquired systems may be reassigned or dismissed, and so are not available to collaborate.

In each of these scenarios, collaboration between the SoS architect and the architects of each constituent system may become a tightly planned and managed high ceremony event. The SoS architect must articulate a precise request for information, for which the constituent system architect estimates the cost to respond. The SoS owner and the constituent system owner negotiate to fund the constituent system architect's work to respond, and eventually the constituent system architect is directed to supply the requested information to the SoS architect. There is often little or no ability to iterate the requests or seek elaboration of the responses, and given these high stakes, the architect of the SoS needs a pedigreed basis for a request for information.

The contribution of this chapter is an architecture documentation viewpoint to assist SoS architects in collecting or creating sufficient documentation about constituent systems in a SoS. The viewpoint addresses stakeholder concerns about SoS design and analysis. This reusable "library viewpoint" conforms to the ISO/IEC 42010 standard for architecture description [79], and provides guidance for SoS architects to request sufficient information about constituent system architectures to satisfy SoS-level concerns about each constituent system operating in the SoS context. The viewpoint was evaluated by an expert panel in a single case mechanism experiment using the active design review method [128]. We found that the baseline version of the viewpoint covered most SoS stakeholder concerns; however, the experiment uncovered a gap in the area of deployment of software units to computers and networks. We describe how the viewpoint was reworked by adding a new model kind to address this gap.

This chapter is organized as follows: §6.2 discusses related work in the areas of SoS and architecture documentation. §6.3 describes our approach to developing and evaluating the viewpoint, which was based on Wieringa's design cycle [156]. §6.4 presents the results of our evaluation experiment, and our analysis and interpretation, including how the baseline version of the viewpoint was reworked based on the results of the experiment. §6.5 summarizes our conclusions, and the reworked viewpoint is included as an appendix.

6.2 Related Work

Generally, concern-driven architecture documentation approaches organize architecture documentation into *views* to address stakeholder concerns [79, 34, 137]. These approaches are widely used for software system architecture documentation, for example in the Rational Unified Process (RUP) 4+1 Views [101].

At the SoS level, view-based frameworks such as DoDAF [48] and MODAF [118] have emerged to document SoS architecture. The EU COMPASS Project [37], which ended in 2014, addressed SoS modeling, and SoS architecture documentation continues to be an area of active research [69], producing new documentation approaches such as S3 [27] and SysML-based approaches from the EU AMADEOS project [119].

The architect of a SoS must depend on the documentation of constituent systems. Our earlier research reported that one challenge to designing a SoS architecture is gaps in the architecture documentation of the constituent systems [91]. The architecture documentation of each constituent system usually focuses on the stand-alone operation of that system, and on the stand-alone development and evolution of that system. The architecture documentation for constituent systems is created during engineering development of the constituent system, for different purposes than SoS, notably constituent system bounds, constituent system goals, different modeling goals, and different characteristics of interest [75]. While functional interaction with external systems in support of the system's standalone operation may be addressed by the architecture documentation, the quality attribute aspects of those interactions are typically not well covered.

Participation of a constituent system in a SoS introduces new concerns about that system; however, a survey by Bianchi and colleagues found that there are no applicable quality attribute frameworks for these concerns [20]. Our earlier research found interoperability to be a primary concern of SoS designers [94], and more recent work by Batista [12] and by Guessi and colleagues [69] confirmed that interoperability is a primary focus of SoS architecture documentation. The system mission characterization of Silva and colleagues provides insight into functional interoperation concerns [145]. Architecture documentation for constituent systems that addresses standalone operation may not address SoS interoperability concerns, which go beyond interface syntax. As the context for interface semantics is expanded to the SoS, behavior

that might have been considered private to the system becomes externally visible. For example, design decisions such as whether to retry a failed request to an external system may not be architectural in the context of standalone operation, but become externally visible and architectural in the context of SoS operation.

The viewpoint that we developed could be considered an extension of the System Context Viewpoint defined by Woods and Rozanski [159], or of the system context diagram in the *Beyond Views* section of a Views and Beyond architecture document [34]. However, each of these focuses on how external interfaces and interactions support the independent operation of the system, and not on how the system interoperates with other systems to achieve an SoS capability.

6.3 Approach

Our objective is to design an artifact that contributes to the achievement of some goal. Here, the artifact is an architecture viewpoint, and the goal is to allow SoS architects to reason about a constituent system to design a SoS. Wieringa labels this a *design problem* and we used the *Design Cycle* approach [156] as follows:

1. Problem Investigation—We built on the related work discussed above to identify stakeholders in the SoS design process and their concerns related to constituent systems operating in the SoS context.

2. Treatment Design—We defined an architecture viewpoint to address those stakeholder concerns.

3. Treatment Evaluation—We evaluated the treatment by a single case mechanism experiment [156], using an expert panel to conduct an active design review [128].

6.3.1 Problem Investigation—Identify Stakeholders and Concerns

There are many stakeholders in a SoS and in its architecture [19]. Our focus is on the architecture design and analysis task, and specifically, reasoning about a constituent system in the context of the SoS, which narrowed the scope to the stakeholder roles listed in Table 6.1.

Table 6.1: Selected SoS Architecture Stakeholders

Stakeholder Name	Stakeholder Role
SoS Architect	Creates architecture designs to allow constituent systems to interoperate to achieve SoS goals. Proposes or defines necessary or desirable changes to constituent systems.
SoS Program Manager	Has ultimate responsibility for achieving SoS goals. Negotiates with program managers of constituent systems to make necessary or desirable changes to constituent systems.
Developer	Makes necessary or desirable changes to the software of the constituent systems.
SoS Testers and Integrators	Installs, configures, and tests the constituent systems interoperating as a SoS.

These stakeholders were selected because they are directly involved in understanding the constituent system architectures, proposing or defining changes to those architectures for use in the SoS, and then constructing, testing, and integrating the constituent systems in the SoS.

While our earlier systematic review found that SoS research has heavily focused on interoperability concerns [94], our state of the practice survey indicated that practitioners designing and analyzing SoS architectures have broader technical and non-technical concerns [91]. Since our treatment will be employed by practitioners, we decided to augment the researcher-oriented findings with a survey of practitioner-focused literature to identify additional concerns about constituent systems when designing and analyzing SoS architectures.

This survey focused on an annual practitioner conference organized by the Systems Engineering Division of the National Defense Industry Association (NDIA)[1]. We reviewed all papers in the SoS Track and Architecture Track for the conferences from 2009 through 2016, and identified 14 papers that discussed SoS architecture concerns. We also reviewed the United States Department of Defense Systems Engineering Handbook for Systems of Systems

[1]See `http://www.ndia.org/divisions/systems-engineering`

[125], which provides guidance to a broad community of practice. From these sources, we identified a set of concerns that are shown in Table 6.2. As described below, these concerns were used to define the architecture viewpoint artifact.

6.3.2 Treatment Design—Define the Architecture Viewpoint

Wieringa defines a *treatment* as "the interaction between the artifact and the problem context" [156, §3.1.1]. We will define an artifact—an architecture viewpoint—that will be applied by an SoS architect to create an architecture view of a constituent system that provides the information needed to reason about that constituent system when it is operating in the context of the SoS. In this section we focus on the design of the architecture viewpoint, however, our evaluation will consider the entire treatment.

The viewpoint definition conforms to the ISO/IEC/IEEE 42010 standard [79], using Annex B of that standard as the template. This approach was selected because of its status as a global standard and because it is compatible with producing documentation using other approaches such as Views and Beyond (see, for example, Appendix E in [34]).

The subset of the ISO 42010 conceptual model related to viewpoint definition is shown in Fig. 6.1.

As shown in Fig. 6.1, the viewpoint definition begins by identifying stakeholders and concerns. The concerns identified in Table 6.2 are somewhat general. In order to define an architecture viewpoint to address the concerns, we refined these by mapping them to the set of quality attributes that Bass and colleagues [10] defined and found to be relevant to all software systems, namely performance, availability, security, testability, modifiability, and usability. In Table 6.3, we consider each of these quality attributes (along with a category for context concerns that cut across stakeholders) as concerns at the SoS level, and then trace down to information needed at the constituent system level in order to address the SoS concern. This tracing was performed by considering the tactics [10] that might be applied to achieved the quality. Tactics that could be applied in the SoS context became concerns about constituent systems in Table 6.3. Bass and colleagues also discuss which stakeholders are typically concerned with each quality attribute, and we include this information in Table 6.3. Note that the SoS architect is concerned with all qualities.

Table 6.2: SoS Stakeholder Concerns from Practitioner-oriented Literature

Publication	**Concerns about constituent systems in an SoS**
Benipayo 2016 [18]	Dependencies on other constituent systems
Sitterle 2016 [146]	Interface adaptability (Interoperability), Recovery
Gump 2014 [70]	Interoperability, Dependencies on other constituent systems
Manas 2014 [115]	Dependencies on other constituent systems, Shared Resources
Carson 2014 [30]	Dependencies on other constituent systems
Guertin 2014 [68]	Portability, Scalability, Dependencies on other constituent systems, Security
Baldwin 2014 [8]	Stakeholders, Dependencies on other constituent systems
Gagliardi 2013 [56]	Shared resources
Pritchett 2013 [132]	Dependencies on other constituent systems
Dahmann 2012 [41]	Interoperability Perceived needs of constituent systems Processes, cultures, working practices between different participating organizations Dependencies at development time and run time
Dahmann 2012 [42]	Dependencies on other constituent systems
Smith 2011 [147]	Interoperability context: assumptions, constraints, drivers
Lane 2010 [102]	Monitoring and measurement
DoD 2008 [125]	Technical and organizational dependencies Interoperability Synchronization of delivery of features across constituent systems (dependencies) Constituent system stakeholders Constituent system needs and constraints Constituent system evolution strategy and built-in variabilities

Fig. 6.1 shows that the viewpoint comprises one or more model kinds. The model kinds were developed iteratively, using the following approach:

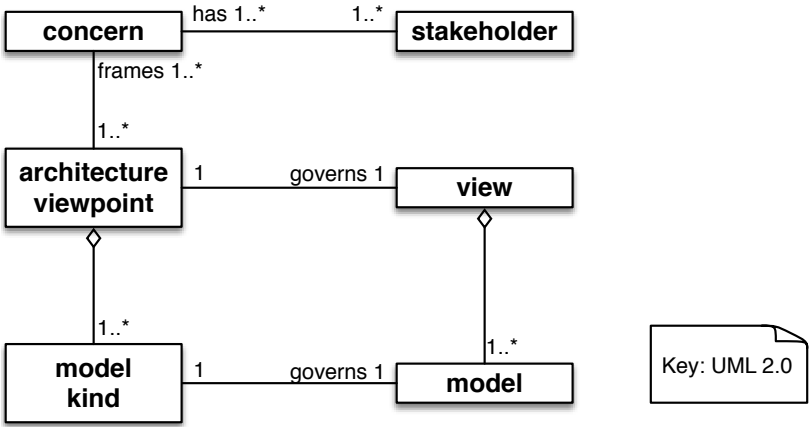


Figure 6.1: Extract from ISO 42010 Conceptual Model (Adapted from [79]

1. Identify the type of elements and relations needed to address each concern.
2. Group concerns that had the same types of elements and relations.
3. Define a model kind for each group of concerns.

Following this approach, we developed five model kinds, each addressing particular concerns from Table 6.3, and collectively addressing all concerns. The model kinds are listed below. Each model kind includes a brief discussion of the model elements and relationships, and the complete definition of the model kinds is provided in the appendix:

- Constituent System Stakeholders/Concerns–provides architecture context for the SoS architect and Program Manager by providing insight into the perceived need of the constituent system, and identifies stakeholders who may be impacted by the constituent system's operation within the SoS. The model elements are constituent system stakeholders and stakeholder concerns, with the relation *stakeholder has a concern*.

Table 6.3: Tracing SoS Concerns into Constituent Systems

SoS Concern	Constituent System Concern	Stakeholder
Performance	Shared resources: what is shared, how is use shared, behavior when insufficient resource is available (run time dependencies, monitoring and measurement, interoperability context)	SoS Architect Program Manager
Security	Authentication: identity validation repository (interoperability) Authorization: remote access to system and resources (interoperability) Encryption: algorithms and key management (interoperability)	SoS Architect Tester/Integrator Program Manager
Testability	Execution time dependencies: startup sequencing (run time dependencies) Fault detection and logging: internal (monitoring and measurement)	SoS Architect Tester/Integrator
Modifiability	Build time dependencies: COTS, FOSS, development environment, process/culture/working practices Run time dependencies on other constituent systems Variabilities: Affecting interfaces, decision model (dependencies) (evolution and built-in variabilities)	SoS Architect Developer
Availability	Fault detection and logging: interfaces (monitoring and measurement) Fault recovery (interoperability context)	SoS Architect Tester/Integrator Program Manager
Usability (for SoS operators)	Configuration dependencies among constituent systems (development time and run time dependencies)	SoS Architect
Context	Perceived needs and constraints of constituent systems Processes, cultures, working practices between different participating organizations Constituent system stakeholders	All

- Constituent System Execution Time Context—addresses concerns related to dependencies at execution time, shared resources, and to a lesser extent, provides overall context. The model elements are the constituent system and external software that the system interacts with during execution. The relations are execution time interactions, e.g., sends/receives message, call/return, read/write data, etc.

- Constituent System Code Context—addresses concerns related to implementation dependencies. The model elements are the constituent system software, and external modules (e.g., libraries, development tools, packages, etc.). The relation is *uses*.

- Constituent System Interface Information Model—addresses concerns related to semantic interoperation of data elements. The elements are information elements of interest to the SoS (e.g., a SoS that deals with geo-location might use concepts like position, elevation, and direction), and information elements in the constituent system software. Relations are *logical associations* (1-to-1, 1-to-N, N-to-M), *specialization/generalization*, and *aggregation*.

- Shared Resource Model—addresses concerns about runtime resource sharing. Elements are components in the SoS representing resources used by the constituent system and by other systems, including processor compute cycles, memory, disk space, network bandwidth, files, databases, virtual infrastructure, or physical resources such as a display, antenna, or radio frequency. Relations are *acquires/releases* and *consumes*.

Table 6.4 shows the mapping from the concerns listed in Table 6.3 to the model kinds listed above.

The 42010 standard permits three approaches to accommodate multiple model kinds:

- Define multiple independent viewpoints, with each viewpoint comprising a single model kind. We rejected this approach because the models are not independent: In our case, omitting one model kind leaves a set of concerns uncovered.

- Define a framework that comprises multiple viewpoints, with each viewpoint comprising a single model kind. The standard characterizes a framework as "establishing a common practice…within a particular domain of application or stakeholder community" [79, §4.5]. We rejected this approach for two reasons: The resulting artifact was applicable beyond a single application domain, and a framework does not inherently imply that all viewpoints are used together, and so we have the model omission issue described above.

Table 6.4: Mapping Concerns to Model Kinds

Concern (from Table 6.3)	**Model kind(s) that address the concern**
Shared resources—what is shared, how is use shared	Shared Resource Execution Time Context Deployment
Behavior when insufficient resource is available (run time dependencies, monitoring and measurement, interoperability context)	Shared Resource Interface Information Execution Time Context
Authentication—identity validation repository (interoperability)	Interface Information Shared Resource Execution Time Context
Authorization—remote access to system and resources (interoperability)	Interface Information Shared Resource Execution Time Context
Encryption—algorithms and key management (interoperability)	Interface Information Shared Resource Execution Time Context
Execution time dependencies—startup sequencing (run time dependencies)	Execution Time Context Deployment
Fault detection and logging—internal (monitoring and measurement)	Interface Information Execution Time Context
Fault recovery (interoperability context)	Execution Time Context Deployment
Build time dependencies—COTS, FOSS assumptions	Code Context
Development environment dependencies (development time dependencies, process/culture/working practices)	Code Context Deployment Stakeholder/Concerns
Variabilities affecting interfaces	Interface Information
Decision model (dependencies, evolution and built-in variabilities)	Code Context Interface Information
Configuration dependencies among constituent systems (development time and run time dependencies)	Code Context Execution Time Context Interface Information
Perceived needs and constraints of constituent systems	Stakeholder/Concerns
Processes, cultures, working practices between different participating organizations	Stakeholder Concerns
Constituent system stakeholders	Stakeholder/Concerns

- Define a single viewpoint that comprises multiple model kinds. We selected this approach because it treated the set of model kinds as an atomic unit.

The single viewpoint was titled "SoS Constituent System Viewpoint". The complete viewpoint definition that conforms to the ISO 42010 standard is presented in the appendix to this chapter (§6.6 below).

6.3.3 Treatment Evaluation—Active Design Review by Expert Panel

Treatment evaluation is "the investigation of a treatment as applied by stakeholders in the field…to investigate how implemented artifacts interact with their real-world context" [156, §3.1.5]. Our evaluation criteria were that the viewpoint provides sufficient coverage of concerns, and that a view created using the viewpoint provides sufficient detail to allow an SoS architect to reason about the constituent system operating in the context of the SoS.

The Introduction, above, described the high stakes involved in acquiring documentation about constituent systems, arising from the managerial independence of the systems. Therefore, an initial evaluation of this treatment through an observational case study [156, §17] or through technical action research [156, §19] would not be a responsible approach. We chose to perform a single case mechanism experiment [156, §18] using an expert panel. According to Hakim [71], small samples are effective to test explanations, particularly in the early stages of work. Expert panel assessment has been used by a number of researchers (e.g., Dyba [49], van den Bosch and colleagues [150], and Beecham and colleagues [16]), and so we saw this approach as both prudent and appropriate. Expert panel recruitment is discussed below in §6.3.3, and panel demographics are shown below in §6.4.1.

Our treatment artifact is architecture documentation. Nord and colleagues provide a six-step structured approach to reviewing architecture documentation [120], which we followed for our evaluation. Steps 1-5 are discussed in subsequent subsections, and the results of the review (Step 6) are discussed below in §6.4. This six-step process is comparable to the eight-step process used by Beecham and colleagues [16], collapsing multiple process steps in several places.

Step 1: Establish the Purpose of the Review

We aim to produce an artifact to guide an SoS architect to request the architecture information about a constituent system that is sufficient for the architect and other stakeholders to reason about that system in the context of a SoS.

As discussed above, our criteria for the treatment evaluation were that the viewpoint provides sufficient coverage of concerns, and that a view created using the viewpoint provides sufficient detail to allow an SoS architect to reason about the constituent system operating in the context of the SoS.

Step 2: Establish the Subject of the Review

We are evaluating the treatment: the artifact applied in context, specifically the viewpoint applied to a system to produce an architecture view. For this, we applied the viewpoint to the Adventure Builder system, which was chosen because it has an openly available architecture description that we used as the basis for constructing the view documentation.

The Adventure Builder is a fictitious company that sells adventure travel packages that bundle airline transportation, hotel accommodations, and guided activities. The system has a customer-facing website that allows customers to shop and purchase adventure packages, and a service-oriented architecture back-end that integrates with external payment processing and travel provider services.

The view documentation that was produced is available as part of the review instrument, at `https://goo.gl/V4cril`.

Step 3: Build or Adapt Appropriate Question Sets

Nord and colleagues identified several review styles:

- Questionnaire—reviewers assess the artifact using a list of questions provided by the review organizer.
- Checklist—reviewers rate the artifact using a list of yes/no questions (a special case of the questionnaire style).
- Subjective review—stakeholders also play the role of reviewer and pose questions to themselves.
- Active review—architects ask questions that require reviewers to use the subject artifact in order to answer the questions.

We chose to use an active review style, as this approach ensures that the reviewers skim the entire artifact and read some parts of the artifact in detail. It also evaluates the treatment (artifact in use), and not just the contents of the artifact.

However, we also wanted to understand how our experts would use their knowledge and experience to approach a SoS design problem, and so we incorporated a subjective review, where each expert formulated questions about a SoS design problem, and then later in the review, answered these as active review questions.

The instrument we created for the review had multiple parts (the entire instrument is available at `https://goo.gl/V4cril`):

1. Demographic information about the reviewer.
2. A narrative vignette that created a usage scenario for the architecture view. It asked each reviewer to play the role of a SoS architect tasked to integrate the Adventure Builder system into an SoS, and specified a design problem with three new SoS capabilities.
3. Subjective review questions—we asked each reviewer to record three questions that they had about the architecture of the Adventure Builder system, related to the design problem.
4. We next provided the architecture view artifact.
5. For each of the three new capabilities, we created several questions about the new SoS design, and the reviewer had to use the architecture view to answer the questions.

   Our questions were refined from the "Key Design Decisions" question list from Nord and colleagues [120, §4.5]. Two examples of the questions are shown here; the entire instrument is available at `https://goo.gl/V4cril`.

   - Does the Adventure Builder system have existing request-response interfaces with external systems? If so, what protocols/technologies are used for these interfaces?
   - The inputs to the new payment processing interface are: Card Type, Card Number, Card Expiration, and Card Security Code/CCV. Can the current Adventure Builder system provide all of these elements?

Reviewers also answered the three subjective review questions that they created earlier in the review. We asked reviewers to annotate their responses indicating the document sections that they consulted to answer each question. We wanted to limit the duration of the review exercise to one hour, and so we presented eight active review questions, which along with the three subjective review questions, required each reviewer to answer a total of 11 questions.

6. The instrument concluded by recording the amount of time that the reviewer spent, and with open-ended questions about the realism of the scenario, the contents of the architecture view, and any comments about the review exercise.

Step 4: Plan the details of the review

Nord and colleagues define three major activities in this step: Constructing the review instrument in light of constraints and intentions for reporting results; identifying actual reviewers; and planning review logistics.

Our entire review instrument is outlined above in §6.3.3. It includes the question set, reflects how we addressed review time constraints, and collects demographic and other information to support our reporting of research results. We decided to present all reviewers with the same set of eight active review questions (i.e. all participants were assigned the same treatment).

Experts were recruited from a population of experienced practitioners in the areas of SoS and enterprise system integration, who have worked in the field for several years and have been responsible for designing the architecture for several SoS. Experts were targeted to represent different backgrounds and system domains, as recommended by Kitchenham and colleagues [89]. Additionally, we sought geographic diversity and organizational diversity. We recruited eight experts from eight different organizations to participate in the review exercise, and seven accepted.

Finally, our review logistics were simple: We emailed the instrument to each reviewer, who worked independently to complete the review and return the completed instrument to us by email.

Step 5: Perform the Review

As noted above, we performed the review by sending an identical survey instrument by email to each reviewer, and receiving a completed instrument sent back to us by email.

6.4 Analysis and Results

6.4.1 Expert Panel Demographics

The expert panel demographics are presented in Table 6.5. These data were self-reported by each expert. These data were collected to verify that the invitee met the inclusion standards in §6.3.3, and to assess the diversity of experience of the panel.

6.4.2 Active Review Question Responses

Here we discuss responses to the eight active review questions that we created and which were assigned to all reviewers.

All of the participants answered the eight active review questions correctly, with some minor variations in responses due to differing interpretations of the question wording and the context. In particular, the use of the unqualified term "interface" in several questions proved confusing: This was interpreted to mean programmatic interface or user interface, or both.

Six of the seven reviewers indicated which model(s) they used to answer each question (Some reviewers used more than one model to answer a question.). These responses showed that every model was used by at least one reviewer to answer at least one question, indicating that the questions covered the breadth of the artifact.

Based on the small sample size (N=7), we hesitate to perform statistical analysis on the relationship between questions and models; however, we show the frequency of each model's use in Table 6.6.

Table 6.5: Expert Panel Demographic Information (N=7)

Variable	**Reported Value(s)**
Current position title	Software Architect (3) Chief Enterprise Architect (1) Chief Technical Officer (1) Solution Architect (1) Information Architect (1)
Current industry	Consulting services—multiple industry domains (4) Government (2) Financial services (1)
Prior industry experience (multiple responses allowed)	Software product development (4) Financial services (2) Academia (2) Consulting services—multiple industry domains (1) Government (1) Utility (1) Telecommunications (1) Transportation (1) Defense (1)
Nationality	USA (3) Netherlands (2) Brazil (1) UK (1)
Number of years of professional experience developing or integrating systems	18-40 years (Average = 30, Median = 30)
Approximate number of system integration projects worked on	Responses ranged from 8 to “more than 100”.

Table 6.6: Active Review Coverage of Models

Model Name	**Number of times used to answer active review question**
Stakeholder/Concerns	13
Execution-time Context Model	12
Code Context Model	8
Interface Information Model	22
Shared Resource Model	3

6.4.3 Subjective Questions

As discussed above in §6.3.3, in addition to the eight active review questions that we developed, we asked each reviewer to specify three questions that they thought were important for this design problem. In addition to helping triangulate to improve the quality of evaluation (discussed in the next section), it provided direct insight into an SoS architect's concerns when presented with a design problem.

Our seven reviewers posed three questions each. There was significant overlap among these 21 questions, and we clustered the questions into six categories, shown in Table 6.7.

The questions in the Platform category could not be readily answered by the reviewers using the architecture documentation provided. The viewpoint included a Code Context model kind, which could represent the dependencies on platform code modules, including application containers, operating systems and database libraries, and virtual machines. However, the viewpoint does not include a deployment model that would directly address this category of concerns by showing the relationship between the execution elements of the constituent system—processes, services, applications, etc.—to computer nodes and networks (e.g., [34, §5.2]).

The questions in the Implementation Quality/Risk category address issues that are important to understand when designing a SoS; however, this information is not part of the architecture (i.e. structures comprising elements, relationships, and properties) of the constituent system, and these concerns can be addressed by reviewing or inspecting non-architectural artifacts such as an issue tracking system or source code repository.

Table 6.7: Subjective Question Categorization

Category	**Example Questions**	**Frequency (N=21)**
Platform	What is the platform/technology stack/runtime environment used by the constituent system?	7
Data Model	What is the logical data model used in the constituent system? How is customer-identifying or user-identifying data handled?	6
Implementation Quality/Risk	What are the known problems in the constituent system? What is the development history (internal, acquired, outsourced, etc.)?	3
User Interface	What is the user interface exposed by the constituent system?	2
Functional Structure	What is the functional structure of the constituent system?	2
Architecturally-Significant Requirements	What are the quality attribute requirements for the constituent system?	1

The questions in the User Interface category could not be readily answered by the reviewers using the architecture documentation provided. Further research is needed into the underlying concern—a possible explanation is that the enterprise business system context for the vignette that we used for the exercise triggered this concern based on the expert's experiences, even though none of the desired new capabilities involved the user interface.

The question about Architecturally-Significant Requirements was not readily answered by that reviewer using the architecture documentation provided. A complete ISO/IEC/IEEE 42010-compliant architecture description would contain rationale that includes architecturally significant requirements [79, §5.8]. Our review instrument included only a subset of a complete architecture description containing the view that was the subject of the evaluation, and the view contained models with no rationale.

Questions in the Data Model and Functional Structure categories were readily answered by the reviewers using the architecture documentation provided.

6.4.4 Interpretation and Viewpoint Rework

Based on the experiment results discussed above, we found that the baseline version of the architecture viewpoint adequately covered the SoS stakeholder concerns that we had identified in our Problem Investigation. However, our expert review panel's subjective review questions uncovered three categories of concerns that we had not identified in our Problem Investigation, and were not addressed by the baseline version of architecture viewpoint.

Below, we discuss each of these categories of concerns, and how the baseline version of viewpoint definition was reworked to produce the viewpoint definition presented in the Appendix (§6.6).

Runtime Deployment Environment Concerns

The first category of concerns that was not addressed by the baseline version of the architecture viewpoint involved the runtime deployment environment of the constituent system. The subjective review questions that raised this concern covered two areas: the software platform (operating system, application server, database manager, service bus, etc.), and the physical deployment (mapping of software to compute nodes and networks).

In developing the viewpoint definition, we expected that the software platform concerns would be addressed by the Execution Time Context Model and/or the Code Context Model; however, those models in the Adventure Builder System artifact provided to the reviewers did not contain sufficient detail to address the concern. We have reworked the viewpoint definition by extending the "Elements" section of these two models to add the software platform elements to the model, and by extending the "What's it for" section to add that the model is used to answer questions such as those posed by the expert panel.

The physical deployment concerns arise from the distributed nature of a SoS. This physical distribution affects performance, availability, and possibly security and other qualities. It is necessary for the SoS architect to understand the physical deployment of the constituent system (how software is mapped

to compute and network resources) because, when the system becomes part of an SoS, those compute and network resources may be shared with other constituent systems, or may be configured differently from the constituent system's stand-alone architecture.

The Shared Resource Model definition identifies network bandwidth and compute resources as elements that may be shared, in order to address concerns about performance. In the Adventure Builder System artifact provided to the reviewers, that model was represented as a table, with deployment information provided as part of the description of each element. This presentation style did not provide sufficient detail to address the reviewer's concern. Many architecture documentation approaches define a Deployment Model, for example the Deployment Style defined by Clements and colleagues [34] or the Physical View defined by Kruchten [101]. In this model, elements are units of software execution (e.g., processes, services, etc.), and physical infrastructure (e.g., computer nodes and networks), and the relation of "executes on" maps software to physical infrastructure. This model is used to address concerns about performance, availability, and possibly security and other qualities. We have reworked the viewpoint definition to add a Deployment Model.

Implementation Quality and Risk Concerns

The second category of concerns that was not addressed by the baseline version of the architecture viewpoint involved the quality of the implementation of the constituent system, and assessing risk in integrating it into the SoS.

These concerns might be seen to intersect with several of the concerns drawn from the practitioner-oriented literature shown in Table 6.2, namely "Processes, cultures, working practices" and "Constituent System Evolution Strategy", but on the whole, they were not considered in developing the baseline version of the viewpoint.

As discussed above, it is important to understand these issues when designing a SoS. The expert panel's questions reflect common architecture approaches, such as the Risk and Cost Driven Architecture approach [130]. However, we think that this is not part of the architecture (i.e. structures comprising elements, relationships, and properties) of the constituent system, and that these concerns can be addressed by reviewing artifacts such as an issue tracking system or source code repository. We did not rework the baseline version of the viewpoint in response to this gap.

User Interface Concerns

The third category of concerns that was not addressed by the baseline version of the architecture viewpoint involved the user interface of the constituent system. As noted above, the constituent systems in a SoS are characterized by operational independence, which would imply one of two user interface integration patterns:

- *Mashup*, where the SoS user interface is developed using APIs of the constituent systems. In this case, the constituent system's user interface is not presented directly.
- *Portal*, where the user interface of a constituent system is presented in a sub-window (e.g., frame or pane) of the SoS user interface. In this case, the constituent system's user interface is presented in its entirety, without modification.

Therefore, concerns about the user interface, per se, do not appear to be generalizable SoS concerns, but there may be system-specific concerns. For example, a mashup approach would introduce concerns that would be addressed by the Execution Time Context Model and the Interface Information Model. A portal could introduce concerns about the user interface display as a shared resource, to be addressed by the Shared Resource Model.

We did not rework the baseline version of the viewpoint in response to this gap; however, this may be an area for further research.

6.4.5 Threats to Validity

Construct validity is the degree to which the case is relevant with respect to the evaluation objectives [138]. Here, our objective was to evaluate the ability of an architecture documentation viewpoint to address the concerns of a SoS architect about a constituent system within the SoS, in order to support SoS design and analysis involving that constituent system. The selection of the active review evaluation method ensured that the reviewers at least skimmed the entire document, and our recording of the sections of the document used to answer each question ensured that certain sections were read in detail. Also, the reviewer's positive comments about the realism of the review vignette

support the construct validity of the experiment. Our objective *was not* to compare this treatment (use of the viewpoint to reason about the constituent system) to other treatments (e.g., using documentation and code from the constituent system to reason about the system).

Internal validity concerns hidden factors, which is a concern when examining causal relations [138]. Our use of the active review method introduced the potential threat to internal validity that the questions created for the review may have been unconsciously influenced by our knowledge of the Adventure Builder system architecture and its documentation. We mitigated this risk by also incorporating subjective review questions: prior to reading the architecture documentation, each reviewer created three questions, and then later used the documentation to answer those questions. This use of triangulation increases the reliability of our results [138].

External validity is related to the generalizability of the results in the context of a specific population. As discussed above in §6.3.3, we chose to use an expert panel with a small sample size. According to Hakim [71], small samples are effective to test explanations, particularly in the early stages of work. By definition, our single case mechanism experiment does not support statistical generalization, and so suffers the same external validity challenges of all case study research [161]. Our total response rate (recruitment to completion) was 87.5%, from an expert panel with a diversity of experience and system domain coverage, so we believe that our findings are valid at least for SoS and constituent systems that are similar in size and scope to the SoS described in the vignette that formed the basis of our review.

6.5 Conclusions and Future Work

In this chapter, we have introduced an architecture viewpoint to address the concerns of a SoS stakeholders about a constituent system within the SoS, in order to support SoS design and analysis involving that constituent system. We evaluated this viewpoint using a single case mechanism experiment: An expert panel performed an active design review using a question set that we provided. The expert panel also created subjective questions, which provided additional insight into the concerns of a SoS architect when solving a design problem and improved the quality of our data by mitigating internal validity concerns inherent in the active review process.

The evaluation results were generally positive, with the viewpoint showing promise in providing guidance for SoS architects seeking architecture knowledge about a constituent system. However, the evaluation identified a gap in the baseline version of the viewpoint definition: It was missing a deployment model for the constituent system that shows the relationship of the software to computer nodes and networks. The viewpoint definition presented in the appendix has been reworked to reflect this change.

The viewpoint conforms to the ISO/IEC/IEEE 42010:2011 (E) standard for architecture description, and the revised viewpoint comprises five model kinds: Constituent System Stakeholders/Concerns, Constituent System Execution Time Context, Constituent System Code Context, Constituent System Interface Information Model, Shared Resource Model, and Deployment Model.

The managerial independence of constituent systems poses challenges for SoS architecture designers, and frequently the architecture knowledge acquisition process involves high stakes activities that risk damage to the architect's reputation and other consequences. Further empirical research in this area must be designed within the constraints of this context. Our results provide the confidence to evaluate this viewpoint using methods such as case study or technical action research.

6.6 Appendix: Viewpoint Definition

The viewpoint defined here is a revised version of the baseline viewpoint that was used to create the artifact that was the subject of the experiment discussed in the body of this chapter. The following revisions were made to the baseline version of the viewpoint, as described in §6.4.4:

- The "Elements" sections of the Execution Time Context Metamodel (Table 6.10) and the Code Context Metamodel (Table 6.11) were revised to specify that platform elements such as operating system, application server, and database manager should be included.
- A new metamodel was added. The Deployment Metamodel (Table 6.14) relates software units of execution (e.g., processes or services) to the execution environment of computers and networks. Table 6.15 and Table 6.16 were revised to add a reference to the new Deployment Metamodel.

This viewpoint definition follows the template in Annex B of ISO 42010 [79].

6.6.1 Viewpoint Name

This defines the "SoS Constituent System Viewpoint", for use in documenting the relevant parts of the architecture of one constituent system in a SoS.

6.6.2 Viewpoint Overview

The need for this viewpoint is discussed in §6.1 of this chapter.

6.6.3 Concerns Addressed by this Viewpoint

Table 6.2 and Table 6.3 in the body of this chapter show the concerns addressed by this viewpoint, and map the concerns to the stakeholder roles identified in the next section, below. §6.3.3 also discusses the method used to identify the concerns.

6.6.4 Typical Stakeholders

The stakeholder roles addressed by this viewpoint are shown in Table 6.1 in the body of this chapter.

These stakeholders were selected because they are directly involved in understanding the constituent system architectures, proposing or defining changes to those architectures for use in the SoS, and then constructing, testing, and integrating the changed constituent systems in the SoS.

6.6.5 Model Kinds/Metamodels

This viewpoint specifies of a number of model kinds[2].

We apply the principle of separation of concerns, and so each model kind is defined using a single architecture style [34]: module styles address development time concerns, component and connector styles address execution time concerns, and allocation styles map between software elements and their environment.

[2] In the terminology of ISO 42010, a *viewpoint* applied to a system yields a *view*. Analogously, the standard defines a *model kind*, which, when applied to a system, yields a *model*.

Each model kind is specified as a metamodel. The metamodel template is shown in Table 6.8, and is based on the Style Guide Template defined by Clements and colleagues [34].

Table 6.8: Template used to specify metamodels for model kinds in this viewpoint

Name:	**Name of the model kind**
Type:	Module, component and connector, or allocation, as defined by Clements and colleagues [34].
Elements:	The types of elements allowed in this model kind, and the properties that should be attached to each element instance.
Relations:	The types of relations among elements allowed in this model kind, and the properties that should be attached to each relation instance.
Constraints:	Any model construction constraints, such as cardinality of element or relation types or topology constraints.
What's it for:	Brief description of how the model kind is used to support SoS architecture tasks such as design, analysis, evolution, or evaluation.
Notations:	Recommended notations for documenting the model kind, such as table, diagram, or list.

The first model kind is defined in Table 6.9, and represents the stakeholders and their concerns for the constituent system. This provides architecture context for the SoS architect and Program Manager by providing insight into the perceived need of the constituent system, and identifies stakeholders who may be impacted by the constituent system's operation within the SoS.

The second metamodel in this viewpoint, shown in Table 6.10, addresses concerns related to dependencies at execution time, shared resources, and to a lesser extent, overall context.

Concerns related to development time are addressed in the metamodel defined in Table 6.11.

The metamodel defined in Table 6.12 addresses general information interoperation concerns.

Concerns about resource sharing are addressed in the metamodel defined in Table 6.13.

Table 6.9: Constituent System Stakeholders/Concerns Metamodel

Name:	**Constituent System Stakeholders/Concerns**
Type:	Allocation
Elements:	Constituent system stakeholders Stakeholder concerns about system architecture
Relations:	A stakeholder has a concern
Constraints:	Stakeholders can have multiple concerns. Multiple stakeholders can have the same concern.
What's it for:	Aids in understanding the scope of the constituent system, and who will be impacted by changes made to the constituent system to allow it to join the SoS.
Adding Assumptions:	List any stakeholders that were considered but intentionally excluded. Note concerns that were identified but not addressed by the architecture.
Notations:	List—one item per stakeholder, with list of concerns. Matrix—one row per stakeholder, one column per unique concern, "x" at row-column intersection means that the stakeholder in that row has the concern in that column

Table 6.10: Constituent System Execution Time Context Metamodel

Name:	**Constituent System Execution Time Context**
Type:	Component and Connector
Elements:	Running system External software that the system interacts with
Relations:	Any interaction at execution time (e.g., sends/receives message, call/return, reads/writes data, interrupts, synchronizes with) Property: Interfaces used for the interaction on self and external software Property: Direction of interaction (initiated by constituent system or external system)
Constraints:	An interface on the constituent system may be used to interact with multiple external systems Multiple external systems may interact with the constituent system through the same interface on the constituent system
What's it for:	Aids in understanding the scope of the constituent system to analyze the impacts of necessary or desired changes Identifying viable SoS subsets and activity sequencing during SoS integration
Adding Assumptions:	Startup behavior should be documented, using a notation such as a message sequence diagram Monitoring and performance measurement behavior should be documented, using notations such as message sequence diagrams and state transition diagrams.
Notations:	Diagram—e.g., Context Diagram from Clements [34] List—one item per constituent system interface, with list of external systems and interfaces that it interacts with Matrix—rows are interfaces on the constituent system, columns are interfaces on external systems, "S" at a row-column intersection means that the constituent system interface sends an interaction to the external system, "R" means that the constituent system interface receives an interaction from the external system

Table 6.11: Constituent System Code Context Metamodel

Name:	**Constituent System Code Context**
Type:	Module
Elements:	Constituent system software External modules (libraries, packages, development tools, etc.) that the constituent software depends on
Relations:	Uses Properties: type of dependency (e.g., code generation, build, unit test, integration test), version identification or key features used for external modules, source of external modules (e.g., FOSS, COTS, GOTS)
Constraints:	Many-to-many
What's it for:	Aids in understanding the scope of the constituent system to analyze the impacts of necessary or desired changes. Identifying mismatches among external dependencies that will constrain deployment decisions or interactions among constituent systems in the SoS.
Adding Assumptions:	What evolution is assumed for the external modules? Are there new features or capabilities that are expected to be available that the constituent system will use?
Notations:	Diagram—e.g., Uses Context Diagram from Clements [34] List Matrix—This structure may be documented in a Dependency Structure Matrix generated for static analysis of the constituent system code.

Table 6.12: Constituent System Interface Information Metamodel

Name:	**Constituent System Interface Information Model**
Type:	Module
Elements:	Information elements of interest to the SoS (e.g., a SoS that deals with geo-location might have concepts like position, elevation, and direction) Information elements in the constituent system software architecture Properties: Should include units, timeliness, precision, security level, etc., as applicable
Relations:	Between SoS and constituent system information elements, and from constituent system elements to sub-elements (to refine details). Logical associations (1-1, 1-n, n-m) Specialization/generalization (is-a) Aggregation
Constraints:	None
What's it for:	Understanding how common concepts in the SoS are represented in a constituent system, and identifying mismatch between representations among constituent systems in the SoS
Adding Assumptions:	Explicitly identify SoS information elements that have no relationship to the constituent system
Notations:	Logical data modeling notations (ERD, UML)

Table 6.13: Shared Resource Metamodel

Name:	**Shared Resource**
Type:	Component and Connector
Elements:	Component(s) representing a resource that is used by the constituent system and by other external systems. These include processor computing cycles, memory, disk space, network interfaces, network bandwidth, files, databases or repository, virtual infrastructure, and system physical resources such as a display, radio frequency, or antenna. Component(s) in the constituent system that use the shared resource
Relations:	Any interaction during execution that acquires, consumes, or releases the shared resource.
Constraints:	None
What's it for:	Analyzing capacity and performance/availability of the SoS. Identifying cases of undesirable SoS behavior due to mismatch between resource sharing approaches of constituent systems.
Adding Assumptions:	Is the resource explicitly or implicitly acquired and released? What is the behavior if insufficient (or no) resources are available?
Notations:	Static diagrams—e.g., from Views and Beyond component and connector style guide [34] Behavior diagrams—message sequence charts, state transition diagrams, etc.

Concerns about deployment of software onto computers and networks are addressed in the metamodel defined in Table 6.14.

6.6.6 Correspondence rules

There are no specific correspondence rules for the models constructed using this viewpoint.

6.6.7 Operations on views

Creating a view of a constituent system using this viewpoint

In some cases, the information needed to create a view using this viewpoint already exists in the architecture documentation for the constituent system. Table 6.15 and Table 6.16 map the information required for this viewpoint to sources in two commonly used documentation frameworks: Views and Beyond [34], and DoDAF [48].

Interpretive, Analysis, and Design Methods

These operations on a view created from this viewpoint are discussed in the "What's it for" section of the metamodels specified above.

6.6.8 Examples and Notes

The evaluation instrument available at `https://goo.gl/V4cril` provides an example of applying this viewpoint to create a view on a constituent system.

Table 6.14: Deployment Metamodel

Name:	**Deployment**
Type:	Allocation
Elements:	Software units of execution, with properties that specify the execution needs and constraints Computers and networks that execute software, with properties that specify the execution resources (e.g., compute cycles, memory, storage, and bandwidth) provided
Relations:	Software units execute on computers and networks.
Constraints:	None
What's it for:	Analyzing performance and availability, and possibly security and other qualities. Assessing operating cost (e.g., required hardware and software, operations staff skills).
Adding Assumptions:	Can any part of the execution environment be virtualized? What are the assumptions about the network's ability to reach the internet or particular resources on a private network?
Notations:	Table—Rows are instances or types of software elements, columns are instances or types of computers and networks Diagram—e.g., Deployment Diagram from Clements [34])

Table 6.15: Mapping from SoS Viewpoint Metamodels to Views and Beyond Approach

Viewpoint Metamodel Name	**Source of information in a Views and Beyond architecture document**
Constituent System Stakeholders/Concerns	Information Beyond Views—Documentation Roadmap. Stakeholder/View Matrix (typically generated by the architect but not explicitly included in the architecture documentation).
Constituent System Execution Time Context	Context diagram from one of the component and connector views, e.g., client-server, SOA, pipe and filter, or publish-subscribe.
Constituent System Code Context	Context diagram from a module uses view.
Constituent System Interface Information Model	Interface documentation for externally-visible interfaces (from component and connector views), or a data model view packet focused on externally-visible information elements.
Shared Resource	Component and connector view.
Deployment	Deployment view primary presentation or context diagram.

Table 6.16: Mapping from SoS Viewpoint Metamodels to DoDAF

Viewpoint Model Name	**Source of information in a DoDAF architecture document**	**Comments**
Constituent System Stakeholders/Concerns	AV-1 Overview and Summary Information PV-1 Project Portfolio Relationships	DoDAF does not generally treat stakeholders as a first-order concern. These DoDAF views provide insight into the operational, maintenance, and development stakeholders.
Constituent System Execution Time Context	SvcV-1 Services Context Description SvcV-3b Services-Services Matrix	
Constituent System Code Context	SvcV-1: Services Context Description	If the information is included, it is most likely to appear in the SvcV-1.
Constituent System Interface Information Model	SvcV-2: Services Resource Flow Description SvcV-6: Services Resource Flow Matrix StdV-1 Standards Profile	DoDAF use the concept of "resource flows" to identify interfaces and protocols.
Shared Resource	SvcV-3b Services-Services Matrix SvcV-10c Services Event-Trace Description	Shared resources may not be explicitly identified, but can be discovered using the SvcV views.
Deployment	SvcV-1 Services Context Description SvcV-3a Systems-Services Matrix	DoDAF "services" usually include both software and hardware elements, without explicit refinement. The DoDAF views noted here may provide insight, but are unlikely to provide all the information needed to create this model.

7

Runtime Performance Challenges in Big Data Systems

Summary

Big data systems are becoming pervasive. They are distributed systems that include redundant processing nodes, replicated storage, and frequently execute on a shared "cloud" infrastructure. For these systems, design-time predictions are insufficient to assure runtime performance in production. This is due to the scale of the deployed system, the continually evolving workloads, and the unpredictable quality of service of the shared infrastructure. Consequently, a solution for addressing performance requirements needs sophisticated runtime observability and measurement. Observability gives real-time insights into a system's health and status, both at the system and application level, and provides historical data repositories for forensic analysis, capacity planning, and predictive analytics. Due to the scale and heterogeneity of big data systems, significant challenges exist in the design, customization and operations of observability capabilities. These challenges include economical creation and insertion of monitors into hundreds or thousands of computation and data nodes; efficient, low overhead collection and storage of measurements (which is itself a big data problem); and application-aware aggregation and visualization. In this paper we propose a reference architecture to address these challenges, which uses a model-driven engineering toolkit to generate architecture-aware monitors and application-specific visualizations.

7.1 Introduction

The exponential growth of data in the last decade has fueled rapid evolution in the scale of software systems. Internet-born organizations such as Google and Facebook are at the cutting edge of this scale-driven revolution, collecting, managing, storing, and analyzing several petabytes of new data every day, and operating some of the largest data repositories ever constructed [154].

Beyond the Internet companies, data-intensive systems and big data applications are becoming pervasive across a wide range of business and scientific domains. For example,

- Modern commercial airliners produce approximately 0.5TB of operational data per flight [53]. This data can be used to diagnose faults, optimize fuel consumption, and predict maintenance. Airlines must build scalable systems to capture, manage, and analyze this data to improve reliability and reduce costs.

- Big data analytics for healthcare could save an estimated $450 billion in the USA [67]. Analysis of petabytes of data across patient populations, taken from diverse sources such as insurance payers, public health, and clinical studies, can extract new insights for disease treatment and prevention, and reduce costs by improving patient outcomes and operational efficiencies.

- In operation, the Square Kilometre Array telescope will generate 1TB/sec of pre-processed data, which results in one exabyte of data every 13 days. Even with significant aggregation, this system will need a sophisticated data archive and distribution system to delivery exabytes of observation data to astronomers[1].

- By one estimate, there are 14,000 million "things" with sensors, generating data that is communicated over the Internet. By the year 2020, this Internet of Things (IoT) will generate 4 zettabytes of data per year supporting automation monitoring, and optimization of processes and services around the world [149].

[1] See `https://www.skatelescope.org/software-and-computing`

With systems at these immense scales, meaningful design time performance prediction becomes essentially infeasible, both theoretically and pragmatically. Building models to represent complex static and dynamic component compositions in both the system and the underlying infrastructure, exploiting multiple architecture styles, and accurately representing combinations of heterogeneous and uncertain workloads challenges the state of the art in performance modeling. Pragmatically, even if it were possible to build such models, post-deployment data growth, shared cloud-based infrastructures, and rapid application evolution would render model results invalid more or less immediately.

For these reasons, assuring runtime performance at scale must be based on observing and analyzing actual application behavior. Observability provides real-time insights into system health and status, both at the infrastructure and application level, and provides historical data repositories for forensic analysis, capacity planning, and potentially for predictive analytics based on statistical techniques.

This paper discusses the challenges of building massively scalable, easily configurable and lightweight observability solutions that can form the basis of performance monitoring and analysis solutions. In response to these challenges, we propose and outline an approach for observability based on a model-driven toolkit that is the focus of our current research.

7.2 Characteristics of Big Data Systems

The major runtime elements of a typical big data system are shown in Fig. 7.1. Inputs from sensors, devices, humans, and other systems are delivered to the system, where they are ingested through a pipeline that translates and transforms the data and writes it to a persistent store. The persistent store is frequently *polyglot*, employing a heterogeneous mix of SQL, NoSQL, and NewSQL technologies [139]. Users query stored data through many types of devices and applications. Some applications are within the system (i.e. under the same design and operational authority as the system), while other applications may be independent of the system and integrated through various endpoint mechanisms. The system executes in a cloud infrastructure, shared with other systems.

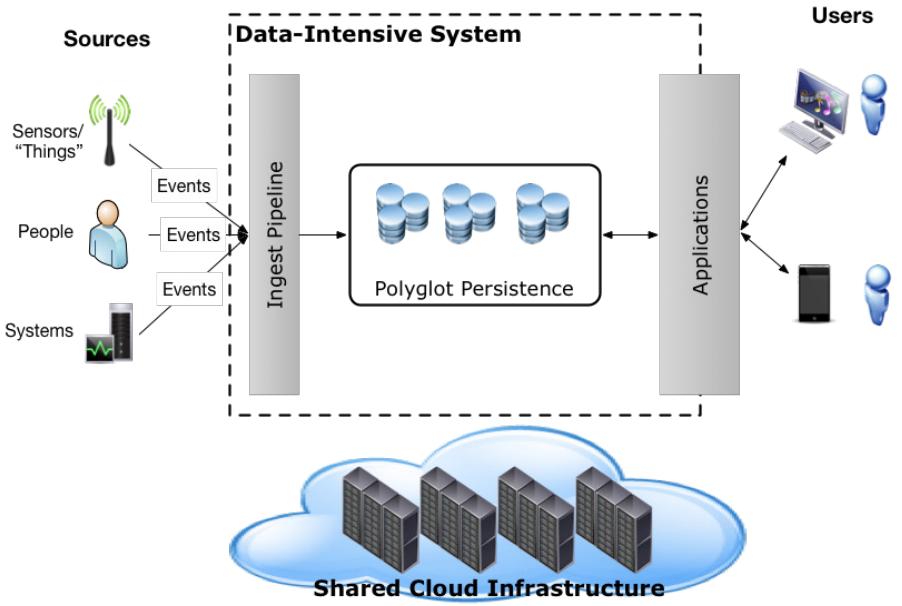


Figure 7.1: Typical Data-Intensive System

Big data systems are typically horizontally scaled distributed systems, operating over clusters of hundreds or thousands of compute and storage nodes [152]. Requests to read and write data fan out to many nodes, executing pipeline and/or parallel topologies, resulting in highly variable response latency [44].

In this context, servers are treated as "cattle, not pets" [21], meaning that nodes are expected to fail, and to be casually replaced with new instances. Redundant processing and replicated storage capabilities shift the primary concern to the herd (cluster), not to any individual (single server). This strategy requires applications to operate with partial failure as a normal condition [152].

7.3 The Need for Observability

In the environment presented in Fig. 7.1, design time predictions of system performance, produced through modeling or prototyping, can help to develop and evaluate the system's architecture and components. However, design time prediction is insufficient to assure the runtime performance of these large-scale systems in production for two reasons. First, at such scale, high fidelity prototyping and model validation is not practical. The time and cost to generate petascale data sets, and the cost and complexity of thousand-node server clusters with millions of clients requires simplification and approximate characterization of the actual architecture styles and technologies to be used.

Furthermore, after the system is deployed and in production, there is typically no control over the number of input sources and associated data rates (unless the system simply discards input data). This means the size of data being managed and analyzed quickly grows beyond predicted volumes, rendering predictions unreliable. There is also no control over the number of users and external applications, and the queries they make (unless the system rejects requests). Finally, a shared cloud infrastructure may not provide the expected continuous quality of service due to contention for resources at all levels (storage, network, and processor) and due to infrastructure failures [13, 158].

These challenges of managing and evolving system performance must therefore be addressed by instrumenting the system to observe runtime behavior, and by aggregating the collected performance measurements to identify trends and issues. In production, there is a close relationship between performance and system health. For example, in a distributed system with asynchronous communication, high response latency is generally indistinguishable from node failure or network partition. In addition, due to the fan out inherent in processing individual requests, many nodes will contribute to the latency of any particular response. In systems with redundant processing capacity and replicated storage, a sequence of identical requests may be executed by different sets of nodes, with each set inevitably having different composite performance/health characteristics. This leads to a complex relationship between the performance/health of an individual node and end-to-end system performance [44].

Redundancy also makes it possible for a system to meet its service level requirements even when experiencing partial system failures. Seemingly catastrophic numbers of node failures may not impact system performance if the failures occur when the system contains spare capacity [158]. In scalable systems, components are often designed to *fail fast* and employ stateless designs where possible to facilitate rapid recovery [124].

The performance data collected from the running system serves multiple purposes. These include operational monitoring, ongoing tuning, and preventative actions, such as adding capacity to handle load surges. These measurements also support system and architecture evolution, serving as design time prototypes for the next generation of the system. Finally, they may be used as part of an emerging autonomic capability [86].

In any runtime monitoring function, there are many common concerns. These include the intrusiveness of the monitoring on the host performance (measured in additional processor and memory utilization), measurement storage footprint (including retention and archiving), and user interface usability. However, in large-scale big data systems, these concerns become challenges that have not been solved in practice, except for point solutions that are highly customized for a specific business need [153] and have required massive investments in time and effort.

Hence, in the dynamic, uncertain runtime environments in which big data systems exist, some specific challenges of observability include:

- Monitoring a heterogeneous deployment at scale (1000s of nodes), with an ever-changing dynamic component configuration and load. While each component may have its own dedicated monitoring interface, nothing exists that can seamlessly integrate metrics from each component into the same monitoring framework. This forces every organization to develop and maintain a custom observability solution to handle the specific components they have composed into their systems.

- Collection, storage, and compression of millions of time series metrics—a big data problem in itself—with acceptable overhead and footprint

- Aggregating individual metrics into meaningful application performance indicators (e.g., total request load on all database nodes per second in a heterogeneous database environment with 1000s of individual nodes)

- Analytical environments supporting user-driven data exploration and analysis, as well as statistical techniques that can find patterns and trends in low level performance metrics.

Collectively, these challenges represent a major area of research for the software performance community. Currently, no open source or commercial technology exists that organizations can acquire, easily customize, and deploy to observe their big data systems [153]. This is a major problem given that the inherent complexity of building a low overhead and robust observability solution is a cost that most organizations are not prepared for as they scale up their systems. Creating a solution requires scalable and flexible mechanisms for observability, as well as new performance analysis techniques, packaged into a framework that organizations can rapidly customize to monitor their systems' performance.

7.4 Related Work

Relevant prior work spans several areas. These include measurement collection, visualization, architecture-aware modeling, and model-based monitoring automation.

There has been significant prior work on collecting general measurements of resource utilization at process and node level. This has produced open source packages such as Collectd (http://www.collectd.org), Ganglia [116], and Nagios [76]. Ganglia and Splunk (http://www.splunk.com) support collection of host-level measurements across clusters, and provide basic monitoring and visualization dashboards. Commercial products from HP (https://goo.gl/t3BK3y), IBM (http://www.ibm.com/software/tivoli), and others also provide similar collection and visualization capabilities. Tools such as Chukwa (http://wiki.apache.org/hadoop/Chukwa) and Sawzall (http://research.google.com/archive/sawzall.html) focus on general analytics on collected log data, including semantics for time series data sets. However, these tools are not designed to enable capture and analysis of detailed application-level performance metrics. They also have limited coverage for the heterogeneous components that commonly comprise big data systems [97], and hence have limited utility for a scalable and comprehensive observability framework. Experimental tools such as Otus [134], have demonstrated how to build on these basic collection capabilities to perform architecture-aware measurement and analysis. Otus is, however, limited to support only the Hadoop

MapReduce technology. In the visualization of large-scale system health and performance, work by Yin and colleagues take a novel approach inspired by video games to enable navigation through a complex data landscape [160]. Their focus is on infrastructure-level measurement data; however, the approach may be extensible for other types of measurements. The Theia system [58] provides architecture-specific visualization for Hadoop-based systems.

Architecture-aware modeling based on architecture styles traces back to some of the earliest work in software architecture [144]. More recent work such as Rainbow [60] uses architectural styles to model and generate a runtime framework focused on dynamic adaptation. The Rainbow framework uses measurement probes, which may include monitoring performance. However, the probes must be built into the components of the system, and the generation focuses on style-based reaction strategies when a probe's measurement crosses a threshold.

Finally, there has been very little work on using model-driven approaches to generate monitors. He and colleagues present an approach that takes steps in this direction [22]. They present a model-driven approach to composing monitors, synthesizing a compatible metamodel and then transforming heterogeneous monitors into that common metamodel. The approach does not leverage knowledge of architectural styles in the transformation, and generates only monitors, without aggregations, a measurement persistence schema, or visualizations.

7.5 Our Approach

We are creating a solution that uses generation and automation to address the runtime performance monitoring challenges of big data systems. A model of the big data system is created and model-driven approaches are used to generate monitors and visualizations. These plug into a runtime framework that automates their deployment and the collection, aggregation, storage, and display of the performance data.

This solution comprises three elements to address the challenges of scale in big data systems. These are:

1. A model-driven design time toolkit for formally specifying the observability requirements for a system, based on an architecture-aware metamodel.

2. An extensible, customizable measurement framework that forms the core of the observability runtime architecture. This includes the distributed metric collection and aggregation framework, adaptors to monitor off-the-shelf components, and a data model for storing time series-based metrics.

3. A visualization toolkit that uses novel metaphors to visualize massive amounts of measured data from executing systems.

Fig. 7.2 shows how these solution elements are used together. Our current focus is on elements (1) and (2), which we expand upon below. Element (3) is discussed briefly in Section 7.6, as future work.

7.5.1 Model-Driven Design Time Toolkit

Our model-driven toolkit is based on a metamodel for big data systems that links the runtime functional structure of the system with the observability and analysis framework discussed in the next section. The metamodel precisely represents, in terms of components, properties, relationships and constraints, the syntax and semantics of an observability framework for scalable, big data architectures.

Fig. 7.3 shows some of the key concepts in the metamodel. The elements represent the common, reusable components that form an observability framework. The Observability element is the root of the metamodel, and has a collection of properties that must be specified in a system model to configure the basic observability framework behavior. These properties include, for example, the data model for measurements capture, default behavior for handling failures, and storage location for the collected measurements.

The metamodel also contains elements that can be used to specify the architecture styles that a system utilizes. An architectural style (also referred to as an architectural pattern) defines a family of related systems, typically by providing a domain-specific design vocabulary together with constraints on relationships among the design elements. System-specific components, such as those that execute business logic, data transformations, and analytics, typically are based on architecture styles, and in a large-scale system, multiple architecture styles will be instantiated. A style's metamodel elements

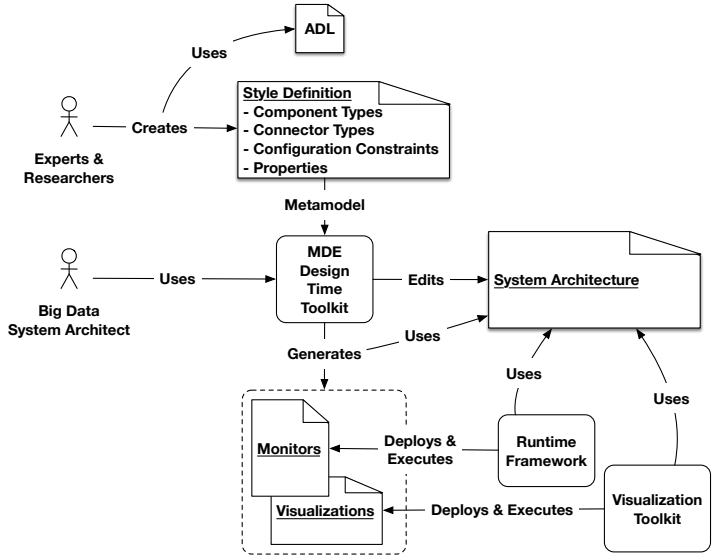


Figure 7.2: Toolset Workflow

have associated properties that a system model must specify. For example, a *MapReduce Style* element has properties that specify the framework type (e.g., Hadoop), job names to monitor, location of the cluster, and end-to-end metrics to capture.

Assets represent the components that comprise a big data system. The selection of an architecture style constrains the types of assets that can be instantiated in a model, and also constrains the topology of connections among the assets. For example, in the MapReduce style, allowable assets include *Source Data Store*, *Mappers*, *Reducers*, and an optional *Destination Data Store* (used only if the result will be persisted).

Asset definitions for a number of off-the-shelf packages will be included in the initial solution, and extensibility to represent custom-developed packages will be provided. A general asset element has properties that include the set of metrics that can be captured from a given asset. Metric properties are expressed in categories that we have defined to mask the differences in terminologies used across different asset types. For example, with data base

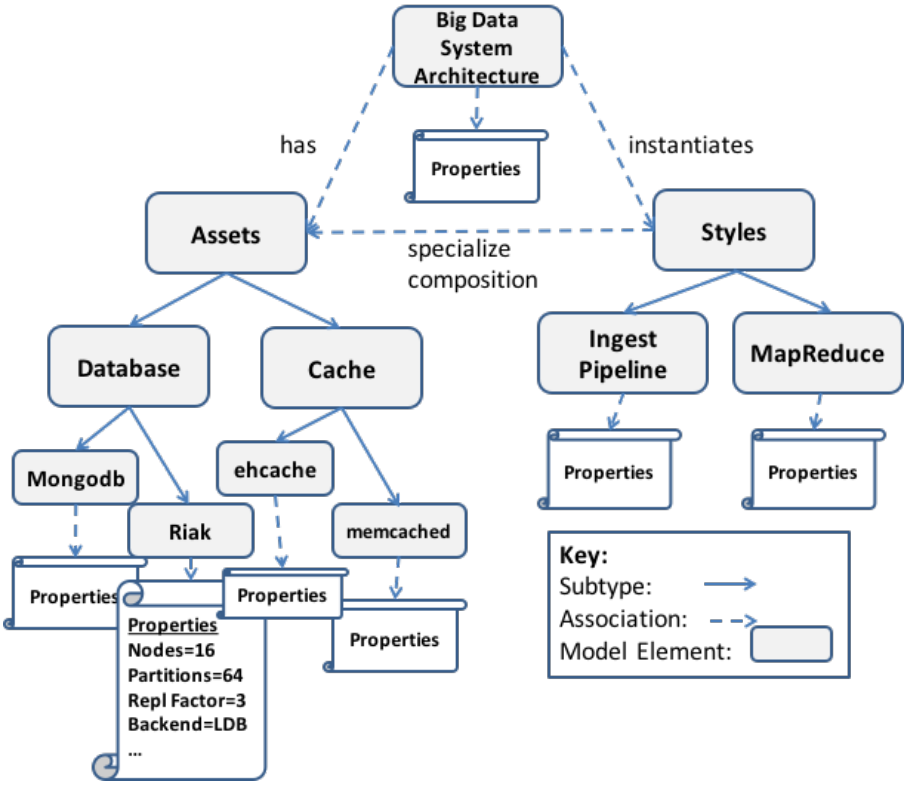


Figure 7.3: Metamodel Key Concepts

assets, properties are described in terms of categories representing database characteristics (e.g., size, number of replicas, data collections), database access (e.g., read and write performance), and host characteristics (e.g., memory usage, CPU load).

Exploiting architectures styles in the metamodel makes our modeling toolkit "architecture aware", as knowledge of the *architectural style* is used to identify the role of runtime component in the system [87]. We are utilizing the Acme architecture description language (ADL) [61] to create a formal description of the semantics of components and connectors within a particular architectural style, and the composition constraints for creating configurations (systems) that conform to the style. However, the solution does not depend

on specific features of Acme, and other ADLs that allow representation of runtime components and connectors (e.g., AADL) could be used. The formal description also specifies the properties that should be exposed as performance metrics by elements of the metamodel that conform to the style. As shown in Fig. 7.2, experts or researchers use an ADL to create the formal definition of an architectural style. This formal definition then becomes part of the metamodel for the model-driven toolkit. This approach has the strength of being extensible, enabling new architecture styles to be formally defined and incorporated into the metamodel.

As shown in Fig. 7.2, based on this metamodel, a system architect uses our model-driven toolkit to select and customize elements to create a system-specific model. We are utilizing an Eclipse Modeling Framework (EMF) tool chain (`http://www.eclipse.org/emf`) to specify the metamodel, build the model editor, and generate custom observability framework code. The EMF provides an extensible platform for both the architecture-aware design time editor and the monitor and visualization generators. The architectural style defined using the ADL is transformed into a representation in the EMF Ecore metamodel, which can then be used by the EMF.Edit and EMF.Codegen frameworks. EMF.Edit supports construction of a graphical editor for use by big data system architects to describe an architecture that is an instantiation of a particular architecture style. After the architecture is described using EMF.Edit, EMF.Codegen will generate monitors compatible with the runtime framework, and visualizations compatible with the visualization toolkit's dashboard.

7.5.2 Monitoring and Analysis Runtime Framework

The generated monitors plug into the MDE toolkit's runtime framework. The main elements of the runtime framework are shown in Fig. 7.4. The runtime framework supports deployment of monitors to thousands of nodes, and provides services for measurement collection from monitors, sampling and compression (as needed), measurement storage and archiving, and the visualization dashboard.

Monitors collect node-level performance metrics. The metamodel identifies the node-level metrics to be collected for a particular architecture style and asset type. Based on the property values specified in the specific system model, appropriate monitors are generated by the MDE Design Time Toolkit.

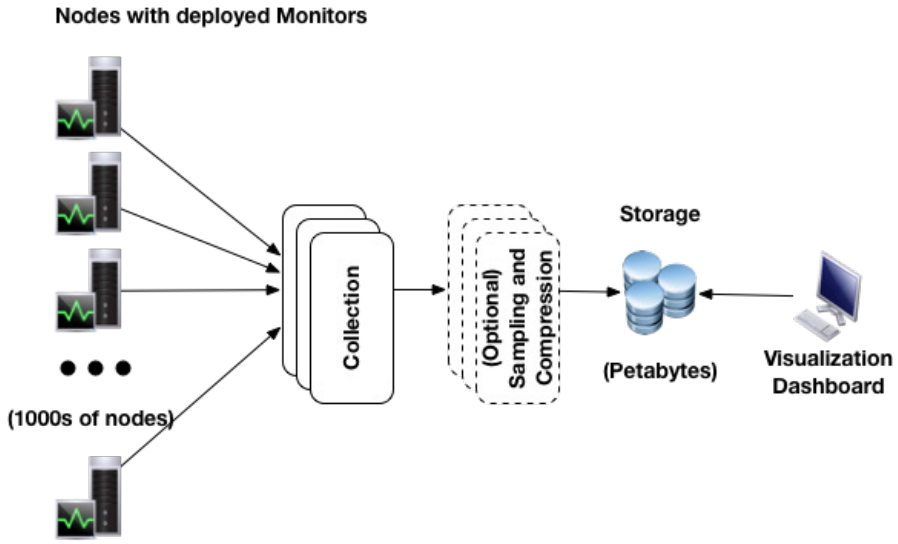


Figure 7.4: Runtime Framework

We use the Ganglia framework [116] for node-level monitor insertion and management, and for measurement collection. The generated monitor code extends the Gmond (Ganglia Monitoring Daemon), which is deployed to every node in the big data system.

The architectural style defined in the metamodel identifies architecture-specific performance metrics, and how these top-level metrics relate to node-level metrics. This knowledge is used by the generator in the MDE Design Time Toolkit to identify opportunities for node-level aggregation of metrics, prior to collection, and to generate extensions for the Ganglia Gmetad daemon, which aggregates data from multiple Gmond instances. These extensions perform architecture-aware measurement aggregation data at the cluster level, with the cluster definition being derived from the architectural styles being used. The number and placement of the generated Gmetad instances is determined by the architectural style and system model properties.

The runtime framework also includes storage for the collected measurements. This storage must scale to support writing millions of records per sampling interval while executing ongoing queries from the visualization dashboard. Here we diverge from the Ganglia framework. While Gan-

glia uses RRDtool (Round-Robin Database) for measurement storage, our initial solution is based on the Cassandra wide column NoSQL database (`http://cassandra.apache.org`), which balances high write performance with query flexibility. Although this NoSQL database does not impose a schema on the data, writers and readers must share a common data model. This data model is architecture-aware, generated from knowledge of the architectural style in the metamodel and from specific property values specified in the system model.

7.6 Future Work

Creation of this model-driven observability solution poses significant research challenges. The first is the identification and characterization of big data architecture styles. A style definition includes the structure of the style (element types and composition rules), and the allowable property values for the elements. Both must be sufficiently general to be reusable for performance monitoring, but sufficiently specific and constrained to enable generation of monitors and visualizations.

Next, the measures collected by the generated monitors will likely require sampling and compression. Resource utilization (CPU and network) for the monitors on each node must be minimized (typically less than 1%), and so time-series compression or other approaches will be needed. For some types of metrics, an approach may include local storage of fine-grained measurements at each node or cluster, along with distributed filtering so that only exceptions or deviations are intelligently reported up to the runtime framework. An essential challenge is that we would like precise and fine-grained measurements when the system is at peak load and resources are already stretched thin. Tradeoffs between push and pull of measurement data, variable sampling frequencies, and aggregation/compression approaches will be explored.

Development of the proposed solution also presents a number of significant engineering challenges. The collection and storage components of the solution are themselves a big data system, which must ingest and store data at terabyte- to petabyte-scale. To address the scale of the data to be monitored, the visualizations will have to use advanced visual metaphors such as cluster-

ing, trending, hotspots, and correlations, performed in real time and at scale. The architectural style definitions must also model the relationships between node-level resource measurements and end-to-end system-level measures, and the visualizations must allow navigation up and down these relationships.

This runtime performance monitoring solution can also be used as the foundation for automation of design time benchmarking and performance characterization tasks. The architecture for the prototype is described using the Design Time Editor, and monitors and visualizations are generated. For benchmarking, an additional component, a test client, is added to the solution, and architecture-aware workloads can be generated for execution by the test client. The test client is based on the Yahoo! Cloud Serving Benchmark (YCSB) framework [38], or the YCSB++ extension [129] that supports coordination of multiple client instances for higher performance.

7.7 Conclusion

At the scale and dynamic runtime environment of big data systems, design time performance predictions must be followed by runtime performance monitoring. We described the architecture of a proposed toolset for runtime performance monitoring. The toolset uses architectural styles to enable generation of architecture-aware monitors and visualizations that execute within a runtime framework for measurement collection, storage, and visualization. Realizing this toolkit presents significant engineering and research challenges, which are the focus of our research.

8

Model-Driven Observability for Big Data Storage

Summary

The scale, heterogeneity, and pace of evolution of the storage components in big data systems makes it impractical to manually insert monitoring code for observability metric collection and aggregation. In this paper we present an architecture that automates these metric collection processes, using a model-driven approach to configure a distributed runtime observability framework. We describe and evaluate an implementation of the architecture that collects and aggregates metrics for a big data system using heterogeneous NoSQL data stores. Our scalability tests demonstrate that the implementation can monitor 20 different metrics from 10,000 database nodes with a sampling interval of 20 seconds. Below this interval, we lose metrics due to the sustained write load required in the metrics database. This indicates that observability at scale must be able to support very high write loads in a metrics collection database.

8.1 Introduction

In the last decade, the world has seen an exponential growth of digital data, Organizations such as Google and Facebook were born on the internet, and are leading this scale-driven revolution [154]. Beyond the Internet companies, big data applications are becoming pervasive across diverse business and scientific

domains. For example, modern commercial airplanes produce approximately 0.5TB of operational data per flight [53], and by 2020, the Internet of Things (IoT) will generate 4 zettabytes of data per year, supporting monitoring and optimization of processes and services globally [149].

At the scale of these systems, meaningful analysis and prediction of end-to-end performance is usually not feasible at design time. Performance models must capture the complex static and dynamic component compositions in both the system and the underlying execution infrastructures. In addition, accurately representing heterogeneous and highly variable workloads challenges the state of the art in performance modeling. Pragmatically, even if it were possible to build such models, rapid post-deployment data growth, shared cloud-based infrastructures, and rapid application evolution would quickly invalidate model results. Assuring runtime performance at big data scale must be based on observing and analyzing in vivo application behavior. This enables observability into system health and status, both at the infrastructure and application level.

This paper builds on our earlier work [92], that presents the challenges of building massively scalable, easily configurable, lightweight observability solutions. In response to these challenges, we describe a model-driven framework for observability that is the focus of our current research. Model-driven approaches facilitate rapid customization of a framework and eliminate custom code for each deployment, hence reducing costs and effort. In our initial experiments, this framework has been able to efficiently collect and aggregate runtime performance metrics in a big data system with 1000s of storage nodes.

The contributions of our research in this area are:

1. A model-driven architecture, toolset, and runtime framework that allows a designer to describe a heterogeneous big data storage system as a model, and deploy the model automatically to configure an observability framework.

2. A reference implementation of the architecture, using the open source Eclipse package to implement the MDE design client, the open source collectd package to implement the metric collection component, and the open source Grafana package to implement the metrics aggregation and visualization component.

3. Performance and availability results from initial experiments, using the reference implementation.

The initial metamodel and implementation focuses on the pervasive big data pattern known as polyglot persistence [139], which uses multiple heterogeneous data stores (often NoSQL/NewSQL) within a single big data system. We note that a model-driven approach would not be strictly necessary (e.g., a discovery-based approach might be a better solution) if the observability scope was limited to just NoSQL/NewSQL technology. However, we intend this architecture to extend to cover complete big data applications, including processing pipelines and analytics. In this broader case, a model-driven approach provides advantages in automating and creating application-aware metric aggregation and visualization.

8.2 Architecture and Implementation

We have developed an architecture and a reference implementation[1] suitable for further research that addresses the challenges of observability in big data systems. The architecture uses model-driven engineering [25] to automate metric collection, aggregation, and visualization.

8.2.1 Overview of the Observability Architecture

The architecture context is shown in Fig. 8.1, depicting three user roles. The first is *modeling*, representing a DevOps engineer who uses a design time client to create a model of the system's data storage topology and specify the configuration for each heterogeneous database. The model identifies the metrics to capture and their collection frequency. At the completion of the modeling phase, the design time client generates the monitoring configuration for a set of metric collection and visualization elements.

The second role is *observing*, representing a system operator who uses a metric visualization client to monitor system performance. The visualization client is configured using the output of the design-time client to reflect the model of system to be observed. It supports real time monitoring of system operations and allows operators to specify and customize views based on their requirements for situational awareness.

[1] Available at `https://github.com/johnrklein/obs-prototype`

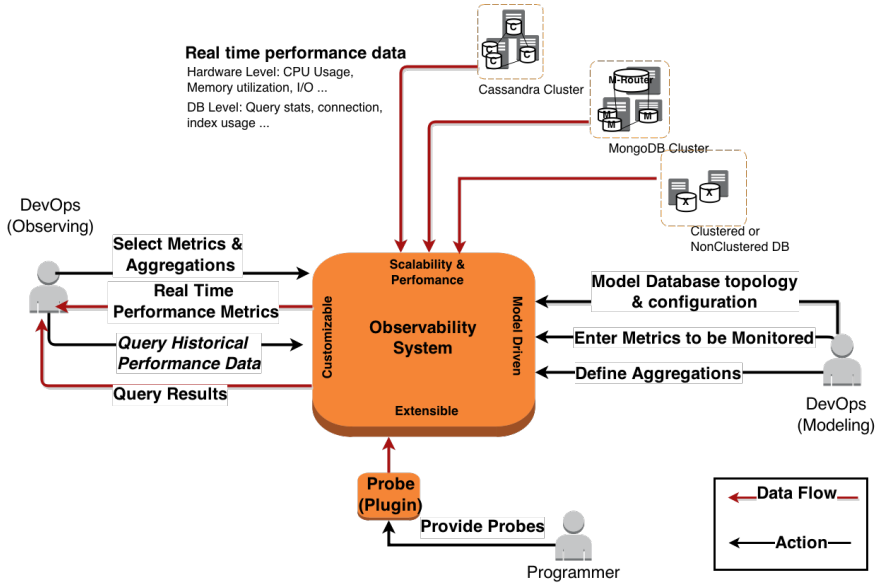


Figure 8.1: Observability Architecture Context Diagram

The third role is *programmers*, who create probes that plug into the architecture to collect metrics. These probes are database-specific adapters that allow any database technology to be incorporated into the architecture. Extensibility is a major feature of our approach, as any solution must be able to efficiently support both existing and future database platforms.

The main run time elements of the observability system architecture are shown in the top-level component and connector diagram in Fig. 8.2. There are two clients, one for each of the main user roles, *modeling* and *observing*, discussed above. The *Server Tier* includes the *Metric Engine*, which implements dynamic metric aggregation and handles concerns related to dependability of connections to *Collection Daemons*. The *Server Tier* also includes the *Grafana Server*, which handles metric visualization. The Model Handler in the Server Tier propagates changes to the design-time model, and the *Notification Server* augments the interactive metric visualization with automated notification of user-defined exception conditions.

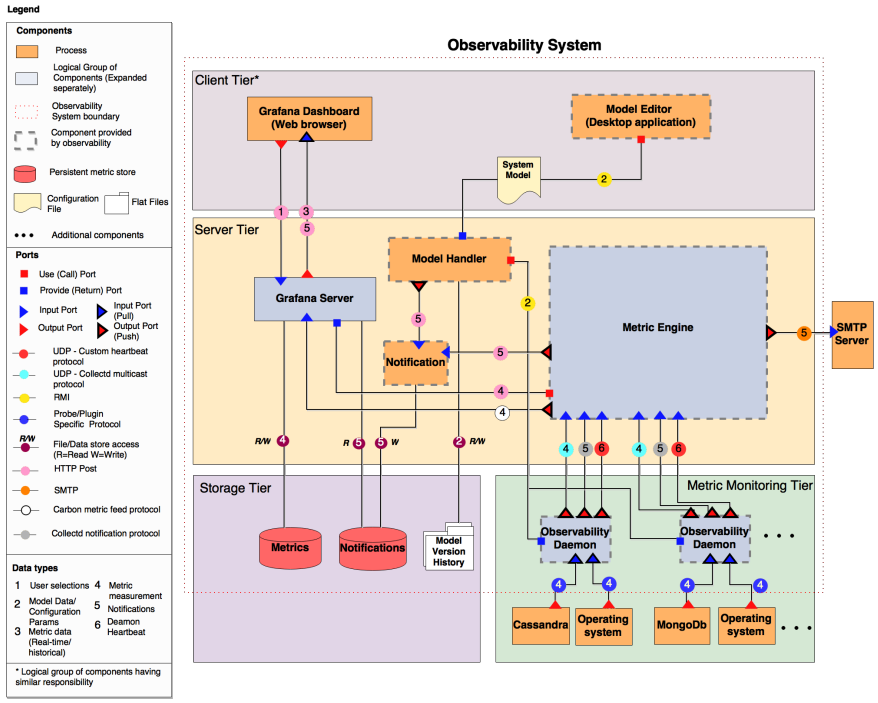


Figure 8.2: Observability System Architecture (Component and Connector View)

The *Storage Tier* provides persistent storage of metric streams and notifications. All metrics for each database are stored as a time series to facilitate visualization and analysis. Metrics are stored with metadata to enable dynamic discovery of the metrics. This is necessary to accommodate changes in monitoring configurations after an existing model has been upgraded and deployed as a new version.

The *Metric Monitoring Tier* uses *Observability Daemons* on each database node to collect metrics from the local database instance and operating system. The daemons exploit database-specific and operating system APIs to periodically sample metrics and forward these to the *Metric Engine*.

8.2.2 Metamodel

Our observability architecture exploits a model-driven engineering approach to address the scale challenge of big data systems. Hence, a model of the system to be observed is created by the *modeling role*. This model is built from elements defined in the metamodel (Fig. 8.3). It specifies and customizes the components in our observability framework. The system model also specifies the metrics to be collected and how the metrics will be aggregated. The metamodel represents the topology as one or more database clusters (*DatabaseCluster* element in the metamodel), with each cluster using a particular technology (*DbType*). A cluster is comprised of a number of nodes (*NodeMachine*).

Metrics (*Metric*) are defined as key-value pairs (*KeyValue*) collected from a database cluster. They may be simple values collected directly from a database's monitoring API (*BaseMetric*), or calculated at run time from one or more simple metrics (*AggregatedMetric*). Each particular database technology (*DbType*) defines a set of metrics that are supported by that technology and are available programmatically. This approach makes it possible to collect both common metrics that are available from all databases (e.g., disk utilization, query processing time) and technology-specific metrics (e.g., automatic rebalancing in MongoDB[2]). These metrics are available for selection when the modeler creates a system model and specifies the databases that will be deployed.

The metamodel also defines the structure of event notifications (*Notification*). These are triggered when simple or aggregated metrics exceed a specified threshold value set by the modeler.

8.2.3 Model Editor Client

In Fig. 8.2, the *Model Editor Client* instantiates the metamodel in a graphical editor, using the Eclipse Modeling Framework (EMF[3]). The graphical model specifying the topology of the observed system and the metrics to be collected and aggregated is transformed using Acceleo[4] into a text representation. This

[2] https://docs.mongodb.org/manual/core/sharding-balancing/
[3] https://eclipse.org/modeling/emf/
[4] http://www.eclipse.org/acceleo/

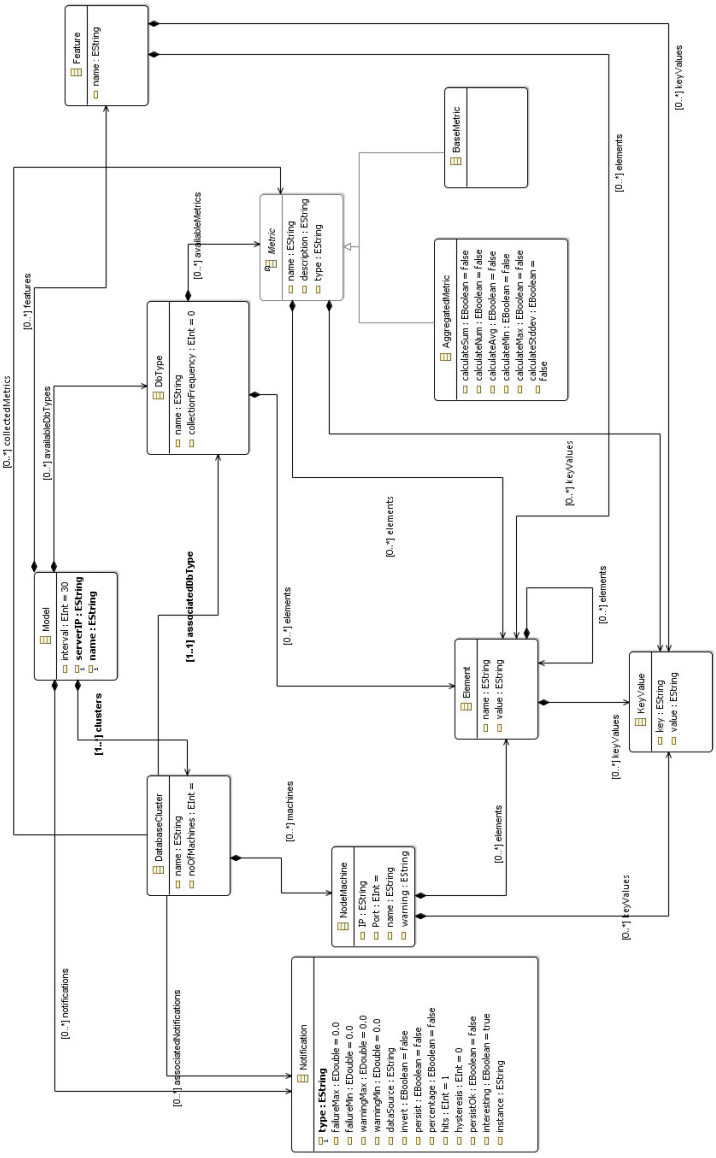


Figure 8.3: Metamodel for Observability of the polyglot persistence pattern

is uploaded to the *Model Handler* server, which propagates model changes to the *Observability Daemons* and *Metrics Engine*. Models are versioned to improve dependability, ensuring all parts of the system are consistent. This also enables rollback to a previous version.

8.2.4 Runtime Metric Collection

An *Observability Daemon* executes on each node in the observed system to collect metrics and forward them to the Metrics Engine. Each *Observability Daemon* is configured by the *Model Handler* based on the system model, so that model changes (e.g., in topology or metrics collected) immediately change the *Observability Daemon* behavior. The *Observability Daemon* is based on the collectd[5] open source package. Our architecture adds a *Daemon Manager* component on top of collectd, so that collectd can be remotely and dynamically configured by the *Model Handler*.

Our architecture uses collectd plug-ins to adapt to each supported database technology, encapsulating the precise mechanism used to obtain database metrics within a database-specific plug-in. The plug-ins exploit the monitoring API provided by the specific database technology (e.g., Cassandra's JMX API[6]), to acquire the metric data from databases. We have developed and tested reference plug-ins for Cassandra, MongoDB, and Riak.

Each *Observability Daemon* sends the collected metrics to the *Metric Engine* server using the collectd notification protocol. Separately, a heartbeat notification is sent by the *Daemon Manager* to the *Missing Daemon* component in the *Metrics Engine*. The *Missing Daemon* component uses the system model to determine which *Observability Daemons* should be executing, compares that to the received heartbeats, and raises an alarm when an *Observability Daemon* appears to have failed. The heartbeat notification was included to improve dependability. Simply monitoring a metric stream to assess the state of an *Observability Daemon* is problematic as individual messages may be delayed due to transient network partitions, or daemon or node failure. Transient partitions can be handled by this protocol, as daemons buffer collected metrics for a configurable time period (e.g., 5 minutes) and can resend missed

[5] `https://collectd.org`

[6] `http://docs.datastax.com/en/cassandra/3.0/cassandra/operations/opsMonitoring.html`

values. Node and daemon failures currently require operator intervention. Automated recovery from such failures simply requires us to incorporate additional monitoring capabilities into our framework, an objective for further work.

8.2.5 Metric Aggregation and Visualization

In our implementation, several metric visualizations are created using the Grafana[7] open source package, which supports time series graphs such as those shown in Fig. 8.4. Grafana comprises a server and a web-based client, as shown in Fig. 8.2. Metrics are stored using self-describing data structures embedded in the metric stream. We utilize key-value pairs, where the metric name comprises the key and the value is the metric reading at a given time.

8.3 Performance Results

To assess the performance and scalability of our observability architecture, we performed a series of tests using Amazon's AWS cloud platform. We created a test daemon that was able to simulate metrics generation from multiple database nodes. We then configured a pool of test daemons to simulate metrics collection from 100 to 10,000 database nodes. We initially set the metrics collection interval to 30 seconds, and configured the daemons to simulate the generation of 20 distinct metrics per node. We also specified the model to aggregate CPU metrics from all nodes to calculate overall system CPU utilization. We deployed the observability architecture on an AWS m3.2xlarge instance type. This comprised an Intel Xeon E5-2670 v2 (Ivy Bridge) server with 8 cores, 8MB RAM, 30GB disk and 160GB SSD. The test daemons were configured to initially simulate metrics generation from 100 database nodes. After 5 minutes, the number of simulated nodes increased to 1000, and then 1000 simulated nodes were added every 5 minutes until the test simulates 10,000 database nodes. We monitored the resource usage on the observability server, and the results are in Fig. 8.5.

Graph 1 shows that the metric transfer time stays constant as the number of metrics per interval increases (the peaks in this graph), and that the metric transfer takes about one-half of the collection interval, leaving margin for growth (the troughs in this graph). The system was able to handle 10,000

[7] `http://grafana.org`

Figure 8.4: Metric visualization user interface

nodes generating 20 metrics during each 30 second interval. The system scaled well to handle network traffic and saved the metrics to disk to be shown at dashboard. The aggregation plugin was able to aggregate metrics from 10,000 nodes successfully.

To summarize the test results:

- With linearly increasing metric collection load, the disk space used also increased linearly. For 10,000 nodes with 20 metrics per node and a 30 second collection interval, disk space required is approximately 50 GB for 7 days monitoring data.
- Server free memory reduced from 25 GB initially to 22 GB with 10000 nodes being monitored. Hence we conclude our solution is not limited in scalability by memory utilization.

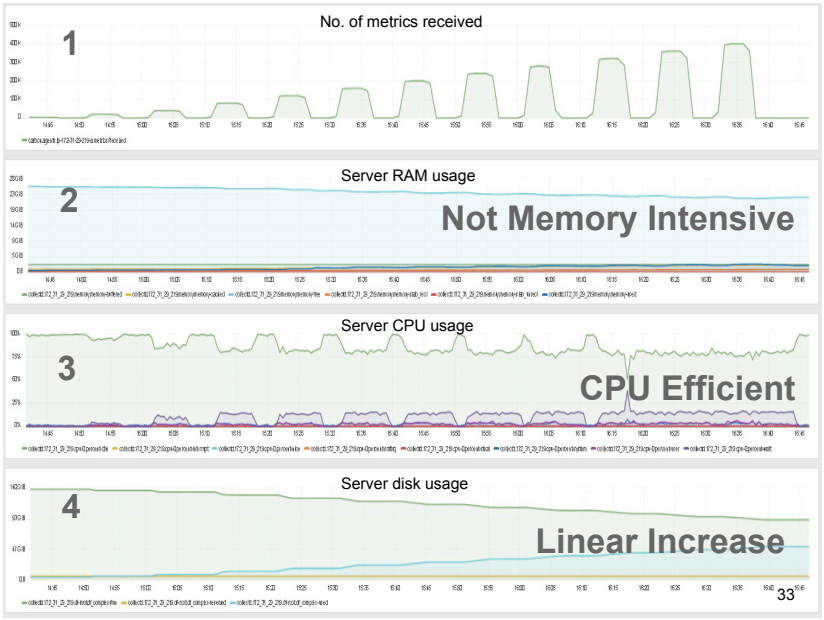


Figure 8.5: Performance and scalability test results

- CPU utilization is low, only showing minor increases in activity as the number of nodes grows.
- With 10000 simulated nodes, the server was processing 293 Kbits/s of network traffic at peak.

To stress test the observability framework, we deployed the test system with 10,000 simulated nodes. The collection interval started at 30 seconds, and every 5 minutes was reduced by 5 seconds. The system operated normally until the sample frequency reached 15 seconds. At this point, some metrics were not written to disk. This situation continued to deteriorate as we reduced the sampling interval to 5 seconds. No components failed, but there was a significant loss of metric data in the database.

Examining execution traces from these failing tests, we saw the CPU, memory, and network utilization levels remained low. Disk writes, however, grew to a peak of 32.7 MB/s. This leads us to believe that the Whisper[8] database in the Grafana server was unable to sustain this relatively heavy write load. This is likely a limitation of this component in our architecture. Replacing this database with a distributed database technology such as Cassandra would consequently make it possible to monitor a significantly larger collection of nodes.

8.4 Prior Work

There has been significant prior work on collecting general measurements of resource utilization at process and node level. This includes Ganglia[9], and Nagios [76]. Ganglia and Splunk[10] support collection of host-level measurements across clusters, and provide basic monitoring and visualization dashboards. Commercial products from HP[11], IBM[12], and others also provide similar collection and visualization capabilities, but incur significant license costs. Tools such as Chukwa[13] and Sawzall[14] focus on general analytics on collected log data, and provide semantics for time series data sets.

In visualizing large-scale system health and performance, Yin and colleagues take an approach inspired by video games to enable navigation through a complex data landscape [160]. In this case, the focus was on infrastructure-level measurement data, however the approach may be extensible for other types of measurements. The Theia system provides architecture-specific visualization for Hadoop-based systems [58].

Architecture-aware modeling based on architecture styles traces back to very early work in software architecture [144]. More recently, Palladio [15] uses architectural styles to generate performance models, and Rainbow [60] uses architectural styles to model and generate a runtime framework focused

[8] http://graphite.wikidot.com/whisper
[9] http://ganglia.sourceforge.net
[10] http://www.splunk.com
[11] https://goo.gl/t3BK3y
[12] http://www.ibm.com/software/tivoli
[13] http://wiki.apache.org/hadoop/Chukwa
[14] http://research.google.com/archive/sawzall.html

on dynamic adaptation. The Rainbow framework uses measurement probes, which may include monitoring performance. However, the probes must be built into the components of the system, and the generation focuses on style-based reaction strategies when a probe's measurement crosses a threshold.

Finally, there has been little work on using model-driven approaches to generate monitors. He and colleagues present a model-driven approach to composing monitors, synthesizing a compatible metamodel and then transforming heterogeneous monitors into that common metamodel. The approach generates only monitors, without aggregations, a measurement persistence schema, or visualizations [73].

8.5 Conclusions and Future Work

In this paper we described the design and prototype implementation of an observability framework for big data systems. We have exploited model-driven techniques to make the core architecture customizable to different system's observability requirements without the need for custom code for each deployment. We have also built the solution by reusing various off-the-shelf components to streamline our development effort and provide advanced capabilities "out of the box". The reference implementation has been publicly released as a research platform. The reference implementation has availability limitations that will be addressed, using standard architecture mechanisms, as we evolve the platform.

Our current implementation only provides observability at the database layer. Extending these model-driven capabilities to other layers in a big data system (e.g., application server and Web servers) and improving the scalability of our framework forms the core of our future work. We also wish to investigate the potential of automated resource discovery approaches to compose an observability system dynamically. Automated approaches have immense potential for dealing with scale and rapid evolution, but face many daunting challenges, for example navigating security perimeters, distributed data centers and logical application partitions.

9 Conclusions

The architecture of a system of systems (SoS) is created by composing systems, with each constituent system retaining independent authority over its operation and evolution. These SoS are increasingly prevalent in practice, for example, as we integrate devices in the Internet of Things (IoT) and as we build data-intensive systems to harness the information produced by disparate sources. In this SoS context, the forces produced by the independent systems cause long-held architecture practices to be ineffective, and different practices are needed. Architecture design practices must enable early technology selection decisions that will shape architecture designs. Architecture documentation practices must work without ready access to stakeholders. Architecture evaluation practices must extend beyond design time to monitor at runtime, at large scale. In this chapter, I revisit the research questions introduced in Chapter 1, summarize the contributions made in answering the questions, and identify further research needed in this area.

9.1 Answer to the Research Questions

In this section, I revisit the research questions, and provide answers. Recall that we identified four main architecture practice areas. I found that the *Identify Architecture Drivers* practice area was relatively mature, overlapping with the well-established requirements engineering practices from systems engineering. However, I found that the forces in the SoS context in the other three practice

areas—*Architecture Design*, *Architecture Documentation*, and *Architecture Evaluation*—caused traditional architecture practices to fail, and new practices were needed. The research questions correspond to each of these three practice areas.

9.1.1 Practice Area—Architecture Design

What decision support is needed to improve the efficiency and quality of technology selection designs of scalable data-intensive systems? (RQ-1)

In a composed architecture such as an SoS, removing abstractions such as the Structured Query Language (SQL) from constituent systems produces a *convergence of concerns*, discussed in our earlier work [65] and summarized above in §1.1.3.

While technology products are described in terms of features, architects reason about concerns in term of quality attributes, patterns, and tactics. Chapter 4 presents a knowledge model that bridges these framings. The knowledge model provides a feature taxonomy for products in a particular technology domain (NoSQL data storage systems). It also identifies the quality attributes that are relevant to an application domain (large-scale data-intensive systems), specifies each quality attribute using a general scenario, and identifies the architecture tactics that promote or inhibit each quality. The bridge from abstract architecture concepts to concrete technology implementations is achieved by relating features in the technology feature taxonomy to specific architecture tactics.

In a technology domain such as NoSQL data storage systems, an architect must choose from among hundreds of alternative products[1]. At such a volume of information, an architect would be helped by a tool that allows storing the knowledge model, and querying and visualization to support architecture reasoning. Chapter 4 also presents a prototype design decision support tool, implemented on the Semantic MediaWiki platform. The tool, called QuABaseBD (QUality At Scale Knowledge Base for Big Data)[2], provides a trusted source of curated architecture knowledge.

[1] See, for example, `https://blogs.the451group.com/information_management/2016/01/26/updated-data-platforms-map-january-2016/`, which lists more than 200 products.

[2] Available at `http://quabase.sei.cmu.edu`

A tool such as QuABaseBD is useful for narrowing the candidate technologies down from hundreds to a handful. Building on earlier research in off-the-shelf technology selection (e.g., [107]), we augmented the QuABaseBD decision support tool with a method, Lightweight Evaluation and Architecture Prototyping for Big Data (LEAP4BD) , presented in Chapter 5. The LEAP4BD method helps architects to efficiently make technology selection decisions in a context where the solution space is evolving rapidly, and so long evaluation cycles will result in choosing technology that is already out-of-date. Chapter 5 also reports on the results and lessons learned from applying the LEAP4BD method in technical action research on a consulting project to select a data store for an electronic health record system.

In summary, architects designing scalable data-intensive systems need decision support to relate product features to architecture concepts such as quality attributes. The scale of the solution space necessitates automated querying and visualization. A trusted knowledge base can help the architect limit the technologies to a few viable candidates, and then a method is needed to make a final decision, using focused prototyping to rapidly collect evidence.

9.1.2 Practice Area—Architecture Documentation

When an existing system will be introduced into an SoS, what additional architecture documentation is needed? (RQ-2)

Architecture documentation is created to address stakeholder concerns. The systematic review of academic peer-reviewed research, presented in Chapter 2, finds that interoperability was a primary concern. However, the state of the practice survey reported in Chapter 3 indicates that practitioners have broader technical and non-technical concerns, which are refined by a survey of practitioner-focused publications, reported in Chapter 6. This survey identifies seven new concerns when a constituent system will be introduced into an SoS. Chapter 6 also presents a documentation viewpoint that specifies and organizes the information needed to address those concerns.

The viewpoint was evaluated in a single case mechanism experiment by a panel of experts, which found a gap in the initial viewpoint specification. The gap was addressed, and the final viewpoint definition comprises six model kinds: Constituent System Stakeholders/Concerns, Constituent System Execution Time Context, Constituent System Code Context, Constituent System Interface Information Model, Shared Resource Model, and Deployment Model.

We found that much of the information needed to assemble this viewpoint would be available in constituent system documentation that uses the View and Beyond documentation approach or the Department of Defense Architecture Framework (DoDAF) documentation approach, and Chapter 6 provides a mapping from these approaches to the new viewpoint.

As discussed in §1.1.3, an SoS architect's request for information about a constituent system can be subject to intense scrutiny, so the viewpoint conforms to the ISO/IEC/IEEE 42010:2011 (E) standard for architecture description to avoid concerns about pedigree.

9.1.3 Practice Area—Architecture Evaluation

What approaches can be used to improve the runtime observability of a large-scale data-intensive system? (RQ-3)

Large-scale data-intensive systems are composed of 100s or 1000s of processes, executing in containers, virtual machines, or (rarely) directly on physical servers. Chapter 7 identifies the key challenges: Economical creation and insertion of monitors into hundreds or thousands of computation and data nodes; efficient, low overhead collection and storage of measurements (which is itself a big data problem); and application-aware aggregation and visualization of the observed data streams.

These challenges are interrelated: Data are observed, transported, stored, and analyzed. While monitor creation and insertion could be addressed simply by a library in each process or local service on each node, application-aware aggregation and visualization requires a more sophisticated approach. Chapter 7 presents a reference architecture that addresses all of these challenges. The architecture style of the system to be observed is used to create monitors for style-specific metrics, and then used to store and analyze the observed streams to assess the system's health based on style-specific qualities. The reference architecture combines model-driven engineering with architecture style modeling. Model-driven engineering provides the automation needed for systems of this scale, and architecture styles provide reuse of architecture knowledge about metrics and qualities.

Chapter 8 presents the results of a single case mechanism experiment in which the reference architecture defined in Chapter 7 was instantiated for the polyglot persistence architecture style. The implementation[3] uses an Eclipse-based custom editor to model the target system using the polyglot persistence style. The model is then used to create technology-specific monitors, a schema to aggregate the observed data streams, and visualizations of the aggregated streams. Model-driven approaches can produce inefficient implementations; however, in this case the performance impact of the metric collection on the target system was acceptable, consuming less than 2% CPU utilization and less than 3GiB memory.

We conclude that the reference architecture presented in Chapter 7, based on model-driven engineering, is a viable approach to achieving runtime observability in a large-scale data-intensive system.

9.2 Answering the Main Research Question

I began this journey by observing clients and colleagues struggle to achieve success when applying traditional architecture practices in complex SoS contexts. This shaped the main research question of this thesis: *How should traditional software architecture practices be changed to address the challenges of large scale, complex SoS contexts?*

The operational and managerial independence of the constituent systems in an SoS stress and break the assumptions that underly traditional architecture practices. The *Identify Architecture Drivers* practice area is more mature than the other practice areas, possibly due to its overlap with the requirements engineering practices of systems engineering. Therefore, I address the main research question by choosing one traditional practice in each of the three remaining architecture practice areas—*Design the Architecture*, *Document the Architecture*, and *Evaluate the Architecture*—and developing a replacement that is sensitive to the forces created by the independence of the constituent systems. For each of these new practices, I created a prototype implementation of a treatment to evaluate the new practice. Note that none of the redefined practices is a radical innovation: Each adapts well-established software archi-

[3] Available at `https://github.com/johnrklein/obs-prototype`.

tecture principles. The *Architecture Design* practices draw from previous work in commercial-off-the-shelf (COTS) evaluation, the *Architecture Documentation* practice follows the metamodel of the ISO/IEC/IEEE 42010:2011 (E) standard, and the *Architecture Evaluation* practice is based on model-driven engineering.

If we interpret the main research question as calling for an exhaustive inventory of all architecture practice changes needed to be successful in the SoS context, then clearly, I have not answered that question. However, if we interpret the question as how to change a traditional practice, then I have answered the question by providing three exemplars. For each, I show how the forces in the SoS context break the traditional process, and how established approaches can be reused to redefine the practice to produce successful results in the SoS context.

No practice is intrinsically good or bad. As a software engineer, our task is to choose or create a practice that is fit for use, given our design context. The contribution of this thesis is guidance for doing this in the SoS context.

9.3 Further Research

This section concludes my thesis, but there is still much work to be done in the area of architecture practices for SoS. Here, I divide this future research into two directions. The first direction continues work on the redefined practices discussed in this thesis, and the second direction considers practices that I did not address in this thesis.

9.3.1 Continuing this work

Chapter 4 concludes by identifying several open research topics related to the QuABaseBD knowledge base. The first is improved visualization of the architecture knowledge to support reasoning and decisions. The LEAP4BD method provides one example of the type of reasoning that architects perform using QuABaseBD , and this also points to the opportunity for a tighter linkage between these artifacts.

The second open research topic related to QuABaseBD is the creation and curation of product feature knowledge. Manually populating the knowledge base by having experts read and extract from product documentation is not scalable. My colleagues started to explore the application of machine learning to automatically populate the knowledge base[4], and found that further work is needed in this area.

The architecture documentation viewpoint defined in Chapter 6 includes mappings from constituent system documentation that is based on either the Views and Beyond approach or the Department of Defense Architecture Framework (DoDAF) approach. This is another area where research could consider using machine learning to automatically extract documentation from one of these frameworks to populate the viewpoint defined here.

Chapter 7 presents a model-driven approach for runtime observability, based on architecture styles. I have done exploratory work on a catalog of architecture styles for big data systems [5], however further work is needed in this area. There is also the possibility that the same model-driven approach used for runtime observability could be applied to creating prototypes in the LEAP4BD method.

9.3.2 Complementing this work

I have framed the *Identify Architecture Drivers* practice area as relatively mature. The qualifier "relatively" is with respect to the three other practice areas, where I found significant gaps. However, this should not imply that no further research is needed in the *Identify Architecture Drivers* practice area. In particular, methods such as the Mission Thread Workshop do not have well-defined stopping criteria, pointing to the need for research to assess what portion of the architecturally-significant requirements space has been covered when using such a method.

[4]This is briefly described in `http://resources.sei.cmu.edu/library/asset-view.cfm?assetid=446274`

[5]This is briefly described in `https://resources.sei.cmu.edu/library/asset-view.cfm?assetID=446337`

The discussion of the *Architecture Design* practice area in §1.1.3 identifies architecture patterns as one element of the architecture body of knowledge used to make architecture design decisions. In the context of SoS, there has been exploratory work on architecture patterns that focused on interoperation [84]; however, as a practitioner, I have found no work on patterns for other architecture qualities and concerns.

My work in the *Architecture Documentation* practice area identifies the information about a constituent system that an SoS architect is concerned with, and the viewpoint specification explains how this information might be extracted from existing system documentation. However, such documentation may not exist, and there are research opportunities to develop approaches to reconstruct the needed information from the system implementation, for example by automatically analyzing the code, or by automatically observing the executing system.

In the *Architecture Evaluation* practice area, my work focused on runtime observability. Design-time evaluation is still needed to assess risk early in the lifecycle. §1.1.3 discusses how traditional methods have scaled up; however, further research is needed to assess the quality of an evaluation (e.g., completeness), and to perform an effective evaluation when information about a constituent system may be limited (e.g., due to the managerial independence forces limiting access to system data).

Finally, SoS are a point on the spectrum of ultra-large-scale (ULS) systems [122], and taking a ULS system perspective opens a wide door to future research in this area.

10 Samenvatting

10.1 Nederlandse samenvatting

De architectuur van een systeem van systemen (system of systems, SoS) komt tot stand door systemen samen te stellen, waarbij ieder deelsysteem onafhankelijke zeggenschap behoudt over zijn werking en evolutie. In de praktijk komen deze SoS systemen steeds meer voor, bijvoorbeeld doordat we apparaten integreren in het Internet of Things (IoT) en doordat we gegevens-intensieve systemen bouwen om informatie uit uiteenlopende bronnen bruikbaar te maken. In deze SoS context zorgen de krachten die deze onafhankelijke systemen produceren ervoor dat gevestigde architectuurpraktijken niet meer werken, en dat er nieuwe praktijken nodig zijn. In plaats van de traditionele praktijk van het kiezen van technologieën om onze architectuurvereisten in te vullen, maken SoS architecten bijvoorbeeld vroeg in de ontwerpcyclus selecties, en bouwen ze vervolgens hun architectuur op rondom die selecties. Samenwerking tussen belanghebbenden wordt ingeperkt door de onafhankelijkheid van de aansturing, waardoor de effectiviteit van traditionele architectuur-documentatiepraktijken wordt beperkt. Tenslotte evolueert de runtime omgeving van SoS systemen continu, waardoor architectuur-evaluatiepraktijken moeten worden uitgebreid, en naast de ontwerpfase bijvoorbeeld runtime-monitoring moet worden ingevoerd.

In dit proefschrift worden drie vervangende parktijken gepresenteerd voor traditionele architectuurpraktijken, te weten *Architectuurontwerp*, *Architectuurdocumentatie* en *Architectuurevaluatie*. Ieder van deze vervangende praktijken is gevoelig voor de krachten die veroorzaakt worden door de onafhankelijkheid van de samenstellende systemen in een SoS.

Het gebied van de *Architectuurontwerp*-praktijk wordt behandeld met een kennismodel waarin een brug wordt geslagen tussen abstracte architectuuranalyse op basis van kwaliteitsattributen, patronen en taktieken, en concrete kenmerken van specifieke technologieën in een specifiek domein (in dit geval NoSQL data-opslag). Gebaseerd op dit kennismodel werd een prototype voor een beslis-ondersteunend systeem ontwikkeld en geëvalueerd.

Op het gebied van de *Architectuurdocumentatie*-praktijk wordt een documentatie-gezichtspunt gedefinieerd. Het gezichtspunt specificeert en organiseert de informatie die een SoS-architect nodig heeft over een deelsysteem dat onderdeel wordt van een SoS. Het gezichtspunt wordt op compleetheid getoetst.

Op het gebied van de *Architectuurevaluatie*-praktijk tenslotte wordt in dit proefschrift een prototype geïntroduceerd van een referentie-architectuur die een modelgedreven constructie-aanpak gebruikt voor de runtime observeerbaarheid van een grootschalig SoS systeem met honderden of duizenden nodes. In de referentiearchitectuur wordt modelgedreven constructie (*model-driven engineering*) gecombineerd met modelleren op basis van architectuurstijlen. Modelgedreven constructie zorgt voor de automatisering die nodig is voor systemen van deze schaal, en door het toepassen van architectuurstijlen wordt kennis over metrieken en kwaliteit hergebruikt.

Geen enkele praktijk is intrinsiek goed of slecht. Als software-engineers is het onze taak om een praktijk te kiezen of creëren die past bij de ontwerpcontext. De bijdrage van dit proefschrift is een leidraad hiervoor in de SoS-context.

10.2 English Summary

The architecture of a system of systems (SoS) is created by composing systems, with each constituent system retaining independent authority over its operation and evolution. These SoS are increasingly prevalent in practice, for example, as we integrate devices in the Internet of Things (IoT) and as we build data-intensive systems to harness the information produced by disparate sources. In this SoS context, the forces produced by the independent systems cause long-held architecture practices to be ineffective, and different practices are needed. For example, rather than the traditional practice of choosing technologies to satisfy our architecture drivers, SoS architects make selections early in the design cycle and then build their architecture around

those selections. Managerial independence restricts stakeholder collaboration, limiting the effectiveness of traditional architecture documentation practices. Finally, the runtime environment of the SoS is ever-evolving, and so architecture evaluation practices must extend from design time to include approaches like runtime monitoring.

This dissertation presents three replacements for traditional architecture practices, addressing the practice areas of *Architecture Design*, *Architecture Documentation*, and *Architecture Evaluation*. Each of these replacement practices is sensitive to the forces created by the independence of the constituent systems in an SoS.

The *Architecture Design* practice area was addressed with a knowledge model that bridges between abstract architecture reasoning based on quality attributes, patterns, and tactics, and concrete features of specific technologies in a particular domain (in this case, NoSQL data storage). A prototype decision support tool based on this knowledge model was developed and evaluated.

In the *Architecture Documentation* practice area, an architecture documentation viewpoint was defined. The viewpoint specifies and organizes the information that an SoS architect needs about a constituent system that will be introduced into an SoS. The viewpoint was evaluated for completeness.

Finally, in the *Architecture Evaluation* practice area, this dissertation introduces and evaluates a prototype of a reference architecture that uses a model-driven engineering approach for runtime observability of a large scale SoS with 100s or 1000s of nodes. The reference architecture combines model-driven engineering with architecture style modeling. Model-driven engineering provides the automation needed for systems of this scale, and architecture styles provide reuse of architecture knowledge about metrics and qualities.

No practice is intrinsically good or bad. As a software engineer, our task is to choose or create a practice that is fit for use, given our design context. The contribution of this dissertation is the guidance for doing this in the SoS context.

References

[1] Daniel J. Abadi. Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. *Computer*, 45(2):37–42, 2012. doi:10.1109/MC.2012.33.

[2] Divyakant Agrawal, Sudipto Das, and Amr El Abbadi. Big data and cloud computing: current state and future opportunities. In *Proc. 14th Int. Conf. on Extending Database Technology*, EDBT/ICDT'11, pages 530–533, March 2011. doi:10.1145/1951365.1951432.

[3] Art Akerman and Jeff Tyree. Using ontology to support development of software architectures. *IBM Systems Journal*, 45(4):813–825, 2006.

[4] M. A Babar, T. Dingsøyr, P. Lago, and H. van Vliet. *Software Architecture Knowledge Management: Theory and Practice*. Springer-Verlag, 2009.

[5] L. T. Babu, M. Seetha Ramaiah, T. V. Prabhakar, and D. Rambabu. ArchVoc-towards an ontology for software architecture. In *Proc. 2nd Workshop on SHAring and Reusing architectural Knowledge-Architecture, Rationale, and Design Intent*, SHARK-ADI '07, 2007.

[6] Felix Bachmann, Len Bass, and Mark Klein. Preliminary design of ArchE: A software architecture design assistant. Technical Report CMU/SEI-2003-TR-021, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA, September 2003. URL: http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=6751 [cited 1 March 2017].

[7] Felix Bachmann, Robert L. Nord, and Ipek Ozkaya. Architectural tactics to support rapid and agile stability. *CrossTalk*, pages 20–25, May/June 2012.

[8] Kristen Baldwin and Judith Dahmann. Sos considerations in the engineering of systems. In *Proc. 17th Ann. NDIA Systems Eng. Conf.* National Defense Industry Association, 2014. URL: http://www.dtic.mil/ndia/2014/system/16865ThursTrack2Baldwin.pdf [cited 1 June 2017].

[9] Mario R. Barbacci, Robert J. Ellison, Anthony J. Lattanze, Judith A. Stafford, Charles B. Weinstock, and William G. Wood. Quality attribute workshops (QAWs), 3rd edn. Technical Report CMU/SEI-2003-TR-016, Software Engineering Institute, Pittsburgh, PA, USA, October 2003. URL: http://resources.sei.cmu.edu/library/asset-view.cfm?assetID=6687 [cited 24 September 2014].

[10] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley, 3rd edition, 2013.

[11] Len Bass, Paul Clements, Rick Kazman, John Klein, Mark Klein, and Jeanine Siviy. A workshop on architecture competence. Technical Note CMU/SEI-2009-TN-005, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2009. URL: http://www.sei.cmu.edu/library/abstracts/reports/09tn005.cfm [cited 15 July 2013].

[12] Thais Batista. Challenges for SoS architecture description. In *Proc. 1st Int. Workshop on Software Engineering for Systems-of-Systems*, SESoS '13, pages 35–37, 2013. doi:10.1145/2489850.2489857.

[13] BBC News. Instagram, Vine and Netflix hit by Amazon glitch [online]. 2013. URL: http://www.bbc.com/news/technology-23839901 [cited 1 March 2017].

[14] C. Becker, M. Kraxner, and M. Plangg. Improving decision support for software component selection through systematic cross-referencing and analysis of multiple decision criteria. In *Proc. 46th Hawaii Int. Conf. on System Sciences*, HICSS, pages 1193–1202, 2013.

[15] Steffen Becker, Heiko Koziolek, and Ralf Reussner. The Palladio component model for model-driven performance prediction. *J. of Systems and Software*, 82(1):3–22, Jan 2009. doi:10.1016/j.jss.2008.03.066.

[16] Sarah Beecham, Tracy Hall, Carol Britton, Michaela Cottee, and Austen Rainer. Using an expert panel to validate a requirements process improvement model. *Journal of Systems and Software*, 76(3):251–275, 2005. doi:10.1016/j.jss.2004.06.004.

[17] S. Bellomo, R. L. Nord, and I. Ozkaya. A study of enabling factors for rapid fielding combined practices to balance speed and stability. In *Proc. 35th Int. Conf. on Software Eng.*, ICSE '13, pages 982–991, May 2013. doi:10.1109/ICSE.2013.6606648.

[18] Caesar Benipayo. Understanding system interdependence to improve resilience of shipboard cyber physical system. In *Proc. 19th Ann. NDIA Systems Eng. Conf.* National Defense Industry Association, 2016. URL: http://www.dtic.mil/ndia/2016/systems/18881_CaesarBenipayo.pdf [cited 1 June 2017].

[19] John Bergey, Jr. Stephen Blanchette, Paul Clements, Michael Gagliardi, Rob Wojcik, William Wood, and John Klein. U.S. Army workshop on exploring enterprise, system of systems, system, and software architectures. Technical Report CMU/SEI-2009-TR-008, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2009. URL: http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=9099 [cited 1 Jan 2017].

[20] Thiago Bianchi, Daniel Soares Santos, and Katia Romero Felizardo. Quality attributes of systems-of-systems: A systematic literature review. In *Proc. 3rd Int. Workshop on Software Engineering for Systems-of-Systems*, SESoS '15, pages 23–30, 2015. doi:DOI10.1109/SESoS.2015.12.

[21] Randy Bias. Architectures for open and scalable clouds. In *Proc. CloudConnect 2012*, February 2012. URL: http://www.slideshare.net/randybias/architectures-for-open-and-scalable-clouds [cited 31 Oct 2014].

[22] J. Boardman and B. Sauser. System of systems - the meaning of of. In *Proc. IEEE/SMC Int. Conf. on System of Systems Eng.*, SoSE, pages 118–123, April 2006. doi:10.1109/SYSOSE.2006.1652284.

[23] Jan Bosch. *Design & Use of Software Architectures*. Addison-Wesley, Harlow, UK, 2000.

[24] Jan Bosch. From software product lines to software ecosystems. In *Proc. 13th Int. Software Product Line Conf.*, SPLC'09, 2009.

[25] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice*. Morgan & Claypool, 2012.

[26] Eric Brewer. CAP twelve years later: How the "rules" have changed. *Computer*, 45(2):23–29, February 2012. doi:10.1109/MC.2012.37.

[27] John Brøndum and Liming Zhu. Towards an architectural viewpoint for systems of software intensive systems. In *Proc. 2010 ICSE Workshop on Sharing and Reusing Arch. Knowledge*, SHARK '10, pages 60–63, 2010. doi:10.1145/1833335.1833344.

[28] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture Volume 1: A System of Patterns*. Wiley, 1996.

[29] P.G. Carlock, S.C. Decker, and R.E. Fenton. Agency-level systems engineering for 'systems of systems'. *Systems and Information Technology Review Journal*, 16:99–110, 1999.

[30] Ronald Carson. Differentiating system architectures. In *Proc. 17th Ann. NDIA Systems Eng. Conf.* National Defense Industry Association, 2014. URL: http://www.dtic.mil/ndia/2014/system/16801ThursTrack6Carson.pdf [cited 1 June 2017].

[31] Humberto Cervantes and Rick Kazman. *Designing Software Architectures: A Practical Approach*. Addison-Wesley, Boston, MA, USA, 2016.

[32] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2), 2008. doi:10.1145/1365815.1365816.

[33] L. Chen, M. Ali Babar, and N. Ali. Variability management in software product lines: a systematic review. In *Proc. 13th Int. Software Product Line Conf.*, SPLC, pages 81–90, 2009.

[34] Paul Clements, Felix Bachman, Len Bass, David Garlan, James Ivers, Reed Little, Paulo Merson, Robert Nord, and Judith Stafford. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley, 2nd edition, 2011.

[35] Paul Clements, Rick Kazman, and Mark Klein. *Evaluating Software Architectures: Methods and Case Studies*. Addison-Wesley, 2002.

[36] Santiago Comella-Dorda, John Dean, Grace Lewis, Edwin Morris, Patricia Oberndorf, and Erin Harper. A process for COTS software product evaluation. Technical Report CMU/SEI-2003-TR-017, Software Engineering Institute, Pittsburgh, PA, USA, July 2004. URL: http://resources.sei.cmu.edu/library/asset-view.cfm?assetid=6701 [cited 14 Mar 2017].

[37] COMPASS. Comprehensive modelling for advanced systems of systems [online]. 2014. URL: http://www.compass-research.eu/index.html [cited 1 June 2017].

[38] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proc. 1st ACM Symp. on Cloud Computing*, SoCC '10, pages 143–154, 2010. doi:10.1145/1807128.1807152.

[39] Michael Cusumano. The evolution of platform thinking. *Comm. ACM*, 53(1):32–34, January 2010. doi:10.1145/1629175.1629189.

[40] J. Dahmann and K. Baldwin. Implications of systems of systems on system design and engineering. In *Proc. 6th Int. Conf. on System of Systems Engineering*, SoSE, pages 131–136, June 2011. doi:10.1109/SYSOSE.2011.5966586.

[41] Judith Dahmann. SoS pain points - INCOSE SoS working group survey. In *Proc. 15th Ann. NDIA Systems Eng. Conf.*, San Diego, CA, 2012. National Defense Industry Association. URL: http://www.dtic.mil/ndia/2012system/ttrack214770.pdf [cited 27 Sep 2013].

[42] Judith Dahmann and Rob Heilmann. SoS Systems Engineering (SE) and Test & Evaluation (T&E): Final Report of the NDIA SE Division SoS SE and T&E Committees. In *15th Annual NDIA Systems Engineering Conference*, San Diego, CA, March 2012. NDIA. URL: http://www.dtic.mil/ndia/2012system/ttrack214771.pdf [cited 27 Sep 2013].

[43] Remco C. de Boer, R. Farenhorst, P. Lago, Hans van Vliet, Viktor Clerc, and Anton Jansen. Architectural knowledge: Getting to the core. In *Proc. 3rd Int. Conf. on the Quality of Software Architectures*, pages 197–214, 2007. doi:10.1007/978-3-540-77619-2_12.

[44] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Comm. ACM*, 56(2):74–80, February 2013. doi:10.1145/2408776.2408794.

[45] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proc. 21st ACM SIGOPS Symp. on Operating Systems Principles*, SOSP '07, pages 205–220. ACM, 2007. doi:10.1145/1294261.1294281.

[46] J. Dimarogonas. A theoretical approach to C4ISR architectures. In *Proc. IEEE Military Comm. Conf.*, MILCOM, pages 28–33, 2004. doi:10.1109/MILCOM.2004.1493242.

[47] Torgeir Dingsøyr and Hans van Vliet. *Software Architecture Knowledge Management: Theory and Practice*, chapter Introduction to Software Architecture and Knowledge Management, pages 1–17. Springer, Heidelberg, 2009.

[48] DOD Deputy Chief Information Officer. The DOD architecture framework version 2.02. Standard, United States Department of Defense, 2010. URL: http://dodcio.defense.gov/Library/DoD-Architecture-Framework/ [cited 18 Oct 2016].

[49] T. Dyba, T. Dingsoyr, and G. K. Hanssen. Applying systematic reviews to diverse study types: An experience report. In *Proc. 1st Int. Symp. on Empirical Software Eng. and Measurement*, ESEM 2007, pages 225–234, Sept 2007. doi:10.1109/ESEM.2007.59.

[50] Thomas Erl. *SOA: Principles of Service Design*. Prentice Hall, 1st edition, July 2007.

[51] European Commission. System-of-systems [online]. 2015. URL: https://ec.europa.eu/digital-single-market/en/system-systems [cited 1 March 2017].

[52] Ricardo De Almeida Falbo, Fabiano Borges Ruy, and Rodrigo Dal Moro. Using ontologies to add semantics to a software engineering environment. In *Proc. SEKE 2005*, SEKE, 2005.

[53] Matthew Finnegan. Boeing 787s to create half a terabyte of data per flight, says Virgin Atlantic. *Computerworld UK*, March 2013. URL: `https://goo.gl/T3zhqG` [cited 20 Feb 2014].

[54] Kevin Forsberg and Harold Mooz. The relationship of system engineering to the project cycle. In *Proc. 1st Ann. Symp. of National Council on System Eng.*, pages 57–65, October 1991.

[55] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, 2002.

[56] Michael Gagliardi, William Wood, and Timothy Morrow. Introduction to the mission thread workshop. Technical Report CMU/SEI-2013-TR-003, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA, October 2013. URL: `http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=63148` [cited 14 March 2017].

[57] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[58] Elmer Garduno, Soila P. Kavulya, Jiaqi Tan, Rajeev Gandhi, and Priya Narasimhan. Theia: Visual signatures for problem diagnosis in large Hadoop clusters. In *Proc. 26th Int. Conf. on Large Installation System Administration: Strategies, Tools, and Techniques*, lisa'12, pages 33–42, 2012.

[59] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch: why reuse is so hard. *IEEE Software*, 12(6):17–26, Nov 1995. `doi:10.1109/52.469757`.

[60] David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley Schmerl, and Peter Steenkiste. Rainbow: Architecture-based self adaptation with reusable infrastructure. *IEEE Computer*, 37(10), October 2004. `doi:10.1109/MC.2004.175`.

[61] David Garlan, Robert T. Monroe, and David Wile. Acme: An architecture description interchange language. In *Proc. 1997 Conf. of Centre for Advanced Studies on Collaborative Research*, CASCON'97, pages 169–183, November 1997.

[62] David A. Garlan. Software engineering in an uncertain world. In *Proc. FSE/SDP Workshop on the Future of Software Engineering Research*, FoSER, pages 125–128, November 2010. `doi:10.1145/1882362.1882389`.

[63] A. Gorod, B. Sauser, and J. Boardman. System-of-systems engineering management: A review of modern history and a path forward. *IEEE Systems Journal*, 2(4):484 –499, December 2008. doi:10.1109/JSYST.2008.2007163.

[64] Ian Gorton. *Essential Software Architecture*. Springer-Verlag, 2nd edition, 2011. doi:10.1007/978-3-642-19176-3.

[65] Ian Gorton and John Klein. Distribution, data, deployment: Software architecture convergence in big data systems. *Software*, 32(3):78–85, May-June 2015. doi:10.1109/MS.2014.51.

[66] Jonathan W. Greenert. Payloads over platforms: Charting a new course. *U.S. Naval Institute Proceedings Magazine*, 138(7), July 2012.

[67] Peter Groves, Basel Kayyali, David Knott, and Steve Van Kuiken. The 'big data' revolution in healthcare. Report, McKinsey & Company, January 2013. URL: http://www.mckinsey.com/insights/health_systems_and_services/the_big-data_revolution_in_us_health_care [cited 20 Feb 2014].

[68] Nickolas Guertin. Capability based technical reference frameworks for open system architecture implementations. In *Proc. 17th Ann. NDIA Systems Eng. Conf.* National Defense Industry Association, 2014. URL: http://www.dtic.mil/ndia/2014/system/16909ThursTrack6Guertin.pdf [cited 1 June 2017].

[69] Milena Guessi, Valdemar V. G. Neto, Thiago Bianchi, Katia R. Felizardo, Flavio Oquendo, and Elisa Y. Nakagawa. A systematic literature review on the description of software architectures for systems of systems. In *Proc. 30th Ann. ACM Symp. on Applied Computing*, SAC '15, pages 1433–1440, 2015. doi:10.1145/2695664.2695795.

[70] Jamieson Gump. An architecture for agile systems engineering of secure commercial-off-the-shelf (COTS) mobile communications. In *Proc. 17th Ann. NDIA Systems Eng. Conf.* National Defense Industry Association, 2014. URL: http://www.dtic.mil/ndia/2014/system/16845ThursTrack6Gump.pdf [cited 1 June 2017].

[71] C. Hakim. *Contemporary Social Research: 13*, chapter Research Design: Strategies and Choices in the Design of Social Research. Routledge, London, UK, 1987.

[72] H-J. Happel and S. Seedorf. Applications of ontologies in software engineering. In *Proc. Workshop on Sematic Web Enabled Software Eng.*, SWESE, 2006.

[73] Yuqin He, Xiangping Chen, and Ge Lin. Composition of monitoring components for on-demand construction of runtime model based on model synthesis. In *Proc. 5th Asia-Pacific Symp. on Internetware*, Internetware '13, pages 20:1–20:4, 2013. doi:10.1145/2532443.2532472.

[74] Gregor Hohpe and Bobby Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Professional, 2003.

[75] Eric Honour. DANSE–an effective, tool-supported methodology for systems of systems engineering in Europe. In *Proc. 16th Ann. NDIA Systems Eng. Conf.* National Defense Industry Association, 2013. URL: http://www.dtic.mil/ndia/2013/system/TH16282_Honour.pdf [cited 1 June 2017].

[76] Emir Imamagic and Dobrisa Dobrenic. Grid infrastructure monitoring system based on nagios. In *Proc. 2007 Workshop on Grid Monitoring*, GMW '07, pages 23–28, 2007. doi:10.1145/1272680.1272685.

[77] INCOSE. A consensus of the INCOSE fellows [online]. October 2006. URL: http://www.incose.org/practice/fellowsconsensus.aspx [cited 18 Aug 2012].

[78] ISO/IEC. ISO/IEC 9075:2003 database language SQL, parts 1-4, 9-11, 13. Standard, ISO/IEC, 2003.

[79] ISO/IEC/IEEE. ISO/IEC/IEEE 42010: Systems and software engineering - architecture description. Standard, ISO/IEC/IEEE, 2011.

[80] A. S. Jadhav and R. M. Sonar. Framework for evaluation and selection of the software packages: A hybrid knowledge based system approach. *J. of Systems and Software*, 84(8):1394–1407, August 2011. doi:10.1016/j.jss.2011.03.034.

[81] Mohommad Jamshidi. *System of Systems Engineering: Innovations for the 21st Century*. Wiley Series in Systems Engineering and Management. John Wiley & Sons, Hoboken, New Jersey, 2009.

[82] A. Jansen and J. Bosch. Software architecture as a set of architectural design decisions. In *Proc. 5th Working IEEE/IFIP Conf. on Software Architecture*, WICSA, pages 109–120, 2005. doi:10.1109/WICSA.2005.61.

[83] A. Jansen, J.S. van der Ven, P. Avgeriou, and Dieter K. Hammer. Tool support for architectural decisions. In *Proc. 6th Working IEEE/IFIP Conf. on Software Architecture*, WICSA, pages 44–53, 2007. doi:10.1109/WICSA.2007.47.

[84] R. Kazman, K. Schmid, C.B. Nielsen, and J. Klein. Understanding patterns for system of systems integration. In *Proc. 8th Int. Conf. on System of Systems Engineering*, SoSE 2013, pages 141–146, June 2013. doi:10.1109/SYSoSE.2013.6575257.

[85] Rick Kazman, Michael Gagliardi, and William Wood. Scaling up software architecture analysis. *J. of Systems and Software*, 85(7):1511–1519, July 2012. doi:10.1016/j.jss.2011.03.050.

[86] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1):41–50, January 2003. doi:10.1109/MC.2003.1160055.

[87] Jung Soo Kim and David Garlan. Analyzing architectural styles. *J. of Systems and Software*, 83:1216–1235, 2010. doi:10.1016/j.jss.2010.01.049.

[88] Barbara Kitchenham and Stuart Charters. Guidelines for performing systematic literature reviews in software engineering, version 2.3. Technical report, Software Engineering Group, School of Computer Science and Mathematics, Keele University, Keele, Staffs ST5 5BG, UK, July 2007.

[89] Barbara A. Kitchenham, Lawrence Pfleeger, Lesley M. Pickard, Peter W. Jones, David C. Hoaglin, Khaled El Emam, and Jarrett Rosenberg. Preliminary guidelines for empirical research in software engineering. *IEEE Trans. Softw. Eng.*, 28(8):721–734, August 2002. doi:10.1109/TSE.2002.1027796.

[90] John Klein, Gary J. Chastek, Sholom Cohen, Rick Kazman, and John D. McGregor. An early look at defining variability requirements for system of systems platforms. In *Proc. 2nd Int. Workshop on Reqts Eng. for Systems of Systems*, RESS'12, pages 30–33, 2012. doi:10.1109/RES4.2012.6347693.

[91] John Klein, Sholom Cohen, and Rick Kazman. Common platforms in system-of-systems architectures: The state of the practice. In *Proc. IEEE/SMC Int. Conf. on System of Systems Eng.*, SoSE, 2013. URL: http://resources.sei.cmu.edu/library/asset-view.cfm?assetID=68315 [cited 1 Jan 2017].

[92] John Klein and Ian Gorton. Runtime performance challenges in big data systems. In *Proc. Workshop on Challenges in Performance Methods for Software Development*, WOSP-C'15, 2015. doi:10.1145/2693561.2693563.

[93] John Klein, Ian Gorton, Neil Ernst, Patrick Donohoe, Kim Pham, and Chrisjan Matser. Performance evaluation of NoSQL databases: A case study. In *Proc. of 1st Workshop on Performance Analysis of Big Data Systems*, PABS 2015, Feb 2015. doi:10.1145/2694730.2694731.

[94] John Klein and Hans van Vliet. A systematic review of system-of-systems architecture research. In *Proc. 9th Int. ACM SIGSOFT Conf. on the Quality of Software Architectures*, QoSA'13, pages 13–22, Vancouver, BC, Canada, June 2013. ACM. doi:10.1145/2465478.2465490.

[95] H. Knublauch. Ontology-driven software development in the context of the semantic web: An example scenario with Protege/OWL. In *Proc. 1st Int. Workshop on the Model-driven Semantic Web*, MDSW2004, 2004.

[96] Kristof Kovacs. Cassandra vs mongodb vs couchdb vs redis vs riak vs hbase vs couchbase vs orientdb vs aerospike vs neo4j vs hypertable vs elasticsearch vs accumulo vs voltdb vs scalaris comparison [online]. 2014. URL: http://kkovacs.eu/cassandra-vs-mongodb-vs-couchdb-vs-redis [cited 7 Oct 2014].

[97] Jonah Kowall and Will Cappelli. Magic quadrant for application performance monitoring. Research Report G00262851, Gartner, Inc., Stamford, CT, USA, October 2014.

[98] P. Kruchten, R. Capilla, and J.C. Dueas. The decision view's role in software architecture practice. *IEEE Software*, 26(2):36–42, 2009. doi:10.1109/MS.2009.52.

[99] P. Kruchten, P. Lago, and H. van Vliet. Building up and reasoning about architectural knowledge. In *Proc. 2nd Int. Conf. on the Quality of Software Architectures*, QoSA '06, pages 39–47, 2006. doi:10.1007/11921998_8.

[100] Philippe Kruchten. An ontology of architectural design decisions in software intensive systems. In *Proc. 2nd Groningen Workshop on Software Variability.*, 2004.

[101] Phillipe Kruchten. *The Rational Unified Process: An Introduction*. Addison-Wesley Professional, Boston, MA, USA, 3rd edition, 2003.

[102] Jo Ann Lane, Judith Dahmann, George Rebovich, and Ralph Lowry. Key system of systems engineering artifacts to guide engineering activities. In *Proc. 13th Ann. NDIA Systems Eng. Conf.*, San Diego, CA, October 2010. National Defense Industry Association. URL: http://www.dtic.mil/ndia/2010systemengr/WednesdayTrack3_10806Lane.pdf [cited 27 Sep 2013].

[103] A. Lapkin. Gartner defines the term 'enterprise architecture'. Research Report G00141795, Gartner Group, Stamford, CT, USA, July 2006.

[104] M. Lehman. Programs, life cycles, and laws of software evolution. *Proc. IEEE*, 68(9):1060–1076, 1980. doi:10.1109/PROC.1980.11805.

[105] Timothy C. Lethbridge, Susan E. Sim, and Janice Singer. Studying software engineers: data collection techniques for software field studies. *Empirical Software Engineering*, 10(3):311–341, 2005. doi:10.1007/s10664-005-1290-x.

[106] Dong Li and Ye Yang. Enhance value by building trustworthy software-reliant system of systems from software product lines. In *Proc. 3rd Int. Workshop on Product Line Approaches in Software Eng.*, PLEASE, pages 13–16, June 2012. doi:10.1109/PLEASE.2012.6229761.

[107] A. Liu and I. Gorton. Accelerating COTS middleware acquisition: the i-Mate process. *IEEE Software*, 20(2):72–79, Mar-Apr 2003. doi:10.1109/MS.2003.1184171.

[108] Yan Liu, Ian Gorton, Len Bass, Cuong Hoang, and Suhail Abanmi. MEMS: A method for evaluating middleware architectures. In *Proc. 2nd Int. Conf. on Quality of Software Architectures*, QoSA'06, pages 9–26, 2006. doi:10.1007/11921998_6.

[109] Azad M. Madni. Adaptable platform-based engineering: Key enablers and outlook for the future. *Systems Engineering*, 15(1):95–107, 2012. doi:10.1002/sys.20197.

[110] Mark W. Maier. Architecting principles for systems-of-systems. *Systems Engineering*, 1(4):267–284, 1998. doi:10.1002/(SICI)1520-6858(1998)1:4<267::AID-SYS3>3.0.CO;2-D.

[111] Mark W. Maier. System and software architecture reconciliation. *Systems Engineering*, 9(2):146–159, 2006. doi:10.1002/sys.20050.

[112] M.W. Maier. On architecting and intelligent transport systems. *IEEE Transactions on Aerospace and Electronic Systems*, 33(2):610–625, April 1997. doi:10.1109/7.588379.

[113] M.W. Maier. Research challenges for systems-of-systems. In *Proc. IEEE Int. Conf. on Systems, Man and Cybernetics*, volume 4 of *SMC*, pages 3149 – 3154, 2005. doi:10.1109/ICSMC.2005.1571630.

[114] Ruth Malan and Dana Bredemeyer. Less is more with minimalist architecture. *IT Pro*, 4(5):47–48, Sep-Oct 2002. doi:10.1109/MITP.2002.1041178.

[115] Joe Manas. Test perspectives for architecture. In *Proc. 17th Ann. NDIA Systems Eng. Conf.* National Defense Industry Association, 2014. URL: http://www.dtic.mil/ndia/2014/system/17006ThursTrack6Manas.pdf [cited 1 June 2017].

[116] Matthew L. Massie, Brent N. Chun, and David E. Culler. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(7):817–840, 2004. doi:10.1016/j.parco.2004.04.001.

[117] David G. Messerschmitt and Clemens Szyperski. *Software Ecosystem: Understanding an Indispensable Technology and Industry*. MIT Press, Cambridge, MA, USA, 2003.

[118] Ministry of Defence. MOD architecture framework [online]. 2012. URL: https://www.gov.uk/guidance/mod-architecture-framework [cited 1 Jan 2017].

[119] Marco Mori, Andrea Ceccarelli, Paolo Lollini, Bernhard Frömel, Francesco Brancati, and Andrea Bondavalli. Systems-of-systems modeling using a comprehensive viewpoint-based sysml profile. *Journal of Software: Evolution and Process*, Early View:e1878–n/a, 2017. doi:10.1002/smr.1878.

[120] Robert Nord, Paul C. Clements, David Emery, and Rich Hilliard. A structured approach for reviewing architecture documentation. Technical Note CMU/SEI-2009-TN-030, Software Engineering Institute, Pittsburgh, PA, December 2009. URL: http://resources.sei.cmu.edu/library/asset-view.cfm?assetID=9045# [cited 22 Dec 2013].

[121] Robert Nord, James McHale, and Felix Bachmann. Combining architecture-centric engineering with the team software process. Technical Report CMU/SEI-2010-TR-031, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2010. URL: http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=9649 [cited 1 March 2017].

[122] Linda Northrop, Peter Feiler, Richard P. Gabriel, John Goodenough, Rick Linger, Tom Longstaff, Rick Kazman, Mark Klein, Douglas Schmidt, Kevin Sullivan, and Kurt Wallnau. Ultra-large-scale systems: The software challenge of the future. Technical report, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2006. URL: http://resources.sei.cmu.edu/library/asset-view.cfm?assetid=30519 [cited 1 Apr 2017].

[123] Linda M. Northrop and Paul C. Clements. A framework for software product line practice, version 5.0. Online, Software Engineering Institute, 2012. URL: http://www.sei.cmu.edu/productlines/frame_report/index.html [cited 28 Feb 2013].

[124] Michael Nygard. *Release It! Design and Deploy Production-ready Software*. Pragmatic Bookshelf, 2007.

[125] ODUSD(A&T). Systems engineering guide for systems of systems, version 1.0. Guide, US Department of Defense, 2008. URL: http://www.acq.osd.mil/se/docs/SE-Guide-for-SoS.pdf.

[126] C. Pahl, S. Giesecke, and W. Hasselbring. An ontology-based approach for modelling architectural styles. In *Proc. 2007 European Conference on Software Architecture*, ECSA 2007, pages 60–75, 2007. doi:10.1007/978-3-540-75132-8_6.

[127] David L. Parnas. On the criteria to be used in decomposing systems into modules. *Comm. ACM*, 15(12):1053–1058, December 1972. doi:10.1145/361598.361623.

[128] David L. Parnas and David M. Weiss. Active design reviews: principles and practices. In *Proc. 8th Int. Conf. on Software Engineering*, ICSE '85, pages 132–136, August 1985.

[129] Swapnil Patil, Milo Polte, Kai Ren, Wittawat Tantisiriroj, Lin Xiao, Julio López, Garth Gibson, Adam Fuchs, and Billie Rinaldi. YCSB++: Benchmarking and performance debugging advanced features in scalable table stores. In *Proc. 2nd ACM Symp. on Cloud Computing*, SOCC '11, pages 9:1–9:14, 2011. doi:10.1145/2038916.2038925.

[130] Eltjo R. Poort and Hans van Vliet. RCDA: Architecting as a risk- and cost management discipline. *Journal of Systems and Software*, 85(9):1995–2013, 2012. doi:10.1016/j.jss.2012.03.071.

[131] Mary Poppendieck and Tom Poppendieck. *Lean Software Development: An Agile Toolkit*. Addison-Wesley Professional, 2003.

[132] William Pritchett. Application of a ground system architecture framework using SysML. In *Proc. 16th Ann. NDIA Systems Eng. Conf.* National Defense Industry Association, 2013. URL: http://www.dtic.mil/ndia/2013/system/W16096_Pritchett.pdf [cited 1 June 2017].

[133] Samuel T. Redwine, Jr. and William E. Riddle. Software technology maturation. In *Proc. 8th Int. Conf. on Software Engineering*, ICSE '85, pages 189–200, 1985.

[134] Kai Ren, Julio López, and Garth Gibson. Otus: Resource attribution in data-intensive clusters. In *Proc. 2nd Int. Workshop on MapReduce and its Applications*, MapReduce'11, June 2011. URL: http://www.pdl.cmu.edu/PDL-FTP/Monitoring/map611-ren.pdf.

[135] Thomas Rischbeck and Thomas Erl. *SOA Design Patterns*. The Prentice Hall Service-Oriented Computing Series from Thomas Erl. Prentice Hall, 1st edition, January 2009.

[136] J.W. Ross, P. Weill, and D.C. Robertson. *Enterprise Architecture as Strategy*. Harvard Business School Press, 2006.

[137] Nick Rozanski and Eoin Woods. *Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives*. Addison-Wesley, 2005.

[138] Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2):131–164, April 2009. doi:10.1007/s10664-008-9102-8.

[139] Pramod J. Sadalage and Martin Fowler. *NoSQL Distilled*. Addison-Wesley Professional, 2012.

[140] Carolyn B. Seaman. *Guide to Advanced Empirical Software Engineering*, chapter Qualitative Methods. Springer, 2008.

[141] M. Shahin, P. Liang, and M.R. Khayyambashi. Architectural design decision: Existing models and tools. In *Proc. 7th Working IEEE/IFIP Conf. on Software Architecture*, WICSA'09, pages 293–296, 2009. doi:10.1109/WICSA.2009.5290823.

[142] Mary Shaw. The coming-of-age of software archititecture research. In *Proc. 23rd Int. Conf. on Software Eng.*, ICSE '01, pages 656–664a, 2001. URL: http://dl.acm.org/citation.cfm?id=381473.381549 [cited 27 Feb 2017].

[143] Mary Shaw. What makes good research in software engineering? *Int. J. of Software Tools for Technology Transfer*, 4(1):1–7, October 2002. doi:10.1007/s10009-002-0083-4.

[144] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[145] Eduardo Silva, Everton Cavalcante, Thais Batista, Flavio Oquendo, Flavia C. Delicato, and Paulo F. Pires. On the characterization of missions of systems-of-systems. In *Proc. of the 2014 European Conf. on Software Architecture Workshops*, ECSAW '14, pages 26:1–26:8, 2014. doi:10.1145/2642803.2642829.

[146] Valerie Sitterle. Cross-scale resilience: Bridging system of systems and consitituent systems engineering and analysis. In *Proc. 19th Ann. NDIA Systems Eng. Conf.* National Defense Industry Association, 2016. URL: http://www.dtic.mil/ndia/2016/systems/18864_ValerieSitterle.pdf [cited 1 June 2017].

[147] Jim Smith, Patrick Place, Marc Novakouski, and David Carney. Examining the role of context in data interoperability. In *Proc. 14th Ann. NDIA Systems Eng. Conf.*, San Diego, CA, 2011. National Defense Industry Association. URL: http://www.dtic.mil/ndia/2011system/13101_SmithWednesday.pdf [cited 27 Sep 2013].

[148] Antony Tang and Hans van Vliet. *Software Architecture Knowledge Management: Theory and Practice*, chapter Software Architecture Design Reasoning, pages 155–174. Springer, Heidelberg, 2009.

[149] Vernon Turner, John F. Gantz, David Reinsel, and Stephen Minton. The digital universe of opportunities: Rich data and the increasing value of the internet of things. White Paper IDC_1672, International Data Corporation, Framingham, MA, USA, April 2014. URL: http://idcdocserv.com/1678 [cited 10 Nov 2014].

[150] Marcel A. P. M. van den Bosch, Marlies E. van Steenbergen, Marcel Lamaitre, and Rik Bos. A selection-method for enterprise application integration solutions. In *Proc. 9th Int. Conf. on Business Informatics Research*, BIR 2010, pages 176–187, Sept 2010. doi:10.1007/978-3-642-16101-8_14.

[151] J.S. van der Ven, A. Jansen, J. Nijhuis, and Jan Bosch. *Rationale Management in Software Engineering*, chapter Design decisions: The Bridge between Rationale and Architecture, pages 329–346. Springer, 2006. doi:10.1007/978-3-540-30998-7_16.

[152] Werner Vogels. Amazon.com: E-commerce at interplanetary scale. In *Proc. O'Reilly Emerging Technology Conf.*, 2005. URL: http://conferences.oreillynet.com/cs/et2005/view/e_sess/5974 [cited 7 Nov 2014].

[153] Cory Watson. Observability at Twitter [online]. 2013. URL: https://blog.twitter.com/2013/observability-at-twitter [cited 10 Nov 2014].

[154] Janet Weiner and Nathan Bronson. Facebook's top open data problems [online]. 2014. URL: https://research.facebook.com/blog/1522692927972019/facebook-s-top-open-data-problems/ [cited 10 Nov 2014].

[155] C. A. Welty and D. A. Ferrucci. A formal ontology for re-use of software architecture documents. In *Proc. 14th IEEE Int. Conf. on Automated Software Eng.*, 1999.

[156] Roel J. Wieringa. *Design Science Methodology for Information Systems and Software Engineering*. Springer, Heidelberg, Germany, 2014.

[157] Wiki. Proto pattern [online]. undated. URL: http://c2.com/cgi/wiki?ProtoPattern [cited 18 December 2012].

[158] Bruce Wong and Christos Kalantzis. A state of Xen - chaos monkey & Cassandra [online]. 2014. URL: http://techblog.netflix.com/2014/10/a-state-of-xen-chaos-monkey-cassandra.html [cited 30 Oct 2014].

[159] E. Woods and N. Rozanski. The system context architectural viewpoint. In *Proc. Joint European Conf. on Software Arch. and Working IEEE/IFIP Conf. on Software Arch.*, WICSA/ECSA'09, pages 333–336, Sept 2009. doi:10.1109/WICSA.2009.5290673.

[160] Jianxiong Yin, Peng Sun, Yonggang Wen, Haigang Gong, Ming Liu, Xuelong Li, Haipeng You, Jinqi Gao, and Cynthia Lin. Cloud3DView: An interactive tool for cloud data center operations. In *Proc. ACM Conf. on SIGCOMM*, SIGCOMM '13, pages 499–500, 2013. doi:10.1145/2486001.2491704.

[161] R.K. Yin. *Case Study Research: Design and Methods*. Applied Social Research Methods Series. Sage Publications, 3rd edition, 2003.

[162] Jia Yu and Rajkumar Buyya. A taxonomy of scientific workflow systems for grid computing. *J. of Grid Computing*, 3(3-4):171–200, September 2005. doi:10.1145/1084805.1084814.

[163] J. Zahid, A. Sattar, and M. Faridi. Unsolved tricky issues on COTS selection and evaluation. *Global Journal of Computer Science and Technology*, 12(10-D), 2012.

[164] He Zhang, Muhammad Ali Babar, and Paolo Tell. Identifying relevant studies in software engineering. *Information and Software Technology*, 53(6):625–637, June 2011. doi:10.1016/j.infsof.2010.12.010.

[165] William Zola. 6 rules of thumb for MongoDB schema design: Part 1 [online]. 2014. URL: http://blog.mongodb.org/post/87200945828/6-rules-of-thumb-for-mongodb-schema-design-part-1 [cited 18 Sep 2014].

SIKS Dissertation Series

2011 01 Botond Cseke (RUN), Variational Algorithms for Bayesian Inference in Latent Gaussian Models
02 Nick Tinnemeier (UU), Organizing Agent Organizations. Syntax and Operational Semantics of an Organization-Oriented Programming Language
03 Jan Martijn van der Werf (TUE), Compositional Design and Verification of Component-Based Information Systems
04 Hado van Hasselt (UU), Insights in Reinforcement Learning; Formal analysis and empirical evaluation of temporal-difference
05 Bas van der Raadt (VU), Enterprise Architecture Coming of Age - Increasing the Performance of an Emerging Discipline.
06 Yiwen Wang (TUE), Semantically-Enhanced Recommendations in Cultural Heritage
07 Yujia Cao (UT), Multimodal Information Presentation for High Load Human Computer Interaction
08 Nieske Vergunst (UU), BDI-based Generation of Robust Task-Oriented Dialogues
09 Tim de Jong (OU), Contextualised Mobile Media for Learning
10 Bart Bogaert (UvT), Cloud Content Contention
11 Dhaval Vyas (UT), Designing for Awareness: An Experience-focused HCI Perspective
12 Carmen Bratosin (TUE), Grid Architecture for Distributed Process Mining
13 Xiaoyu Mao (UvT), Airport under Control. Multiagent Scheduling for Airport Ground Handling
14 Milan Lovric (EUR), Behavioral Finance and Agent-Based Artificial Markets
15 Marijn Koolen (UvA), The Meaning of Structure: the Value of Link Evidence for Information Retrieval
16 Maarten Schadd (UM), Selective Search in Games of Different Complexity
17 Jiyin He (UVA), Exploring Topic Structure: Coherence, Diversity and Relatedness
18 Mark Ponsen (UM), Strategic Decision-Making in complex games
19 Ellen Rusman (OU), The Mind's Eye on Personal Profiles
20 Qing Gu (VU), Guiding service-oriented software engineering - A view-based approach
21 Linda Terlouw (TUD), Modularization and Specification of Service-Oriented Systems
22 Junte Zhang (UVA), System Evaluation of Archival Description and Access
23 Wouter Weerkamp (UVA), Finding People and their Utterances in Social Media
24 Herwin van Welbergen (UT), Behavior Generation for Interpersonal Coordination with Virtual Humans On Specifying, Scheduling and Realizing Multimodal Virtual Human Behavior

25 Syed Waqar ul Qounain Jaffry (VU), Analysis and Validation of Models for Trust Dynamics
26 Matthijs Aart Pontier (VU), Virtual Agents for Human Communication - Emotion Regulation and Involvement-Distance Trade-Offs in Embodied Conversational Agents and Robots
27 Aniel Bhulai (VU), Dynamic website optimization through autonomous management of design patterns
28 Rianne Kaptein (UVA), Effective Focused Retrieval by Exploiting Query Context and Document Structure
29 Faisal Kamiran (TUE), Discrimination-aware Classification
30 Egon van den Broek (UT), Affective Signal Processing (ASP): Unraveling the mystery of emotions
31 Ludo Waltman (EUR), Computational and Game-Theoretic Approaches for Modeling Bounded Rationality
32 Nees-Jan van Eck (EUR), Methodological Advances in Bibliometric Mapping of Science
33 Tom van der Weide (UU), Arguing to Motivate Decisions
34 Paolo Turrini (UU), Strategic Reasoning in Interdependence: Logical and Game-theoretical Investigations
35 Maaike Harbers (UU), Explaining Agent Behavior in Virtual Training
36 Erik van der Spek (UU), Experiments in serious game design: a cognitive approach
37 Adriana Burlutiu (RUN), Machine Learning for Pairwise Data, Applications for Preference Learning and Supervised Network Inference
38 Nyree Lemmens (UM), Bee-inspired Distributed Optimization
39 Joost Westra (UU), Organizing Adaptation using Agents in Serious Games
40 Viktor Clerc (VU), Architectural Knowledge Management in Global Software Development
41 Luan Ibraimi (UT), Cryptographically Enforced Distributed Data Access Control
42 Michal Sindlar (UU), Explaining Behavior through Mental State Attribution
43 Henk van der Schuur (UU), Process Improvement through Software Operation Knowledge
44 Boris Reuderink (UT), Robust Brain-Computer Interfaces
45 Herman Stehouwer (UvT), Statistical Language Models for Alternative Sequence Selection
46 Beibei Hu (TUD), Towards Contextualized Information Delivery: A Rule-based Architecture for the Domain of Mobile Police Work
47 Azizi Bin Ab Aziz (VU), Exploring Computational Models for Intelligent Support of Persons with Depression
48 Mark Ter Maat (UT), Response Selection and Turn-taking for a Sensitive Artificial Listening Agent

49 Andreea Niculescu (UT), Conversational interfaces for task-oriented spoken dialogues: design aspects influencing interaction quality

2012 01 Terry Kakeeto (UvT), Relationship Marketing for SMEs in Uganda
02 Muhammad Umair (VU), Adaptivity, emotion, and Rationality in Human and Ambient Agent Models
03 Adam Vanya (VU), Supporting Architecture Evolution by Mining Software Repositories
04 Jurriaan Souer (UU), Development of Content Management System-based Web Applications
05 Marijn Plomp (UU), Maturing Interorganisational Information Systems
06 Wolfgang Reinhardt (OU), Awareness Support for Knowledge Workers in Research Networks
07 Rianne van Lambalgen (VU), When the Going Gets Tough: Exploring Agent-based Models of Human Performance under Demanding Conditions
08 Gerben de Vries (UVA), Kernel Methods for Vessel Trajectories
09 Ricardo Neisse (UT), Trust and Privacy Management Support for Context-Aware Service Platforms
10 David Smits (TUE), Towards a Generic Distributed Adaptive Hypermedia Environment
11 J.C.B. Rantham Prabhakara (TUE), Process Mining in the Large: Preprocessing, Discovery, and Diagnostics
12 Kees van der Sluijs (TUE), Model Driven Design and Data Integration in Semantic Web Information Systems
13 Suleman Shahid (UvT), Fun and Face: Exploring non-verbal expressions of emotion during playful interactions
14 Evgeny Knutov (TUE), Generic Adaptation Framework for Unifying Adaptive Web-based Systems
15 Natalie van der Wal (VU), Social Agents. Agent-Based Modelling of Integrated Internal and Social Dynamics of Cognitive and Affective Processes.
16 Fiemke Both (VU), Helping people by understanding them - Ambient Agents supporting task execution and depression treatment
17 Amal Elgammal (UvT), Towards a Comprehensive Framework for Business Process Compliance
18 Eltjo Poort (VU), Improving Solution Architecting Practices
19 Helen Schonenberg (TUE), What's Next? Operational Support for Business Process Execution
20 Ali Bahramisharif (RUN), Covert Visual Spatial Attention, a Robust Paradigm for Brain-Computer Interfacing
21 Roberto Cornacchia (TUD), Querying Sparse Matrices for Information Retrieval
22 Thijs Vis (UvT), Intelligence, politie en veiligheidsdienst: verenigbare grootheden?

23 Christian Muehl (UT), Toward Affective Brain-Computer Interfaces: Exploring the Neurophysiology of Affect during Human Media Interaction
24 Laurens van der Werff (UT), Evaluation of Noisy Transcripts for Spoken Document Retrieval
25 Silja Eckartz (UT), Managing the Business Case Development in Inter-Organizational IT Projects: A Methodology and its Application
26 Emile de Maat (UVA), Making Sense of Legal Text
27 Hayrettin Gurkok (UT), Mind the Sheep! User Experience Evaluation & Brain-Computer Interface Games
28 Nancy Pascall (UvT), Engendering Technology Empowering Women
29 Almer Tigelaar (UT), Peer-to-Peer Information Retrieval
30 Alina Pommeranz (TUD), Designing Human-Centered Systems for Reflective Decision Making
31 Emily Bagarukayo (RUN), A Learning by Construction Approach for Higher Order Cognitive Skills Improvement, Building Capacity and Infrastructure
32 Wietske Visser (TUD), Qualitative multi-criteria preference representation and reasoning
33 Rory Sie (OUN), Coalitions in Cooperation Networks (COCOON)
34 Pavol Jancura (RUN), Evolutionary analysis in PPI networks and applications
35 Evert Haasdijk (VU), Never Too Old To Learn – On-line Evolution of Controllers in Swarm- and Modular Robotics
36 Denis Ssebugwawo (RUN), Analysis and Evaluation of Collaborative Modeling Processes
37 Agnes Nakakawa (RUN), A Collaboration Process for Enterprise Architecture Creation
38 Selmar Smit (VU), Parameter Tuning and Scientific Testing in Evolutionary Algorithms
39 Hassan Fatemi (UT), Risk-aware design of value and coordination networks
40 Agus Gunawan (UvT), Information Access for SMEs in Indonesia
41 Sebastian Kelle (OU), Game Design Patterns for Learning
42 Dominique Verpoorten (OU), Reflection Amplifiers in self-regulated Learning
43 Withdrawn
44 Anna Tordai (VU), On Combining Alignment Techniques
45 Benedikt Kratz (UvT), A Model and Language for Business-aware Transactions
46 Simon Carter (UVA), Exploration and Exploitation of Multilingual Data for Statistical Machine Translation
47 Manos Tsagkias (UVA), Mining Social Media: Tracking Content and Predicting Behavior
48 Jorn Bakker (TUE), Handling Abrupt Changes in Evolving Time-series Data
49 Michael Kaisers (UM), Learning against Learning - Evolutionary dynamics of reinforcement learning algorithms in strategic interactions

50 Steven van Kervel (TUD), Ontologogy driven Enterprise Information Systems Engineering
51 Jeroen de Jong (TUD), Heuristics in Dynamic Sceduling; a practical framework with a case study in elevator dispatching

---

2013 01 Viorel Milea (EUR), News Analytics for Financial Decision Support
02 Erietta Liarou (CWI), MonetDB/DataCell: Leveraging the Column-store Database Technology for Efficient and Scalable Stream Processing
03 Szymon Klarman (VU), Reasoning with Contexts in Description Logics
04 Chetan Yadati (TUD), Coordinating autonomous planning and scheduling
05 Dulce Pumareja (UT), Groupware Requirements Evolutions Patterns
06 Romulo Goncalves (CWI), The Data Cyclotron: Juggling Data and Queries for a Data Warehouse Audience
07 Giel van Lankveld (UvT), Quantifying Individual Player Differences
08 Robbert-Jan Merk (VU), Making enemies: cognitive modeling for opponent agents in fighter pilot simulators
09 Fabio Gori (RUN), Metagenomic Data Analysis: Computational Methods and Applications
10 Jeewanie Jayasinghe Arachchige (UvT), A Unified Modeling Framework for Service Design.
11 Evangelos Pournaras (TUD), Multi-level Reconfigurable Self-organization in Overlay Services
12 Marian Razavian (VU), Knowledge-driven Migration to Services
13 Mohammad Safiri (UT), Service Tailoring: User-centric creation of integrated IT-based homecare services to support independent living of elderly
14 Jafar Tanha (UVA), Ensemble Approaches to Semi-Supervised Learning Learning
15 Daniel Hennes (UM), Multiagent Learning - Dynamic Games and Applications
16 Eric Kok (UU), Exploring the practical benefits of argumentation in multi-agent deliberation
17 Koen Kok (VU), The PowerMatcher: Smart Coordination for the Smart Electricity Grid
18 Jeroen Janssens (UvT), Outlier Selection and One-Class Classification
19 Renze Steenhuizen (TUD), Coordinated Multi-Agent Planning and Scheduling
20 Katja Hofmann (UvA), Fast and Reliable Online Learning to Rank for Information Retrieval
21 Sander Wubben (UvT), Text-to-text generation by monolingual machine translation
22 Tom Claassen (RUN), Causal Discovery and Logic
23 Patricio de Alencar Silva (UvT), Value Activity Monitoring
24 Haitham Bou Ammar (UM), Automated Transfer in Reinforcement Learning

25 Agnieszka Anna Latoszek-Berendsen (UM), Intention-based Decision Support. A new way of representing and implementing clinical guidelines in a Decision Support System
26 Alireza Zarghami (UT), Architectural Support for Dynamic Homecare Service Provisioning
27 Mohammad Huq (UT), Inference-based Framework Managing Data Provenance
28 Frans van der Sluis (UT), When Complexity becomes Interesting: An Inquiry into the Information eXperience
29 Iwan de Kok (UT), Listening Heads
30 Joyce Nakatumba (TUE), Resource-Aware Business Process Management: Analysis and Support
31 Dinh Khoa Nguyen (UvT), Blueprint Model and Language for Engineering Cloud Applications
32 Kamakshi Rajagopal (OUN), Networking For Learning; The role of Networking in a Lifelong Learner's Professional Development
33 Qi Gao (TUD), User Modeling and Personalization in the Microblogging Sphere
34 Kien Tjin-Kam-Jet (UT), Distributed Deep Web Search
35 Abdallah El Ali (UvA), Minimal Mobile Human Computer Interaction
36 Than Lam Hoang (TUe), Pattern Mining in Data Streams
37 Dirk Börner (OUN), Ambient Learning Displays
38 Eelco den Heijer (VU), Autonomous Evolutionary Art
39 Joop de Jong (TUD), A Method for Enterprise Ontology based Design of Enterprise Information Systems
40 Pim Nijssen (UM), Monte-Carlo Tree Search for Multi-Player Games
41 Jochem Liem (UVA), Supporting the Conceptual Modelling of Dynamic Systems: A Knowledge Engineering Perspective on Qualitative Reasoning
42 Léon Planken (TUD), Algorithms for Simple Temporal Reasoning
43 Marc Bron (UVA), Exploration and Contextualization through Interaction and Concepts

---

2014 01 Nicola Barile (UU), Studies in Learning Monotone Models from Data
02 Fiona Tuliyano (RUN), Combining System Dynamics with a Domain Modeling Method
03 Sergio Raul Duarte Torres (UT), Information Retrieval for Children: Search Behavior and Solutions
04 Hanna Jochmann-Mannak (UT), Websites for children: search strategies and interface design - Three studies on children's search performance and evaluation
05 Jurriaan van Reijsen (UU), Knowledge Perspectives on Advancing Dynamic Capability
06 Damian Tamburri (VU), Supporting Networked Software Development

07 Arya Adriansyah (TUE), Aligning Observed and Modeled Behavior
08 Samur Araujo (TUD), Data Integration over Distributed and Heterogeneous Data Endpoints
09 Philip Jackson (UvT), Toward Human-Level Artificial Intelligence: Representation and Computation of Meaning in Natural Language
10 Ivan Salvador Razo Zapata (VU), Service Value Networks
11 Janneke van der Zwaan (TUD), An Empathic Virtual Buddy for Social Support
12 Willem van Willigen (VU), Look Ma, No Hands: Aspects of Autonomous Vehicle Control
13 Arlette van Wissen (VU), Agent-Based Support for Behavior Change: Models and Applications in Health and Safety Domains
14 Yangyang Shi (TUD), Language Models With Meta-information
15 Natalya Mogles (VU), Agent-Based Analysis and Support of Human Functioning in Complex Socio-Technical Systems: Applications in Safety and Healthcare
16 Krystyna Milian (VU), Supporting trial recruitment and design by automatically interpreting eligibility criteria
17 Kathrin Dentler (VU), Computing healthcare quality indicators automatically: Secondary Use of Patient Data and Semantic Interoperability
18 Mattijs Ghijsen (UVA), Methods and Models for the Design and Study of Dynamic Agent Organizations
19 Vinicius Ramos (TUE), Adaptive Hypermedia Courses: Qualitative and Quantitative Evaluation and Tool Support
20 Mena Habib (UT), Named Entity Extraction and Disambiguation for Informal Text: The Missing Link
21 Kassidy Clark (TUD), Negotiation and Monitoring in Open Environments
22 Marieke Peeters (UU), Personalized Educational Games - Developing agent-supported scenario-based training
23 Eleftherios Sidirourgos (UvA/CWI), Space Efficient Indexes for the Big Data Era
24 Davide Ceolin (VU), Trusting Semi-structured Web Data
25 Martijn Lappenschaar (RUN), New network models for the analysis of disease interaction
26 Tim Baarslag (TUD), What to Bid and When to Stop
27 Rui Jorge Almeida (EUR), Conditional Density Models Integrating Fuzzy and Probabilistic Representations of Uncertainty
28 Anna Chmielowiec (VU), Decentralized k-Clique Matching
29 Jaap Kabbedijk (UU), Variability in Multi-Tenant Enterprise Software
30 Peter de Cock (UvT), Anticipating Criminal Behaviour
31 Leo van Moergestel (UU), Agent Technology in Agile Multiparallel Manufacturing and Product Support
32 Naser Ayat (UvA), On Entity Resolution in Probabilistic Data

33 Tesfa Tegegne (RUN), Service Discovery in eHealth
34 Christina Manteli (VU), The Effect of Governance in Global Software Development: Analyzing Transactive Memory Systems.
35 Joost van Ooijen (UU), Cognitive Agents in Virtual Worlds: A Middleware Design Approach
36 Joos Buijs (TUE), Flexible Evolutionary Algorithms for Mining Structured Process Models
37 Maral Dadvar (UT), Experts and Machines United Against Cyberbullying
38 Danny Plass-Oude Bos (UT), Making brain-computer interfaces better: improving usability through post-processing.
39 Jasmina Maric (UvT), Web Communities, Immigration, and Social Capital
40 Walter Omona (RUN), A Framework for Knowledge Management Using ICT in Higher Education
41 Frederic Hogenboom (EUR), Automated Detection of Financial Events in News Text
42 Carsten Eijckhof (CWI/TUD), Contextual Multidimensional Relevance Models
43 Kevin Vlaanderen (UU), Supporting Process Improvement using Method Increments
44 Paulien Meesters (UvT), Intelligent Blauw. Met als ondertitel: Intelligence-gestuurde politiezorg in gebiedsgebonden eenheden.
45 Birgit Schmitz (OUN), Mobile Games for Learning: A Pattern-Based Approach
46 Ke Tao (TUD), Social Web Data Analytics: Relevance, Redundancy, Diversity
47 Shangsong Liang (UVA), Fusion and Diversification in Information Retrieval

---

2015 01 Niels Netten (UvA), Machine Learning for Relevance of Information in Crisis Response
02 Faiza Bukhsh (UvT), Smart auditing: Innovative Compliance Checking in Customs Controls
03 Twan van Laarhoven (RUN), Machine learning for network data
04 Howard Spoelstra (OUN), Collaborations in Open Learning Environments
05 Christoph Bösch (UT), Cryptographically Enforced Search Pattern Hiding
06 Farideh Heidari (TUD), Business Process Quality Computation - Computing Non-Functional Requirements to Improve Business Processes
07 Maria-Hendrike Peetz (UvA), Time-Aware Online Reputation Analysis
08 Jie Jiang (TUD), Organizational Compliance: An agent-based model for designing and evaluating organizational interactions
09 Randy Klaassen (UT), HCI Perspectives on Behavior Change Support Systems
10 Henry Hermans (OUN), OpenU: design of an integrated system to support lifelong learning

11 Yongming Luo (TUE), Designing algorithms for big graph datasets: A study of computing bisimulation and joins
12 Julie M. Birkholz (VU), Modi Operandi of Social Network Dynamics: The Effect of Context on Scientific Collaboration Networks
13 Giuseppe Procaccianti (VU), Energy-Efficient Software
14 Bart van Straalen (UT), A cognitive approach to modeling bad news conversations
15 Klaas Andries de Graaf (VU), Ontology-based Software Architecture Documentation
16 Changyun Wei (UT), Cognitive Coordination for Cooperative Multi-Robot Teamwork
17 André van Cleeff (UT), Physical and Digital Security Mechanisms: Properties, Combinations and Trade-offs
18 Holger Pirk (CWI), Waste Not, Want Not! - Managing Relational Data in Asymmetric Memories
19 Bernardo Tabuenca (OUN), Ubiquitous Technology for Lifelong Learners
20 Lois Vanhée (UU), Using Culture and Values to Support Flexible Coordination
21 Sibren Fetter (OUN), Using Peer-Support to Expand and Stabilize Online Learning
22 Zhemin Zhu (UT), Co-occurrence Rate Networks
23 Luit Gazendam (VU), Cataloguer Support in Cultural Heritage
24 Richard Berendsen (UVA), Finding People, Papers, and Posts: Vertical Search Algorithms and Evaluation
25 Steven Woudenberg (UU), Bayesian Tools for Early Disease Detection
26 Alexander Hogenboom (EUR), Sentiment Analysis of Text Guided by Semantics and Structure
27 Sándor Héman (CWI), Updating compressed colomn stores
28 Janet Bagorogoza (TiU), Knowledge Management and High Performance; The Uganda Financial Institutions Model for HPO
29 Hendrik Baier (UM), Monte-Carlo Tree Search Enhancements for One-Player and Two-Player Domains
30 Kiavash Bahreini (OU), Real-time Multimodal Emotion Recognition in E-Learning
31 Yakup Koç (TUD), On the robustness of Power Grids
32 Jerome Gard (UL), Corporate Venture Management in SMEs
33 Frederik Schadd (TUD), Ontology Mapping with Auxiliary Resources
34 Victor de Graaf (UT), Gesocial Recommender Systems
35 Jungxao Xu (TUD), Affective Body Language of Humanoid Robots: Perception and Effects in Human Robot Interaction

2016 01 Syed Saiden Abbas (RUN), Recognition of Shapes by Humans and Machines
02 Michiel Christiaan Meulendijk (UU), Optimizing medication reviews through decision support: prescribing a better pill to swallow

03 Maya Sappelli (RUN), Knowledge Work in Context: User Centered Knowledge Worker Support
04 Laurens Rietveld (VU), Publishing and Consuming Linked Data
05 Evgeny Sherkhonov (UVA), Expanded Acyclic Queries: Containment and an Application in Explaining Missing Answers
06 Michel Wilson (TUD), Robust scheduling in an uncertain environment
07 Jeroen de Man (VU), Measuring and modeling negative emotions for virtual training
08 Matje van de Camp (TiU), A Link to the Past: Constructing Historical Social Networks from Unstructured Data
09 Archana Nottamkandath (VU), Trusting Crowdsourced Information on Cultural Artefacts
10 George Karafotias (VUA), Parameter Control for Evolutionary Algorithms
11 Anne Schuth (UVA), Search Engines that Learn from Their Users
12 Max Knobbout (UU), Logics for Modelling and Verifying Normative Multi-Agent Systems
13 Nana Baah Gyan (VU), The Web, Speech Technologies and Rural Development in West Africa - An ICT4D Approach
14 Ravi Khadka (UU), Revisiting Legacy Software System Modernization
15 Steffen Michels (RUN), Hybrid Probabilistic Logics - Theoretical Aspects, Algorithms and Experiments
16 Guangliang Li (UVA), Socially Intelligent Autonomous Agents that Learn from Human Reward
17 Berend Weel (VU), Towards Embodied Evolution of Robot Organisms
18 Albert Meroño Peñuela (VU), Refining Statistical Data on the Web
19 Julia Efremova (Tu/e), Mining Social Structures from Genealogical Data
20 Daan Odijk (UVA), Context & Semantics in News & Web Search
21 Alejandro Moreno Célleri (UT), From Traditional to Interactive Playspaces: Automatic Analysis of Player Behavior in the Interactive Tag Playground
22 Grace Lewis (VU), Software Architecture Strategies for Cyber-Foraging Systems
23 Fei Cai (UVA), Query Auto Completion in Information Retrieval
24 Brend Wanders (UT), Repurposing and Probabilistic Integration of Data; An Iterative and data model independent approach
25 Julia Kiseleva (TU/e), Using Contextual Information to Understand Searching and Browsing Behavior
26 Dilhan Thilakarathne (VU), In or Out of Control: Exploring Computational Models to Study the Role of Human Awareness and Control in Behavioural Choices, with Applications in Aviation and Energy Management Domains
27 Wen Li (TUD), Understanding Geo-spatial Information on Social Media
28 Mingxin Zhang (TUD), Large-scale Agent-based Social Simulation - A study on epidemic prediction and control

29 Nicolas Höning (TUD), Peak reduction in decentralised electricity systems - Markets and prices for flexible planning
30 Ruud Mattheij (UvT), The Eyes Have It
31 Mohammad Khelghati (UT), Deep web content monitoring
32 Eelco Vriezekolk (UT), Assessing Telecommunication Service Availability Risks for Crisis Organisations
33 Peter Bloem (UVA), Single Sample Statistics, exercises in learning from just one example
34 Dennis Schunselaar (TUE), Configurable Process Trees: Elicitation, Analysis, and Enactment
35 Zhaochun Ren (UVA), Monitoring Social Media: Summarization, Classification and Recommendation
36 Daphne Karreman (UT), Beyond R2D2: The design of nonverbal interaction behavior optimized for robot-specific morphologies
37 Giovanni Sileno (UvA), Aligning Law and Action - a conceptual and computational inquiry
38 Andrea Minuto (UT), Materials that Matter - Smart Materials meet Art & Interaction Design
39 Merijn Bruijnes (UT), Believable Suspect Agents; Response and Interpersonal Style Selection for an Artificial Suspect
40 Christian Detweiler (TUD), Accounting for Values in Design
41 Thomas King (TUD), Governing Governance: A Formal Framework for Analysing Institutional Design and Enactment Governance
42 Spyros Martzoukos (UVA), Combinatorial and Compositional Aspects of Bilingual Aligned Corpora
43 Saskia Koldijk (RUN), Context-Aware Support for Stress Self-Management: From Theory to Practice
44 Thibault Sellam (UVA), Automatic Assistants for Database Exploration
45 Bram van de Laar (UT), Experiencing Brain-Computer Interface Control
46 Jorge Gallego Perez (UT), Robots to Make you Happy
47 Christina Weber (UL), Real-time foresight - Preparedness for dynamic innovation networks
48 Tanja Buttler (TUD), Collecting Lessons Learned
49 Gleb Polevoy (TUD), Participation and Interaction in Projects. A Game-Theoretic Analysis
50 Yan Wang (UVT), The Bridge of Dreams: Towards a Method for Operational Performance Alignment in IT-enabled Service Supply Chains

---

2017 01 Jan-Jaap Oerlemans (UL), Investigating Cybercrime
02 Sjoerd Timmer (UU), Designing and Understanding Forensic Bayesian Networks using Argumentation
03 Daniël Harold Telgen (UU), Grid Manufacturing; A Cyber-Physical Approach with Autonomous Products and Reconfigurable Manufacturing Machines

04 Mrunal Gawade (CWI), Multi-core Parallelism in a Column-store
05 Mahdieh Shadi (UVA), Collaboration Behavior
06 Damir Vandic (EUR), Intelligent Information Systems for Web Product Search
07 Roel Bertens (UU), Insight in Information: from Abstract to Anomaly
08 Rob Konijn (VU) , Detecting Interesting Differences:Data Mining in Health Insurance Data using Outlier Detection and Subgroup Discovery
09 Dong Nguyen (UT), Text as Social and Cultural Data: A Computational Perspective on Variation in Text
10 Robby van Delden (UT), (Steering) Interactive Play Behavior
11 Florian Kunneman (RUN), Modelling patterns of time and emotion in Twitter #anticipointment
12 Sander Leemans (TUE), Robust Process Mining with Guarantees
13 Gijs Huisman (UT), Social Touch Technology - Extending the reach of social touch through haptic technology
14 Shoshannah Tekofsky, You Are Who You Play You Are: Modelling Player Traits from Video Game Behavior
15 Peter Berck, Memory-Based Text Correction
16 Aleksandr Chuklin, Understanding and Modeling Users of Modern Search Engines
17 Daniel Dimov, Crowdsourced Online Dispute Resolution
18 Ridho Reinanda (UVA), Entity Associations for Search
19 Jeroen Vuurens (UT), Proximity of Terms, Texts and Semantic Vectors in Information Retrieval
20 Mohammadbashir Sedighi (TUD), Fostering Engagement in Knowledge Sharing: The Role of Perceived Benefits, Costs and Visibility
21 Jeroen Linssen (UT), Meta Matters in Interactive Storytelling and Serious Gaming (A Play on Worlds)
22 Sara Magliacane (VU), Logics for causal inference under uncertainty
23 David Graus (UVA), Entities of Interest — Discovery in Digital Traces
24 Chang Wang (TUD), Use of Affordances for Efficient Robot Learning
25 Veruska Zamborlini (VU), Knowledge Representation for Clinical Guidelines, with applications to Multimorbidity Analysis and Literature Search
26 Merel Jung (UT), Socially intelligent robots that understand and respond to human touch
27 Michiel Joosse (UT), Investigating Positioning and Gaze Behaviors of Social Robots: People's Preferences, Perceptions and Behaviors
28 John Klein (VU), Architecture Practices for Complex Contexts