Software Engineering Institute | Carnegie Mellon University

Distribution, Data, Deployment: Software Architecture Convergence in Big Data Systems

Ian Gorton

John Klein

May 2014

INTRODUCTION

The exponential growth of data in the last decade has fueled a new specialization for software technology, namely that of data-intensive, or big data, software systems [1]. Internet-born organizations such as Google and Amazon are at the cutting edge of this revolution, collecting, managing, storing, and analyzing some of the largest data repositories that have ever been constructed. Their pioneering efforts [2,3], along with those of numerous other big data innovators, have made a variety of open source and commercial data-management technologies available for any organization to exploit to construct and operate massively scalable, highly available data repositories.

Data-intensive systems have long been built on SQL database technology, which relies primarily on vertical scaling—faster processors and bigger disks—as workload or storage requirements increase. Inherent vertical-scaling limitations of SQL databases [4] have led to new products that relax many core tenets of relational databases. Strictly defined normalized data models, strong data consistency guarantees, and the SQL standard have been replaced by schema-less and intentionally denormalized data models, weak consistency, and proprietary APIs that expose the underlying data-management mechanisms to the programmer. These "NoSQL" products [4] 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 the cluster. Prominent examples of NoSQL databases include Cassandra, Riak, and MongoDB; the sidebar "NoSQL Databases" surveys these advances.

NoSQL Databases

The rise of big data applications has caused significant flux in database technologies. While mature, relational database technologies continue to evolve, a spectrum of databases labeled *NoSQL* has emerged in the past decade. The relational model imposes a strict schema, which inhibits data evolution and causes difficulties with scaling across clusters. In response, NoSQL databases have adopted simpler data models. Common features include schema-less records, allowing data models to evolve

Software Engineering Institute
Carnegie Mellon University
4500 Fifth Avenue
Pittsburgh, PA 15213-2612

Phone: 412-268-5800
Toll-free: 1-888-201-4479

www.sei.cmu.edu

dynamically, and horizontal scaling, by sharding and replicating data collections across large clusters. Figure S1 illustrates the four most prominent data models, and we summarize their characteristics below. More comprehensive information can be found at http://nosql-database.org.

- *Document databases* store collections of objects, typically encoded using JSON or XML. Documents have keys, and secondary indexes can be built on non-key fields. Document formats are self-describing, and a collection may include documents with different formats. Leading examples are MongoDB (http://www.mongodb.org) and CouchDB (http://couchdb.apache.org).
- *Key-value databases* implement a distributed hash map. Records can be accessed only through key searches, and the value associated with each key is treated as opaque, requiring reader interpretation. This simple model facilitates sharding and replication to create highly scalable and available systems. Examples are Riak (http://riak.basho.com) and DynamoDB (http://aws.amazon.com/dynamodb).
- *Column-oriented databases* extend the key-value model by organizing keyed records as a collection of columns, where a column is a key-value pair. The key becomes the column name, and the value can be an arbitrary data type such as a JSON document or a binary image. A collection may contain records that have different numbers of columns. Examples are HBase (http://hadoop.apache.org) and Cassandra (https://cassandra.apache.org).
- *Graph databases* organize data in a highly connected structure, typically some form of directed graph. They can provide exceptional performance for problems involving graph traversals and subgraph matching. As efficient graph partitioning is an NP-hard problem, these databases tend to be less concerned with horizontal scaling and commonly offer ACID transactions to provide strong consistency. Examples include Neo4j (http://www.neo4j.org) and GraphBase (http://graphbase.net).

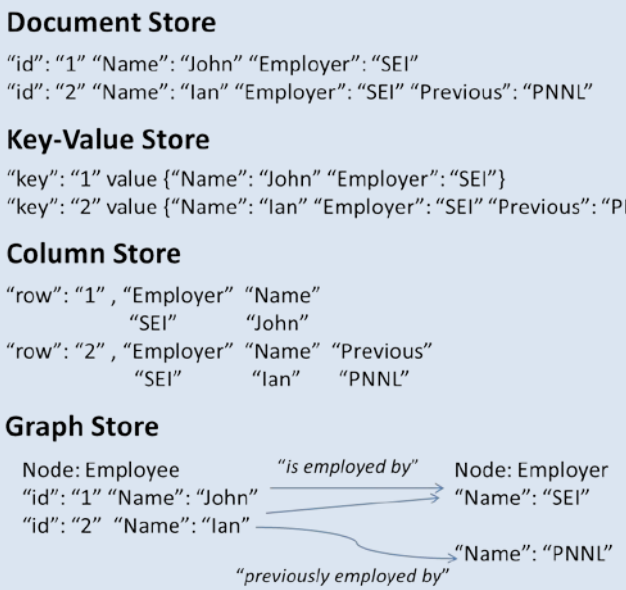

Figure S1: Examples of Major NoSQL Data Models

NoSQL technologies have many implications for application design. As there is no equivalent of SQL, each technology supports its own specific query mechanism. These typically make the application programmer responsible for explicitly formulating query executions, rather than relying on query planners that execute queries based on declarative specifications. The ability to combine results from different data collections also becomes the programmer's responsibility. This lack of ability to perform JOINs forces extensive denormalization of data models so that JOIN-style queries can be efficiently executed by accessing a single data collection. When databases are sharded and replicated, it further becomes the programmer's responsibility to manage consistency when concurrent updates occur and to design applications that tolerate stale data due to latency in update replication.

Distributed databases have fundamental quality constraints, defined by Brewer's CAP Theorem [5]. 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). A practical interpretation of this theorem is provided by Abadi's PACELC [6], which states that if there is a partition (P), a system must trade availability (A) against consistency (C); else (E) in the usual case of no partition, a system must trade latency (L) against consistency (C).

Additional design challenges for scalable data-intensive systems stem from the following issues:

- Achieving high levels of scalability and availability leads to highly distributed systems. Distribution occurs in all tiers, from web server farms and caches to back-end storage.
- The abstraction of a single-system image, with transactional writes and consistent reads using SQL-like query languages, is difficult to achieve at scale [7]. Applications must be aware of data replicas, handle inconsistencies from conflicting replica updates, and continue operation in spite of inevitable failures of processors, networks, and software.
- Each NoSQL product embodies a specific set of quality attribute tradeoffs, especially in terms of their performance, scalability, durability, and consistency. Architects must diligently evaluate candidate database technologies and select databases that can satisfy application requirements. This often leads to polyglot persistence [8], using different database technologies to store different data sets within a single system, in order to meet quality attribute requirements.

As data volumes grow to petascale and beyond, the required hardware resources may grow to thousands of servers.

Further, as data volumes grow to petascale and beyond, the hardware resources required grow from hundreds up to tens of thousands servers. At this deployment scale, many widely used software architecture patterns are unsuitable, and overall costs can be significantly reduced by architectural and algorithmic approaches

that are sensitive to hardware resource utilization. These issues are explained in the sidebar "Why Scale Matters."

Addressing these challenges requires making careful design tradeoffs that span distributed software, data, and deployment architectures, and demands extensions to traditional software architecture design knowledge to account for this tight coupling in scalable systems. Scale drives a *consolidation of concerns*, so that software, data, and deployment architectural qualities can no longer be effectively considered separately. In this paper, we draw from our current work in healthcare informatics to illustrate this.

Why Scale Matters

Scale has many implications for software architecture, and we describe two of them here. The first focuses on how scale changes the problem space of our designs. The second is based on economics: At very large scales, small optimizations in resource usage can lead to very large cost reductions in absolute terms.

Designing for scale: Big data systems are inherently distributed systems, and their architectures must explicitly handle partial failures, communications latencies, concurrency, consistency, and replication. As systems grow to utilize thousands of processing nodes and disks, and become geographically distributed, these issues are exacerbated as the probability of a hardware failure increases. One study found that 8% of servers in a typical data center experience a hardware problem annually, with disk failure most common [1]. Applications must also deal with unpredictable communication latencies and network connection failures. Scalable applications must treat failures as common events that are handled gracefully to ensure that operation is not interrupted.

To address these requirements, resilient architectures must

- replicate data to ensure availability in the case of disk failure or network partition. Replicas must be kept strictly or eventually consistent, using either master-slave or multi-master protocols. The latter needs mechanisms such as Lamport clocks [2] to resolve inconsistencies due to concurrent writes.
- design components to be stateless, replicated, and tolerant of failures of dependent services. For example, by using the Circuit Breaker pattern [3] and returning cached or default results whenever failures are detected, the architecture limits failures and allows time for recovery.

Economics at scale: Big data systems can use many thousands of servers and disks. Whether these are capital purchases or rented from a service provider, they remain a major cost and hence a target for reduction. Elasticity is a tactic to reduce resource usage by dynamically deploying new servers as the load increases and releasing them as the load decreases. This requires servers that boot and initialize quickly and application-specific strategies to avoid premature resource release.

Other strategies target the development tool chain to maintain developer productivity while decreasing resource utilization. For example, Facebook built HipHop, a PHP-

to-C++ transformation engine [4] that reduced the CPU load for serving web pages by 50%. At the scale of Facebook's deployment, this creates significant operational cost saving. Other targets for reduction are software license costs, which can be prohibitive at scale. This has led some organizations to create custom database and middleware technologies, many of which have been released as open source. Leading examples of technologies for big data systems are from Netflix (http://netflix.github.io) and LinkedIn (http://linkedin.github.io).

Other implications of scale for architecture include testing and fault diagnosis. Due to the deployment footprint of these systems and the massive data sets that they manage, it can be impossible to comprehensively validate code before deployment to production. Canary testing and "simian armies" are examples of the state of the art in testing at scale [5]. When problems occur in production, advanced monitoring and logging are needed for rapid diagnosis. In large-scale systems, log collection and analysis quickly become big data problems. Solutions must include a low overhead, scalable logging infrastructure such as Blitz4J [6].

References

1. K.V. Vishwanath and N. Nagappan, "Characterizing Cloud Computing Hardware Reliability," *Proc. 1st ACM Symp. Cloud Computing* (SoCC 10), 2010, pp. 193–204.
2. L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Comm. ACM*, vol. 21, no. 7, 1978, pp. 558–565.
3. M.T. Nygard, *Release It! Design and Deploy Production-Ready Software*, Pragmatic Bookshelf, 2007.
4. H. Zhao, "HipHop for PHP: Move Fast," Feb. 2010. https://developers.facebook.com/blog/post/2010/02/02/hiphop-for-php--move-fast
5. B. Schmaus, "Deploying the Netflix API," Aug. 2013. http://techblog.netflix.com/2013/08/deploying-netflix-api.html
6. K. Ranganathan, "Announcing Blitz4j: A Scalable Logging Framework," Nov. 2012. http://techblog.netflix.com/search/label/appender

CHARACTERISTICS OF BIG DATA APPLICATIONS

Big data applications are rapidly becoming pervasive across a wide range of business domains. Two examples for which big data–driven analytics loom prominently on the horizon are the airline and healthcare industries:

1. Modern commercial airliners produce approximately 0.5 TB of operational data per flight [9]. This data can be used to diagnose faults in real time, optimize fuel consumption, and predict maintenance needs. Airlines must build scalable systems to capture, manage, and analyze this data to improve reliability and reduce costs.
2. Across the healthcare field, big data analytics could save an estimated $450 billion in the United States [10]. 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

Requirements for big data systems represent a significant departure from traditional business systems.

and prevention, and reduce costs by improving patient outcomes and operational efficiencies.

Across these and many other domains, big data systems share common requirements that drive the design of suitable software solutions. Collectively, these requirements represent a significant departure from traditional business systems, which are relatively well constrained in terms of data growth, analytics, and scale. Requirements common to big data systems include

- **write-heavy workloads:** From social media sites to high-resolution sensor data collection in the power grid, big data systems must be able to sustain write-heavy workloads [1]. As writes are more costly than reads, data partitioning and distribution (sharding) can be used to spread write operations across disks, and replication can be used to provide high availability. Sharding and replication introduce availability and consistency issues that the system must address.
- **variable request loads:** Business and government systems experience highly variable workloads for reasons that include product promotions, emergencies, and statutory deadlines such as tax submissions. To avoid the costs of over-provisioning to handle these occasional spikes, cloud platforms are *elastic*, allowing an application to add processing capacity when needed and release resources when load drops. Effectively exploiting this deployment mechanism requires an architecture that has application-specific strategies to detect increased load, rapidly add new resources to share load, and release resources as load decreases.
- **computation-intensive analytics:** Most big data systems must support diverse query workloads, mixing requests that require rapid responses with long-running requests that perform complex analytics on significant portions of the data collection. This leads to software and data architectures that are explicitly structured to meet these varying latency demands. Netflix's Recommendations Engine [11] is a pioneering example of how software and data architectures can be designed to partition simultaneously between handling low-latency requests and performing advanced analytics on large data collections to continually enhance the quality of personalized recommendations.
- **high availability:** With hundreds of nodes in a horizontally scaled deployment, there are inevitable hardware and network failures. Distributed software and data architectures must be designed to be resilient. Common approaches for high availability include replicating data across geographical regions [12], stateless services, and application-specific mechanisms to provide degraded service in the face of failures.

The solutions to these requirements crosscut the distributed software, data model, and physical deployment architectures. For example, elasticity requires processing capacity that can be acquired from the execution platform on demand, policies and mechanisms to appropriately start and stop services as application load varies, and a database architecture that can reliably satisfy queries under increased load. This coupling of architectures to satisfy a particular quality attribute is commonplace in big data applications and can be regarded as a tight coupling of the process, logical, and physical views in the 4+1 View Model [13].

EXAMPLE: CONSOLIDATION OF CONCERNS

At the Software Engineering Institute, we are evolving a healthcare informatics system to aggregate data from multiple petascale medical-record databases for clinical applications. To attain high scalability and availability at low cost, we are investigating the use of NoSQL databases for this data aggregation. The design uses geographically distributed data centers to increase availability and reduce latency for users distributed globally.

Consider the consistency requirements for two categories of data in this system: patient demographics (e.g., name, insurance provider) and diagnostic test results (e.g., results of blood or imaging tests). Patient demographic records are updated infrequently. These updates must be immediately visible at the local site where the data was modified ("read your writes"), but a delay is acceptable before the update is visible at other sites ("eventual consistency"). In contrast, diagnostic test results are updated more frequently, and changes must be immediately visible everywhere to support telemedicine and remote consultations with specialists ("strong consistency").

We are prototyping solutions using several NoSQL databases. We focus here on one prototype using MongoDB to illustrate the architecture drivers and design decisions. The design segments data across three shards and replicates data across two data centers (Figure 1).

MongoDB enforces a master-slave architecture, in which every data collection has a master replica that serves all write requests and propagates changes to other replicas. Clients may read from any replica, opening an inconsistency window between writes to the master and reads from other replicas.

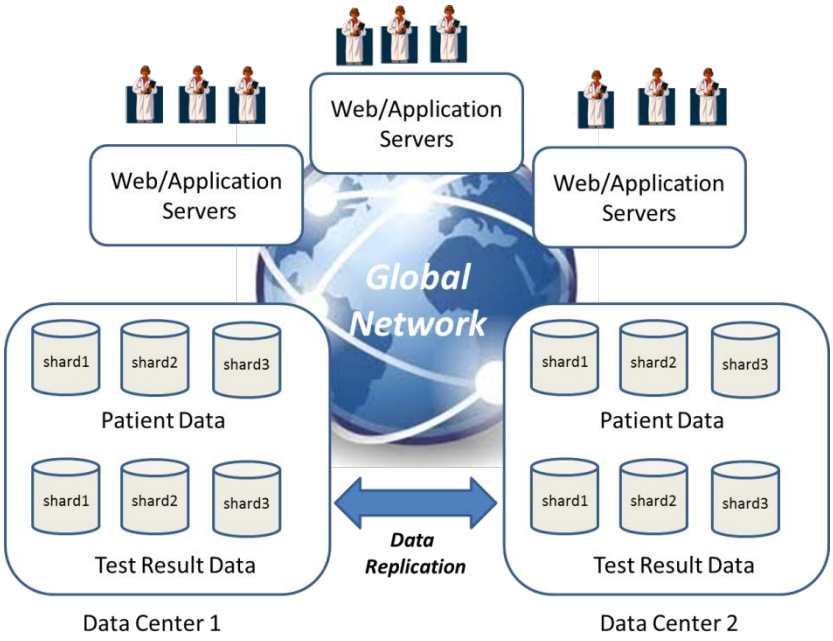

Figure 1: MongoDB-Based Healthcare Data-Management Solution

MongoDB allows tradeoffs between consistency and latency through parameter options on each write and read. A write can be unacknowledged (no assurance of durability, low latency), durable on the master replica, or durable on the master and one or more replicas (consistent, high latency). A read can prefer the closest replica (low latency, potentially inconsistent), be restricted to the master replica (consistent, partition intolerant), or require a majority of replicas to agree on the data value to be read (consistent, partition tolerant).

The application developer must choose appropriate write and read options to achieve the desired performance, consistency, and durability and must handle partition errors to achieve the desired availability. In our example, patient-demographic data writes must be durable on the primary replica, but reads may be directed to the closest replica for low latency, making patient-demographic reads insensitive to network partitions at the cost of potentially inconsistent responses.

In contrast, writes for diagnostic test results must be durable on all replicas. Reads can be performed from the closest replica since the write ensures that all replicas are consistent. This means that writes must handle failures caused by network partitions, while read operations are insensitive to partitions.

Today, our healthcare informatics application runs atop a SQL database, which hides the physical data model and deployment topology from developers. SQL databases provide a *single-system image* abstraction, which separates concerns

between the application and database by hiding the details of data distribution across processors, storage, and networks behind a transactional read/write interface [14]. In shifting to a NoSQL environment, an application must directly handle the faults that will depend on the physical data distribution (sharding and replication) and the number of replica sites and servers. These low-level infrastructure concerns, traditionally hidden under the database interface, must now be explicitly handled in application logic.

Low-level infrastructure concerns, traditionally hidden under the database interface, must be explicitly handled in big data systems.

SYSTEMATIC DESIGN USING TACTICS

In designing an architecture to satisfy quality drivers like those discussed in this healthcare example, one proven approach is to systematically select and apply a sequence of architecture tactics [15]. Tactics are elemental design decisions that embody architecture knowledge of how to satisfy one design concern of a quality attribute. Tactic catalogs enable reuse of this knowledge. However, existing catalogs do not contain tactics specific to big data systems. In Figures 2 and 3, we extend the basic tactics [15] for performance and availability specifically for big data systems, and in Figure 4 we define tactics for scalability, focused on the design concern of increased workload. Each figure shows how the design decisions span data, distribution, and deployment architectures.

For example, achieving availability requires masking faults that inevitably occur in a distributed system. At the data level, replicating data items is an essential step to handle network partitions. When an application cannot access any database partition, another tactic to enhance availability is to design a data model that can return meaningful default values without accessing the data. At the distributed software layer, caching is a tactic to achieve the “default values” tactic defined in the data model. At the deployment layer, an availability tactic is to geographically replicate the data and distributed application software layers to protect against power and network outages. Each of these tactics is required to handle the different types of faults that threaten availability, and their combined representation in Figure 3 provides an architect with comprehensive guidance to achieve a highly available system.

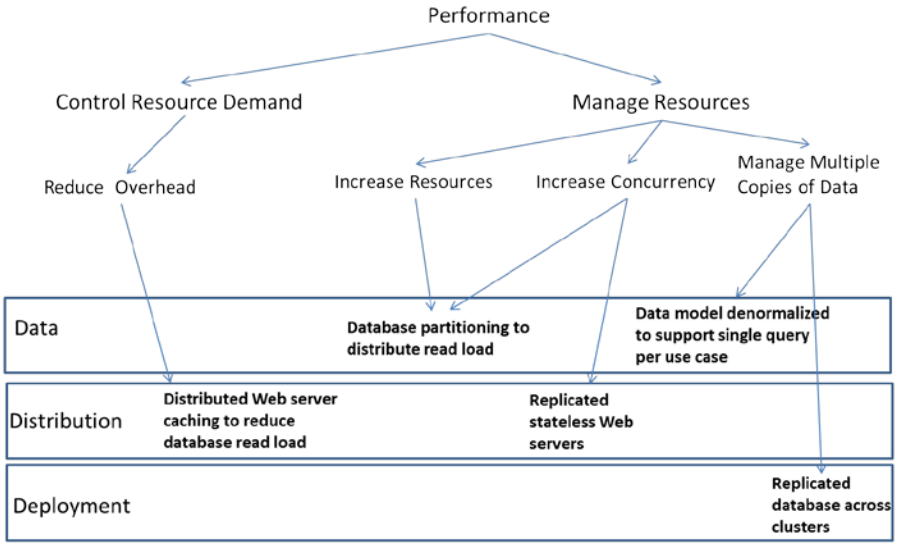

Figure 2: Performance Tactics

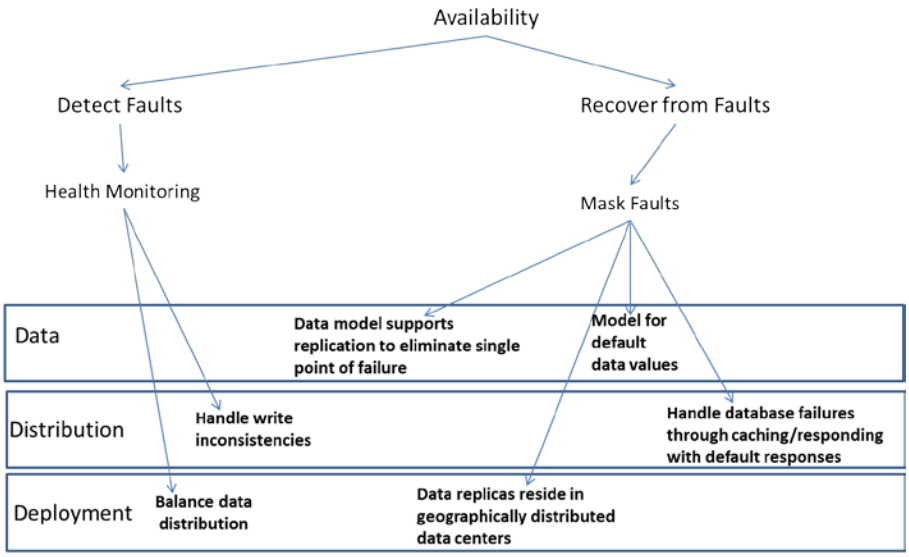

Figure 3: Availability Tactics

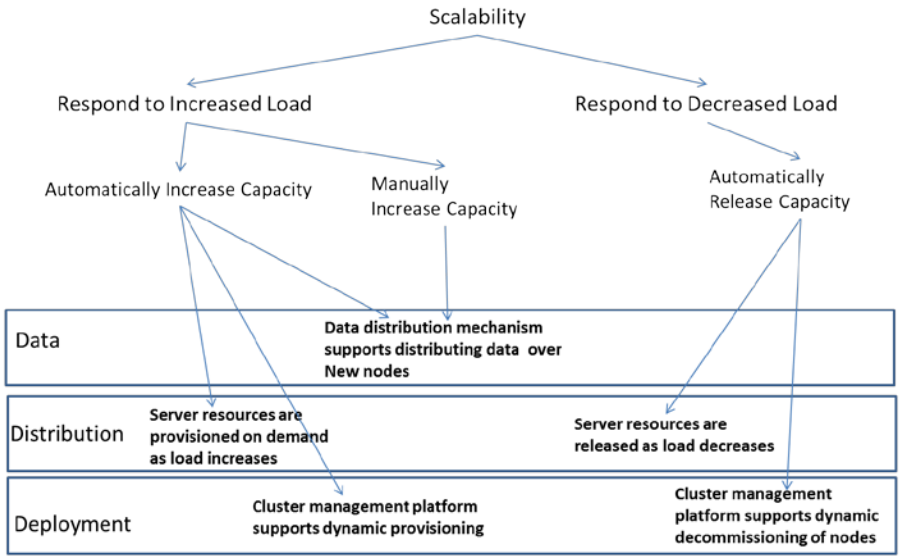

Figure 4: Scalability Tactics

ARCHITECTURE CONVERGENCE IMPLICATIONS

Big data applications are pushing the limits of software engineering on multiple horizons. Successful solutions span the design of the data, distribution, and deployment architectures. It is essential that the body of software architecture knowledge evolves to capture this advanced design knowledge for big data systems.

This paper is a first step on this path. Our work is proceeding in two complementary directions. First, we are expanding the collection of architecture tactics presented in this paper and encoding these in an environment that supports navigation between quality attributes and tactics, making crosscutting concerns for design choices explicit. We are also linking tactics to design solutions based on specific big data technologies, enabling architects to rapidly relate the capabilities of a particular technology to a specific set of architecture tactics.

AUTHORS

Ian Gorton is a Senior Member of the Technical Staff in Architecture Practices at the Carnegie Mellon Software Engineering Institute, where he is investigating issues related to software architecture at scale. This includes designing large-scale data-management and analytics systems, and understanding the inherent connections and tensions between software, data, and deployment architectures. He has a PhD from Sheffield Hallam University and is a Senior Member of the IEEE Computer Society.

John Klein is a Senior Member of the Technical Staff at the Carnegie Mellon Software Engineering Institute, where he does consulting and research on scalable software systems as a member of the Architecture Practices team. He received a BE from Stevens Institute of Technology and an ME from Northeastern University. He serves as Secretary of the IFIP Working Group 2.10 on Software Architecture, is a member of the IEEE Computer Society, and is a Senior Member of the ACM.

REFERENCES

1. D. Agrawal, S. Das, and A. El Abbadi. “Big Data and Cloud Computing: Current State and Future Opportunities,” *Proc. 14th Int’l Conf. Extending Database Technology* (EDBT/ICDT 11), ACM, 2011, pp. 530–533.
2. W. Vogels, “Amazon DynamoDB: A Fast and Scalable NoSQL Database Service Designed for Internet Scale Applications,” Jan. 2012. http://www.allthingsdistributed.com/2012/01/amazon-dynamodb.html
3. F. Chang, J. Dean, S. Ghemawat, et al., “Bigtable: A Distributed Storage System for Structured Data,” *ACM Trans. Comput. Syst.*, vol. 26, no. 2, 2008.
4. P.J. Sadalage and M. Fowler, *NoSQL Distilled*, Addison-Wesley Professional, 2012.
5. E. Brewer, “CAP Twelve Years Later: How the ‘Rules’ Have Changed,” *Computer*, vol. 45, no. 2, 2012, pp. 23–29.
6. D.J. Abadi, “Consistency Tradeoffs in Modern Distributed Database System Design: CAP Is Only Part of the Story,” *Computer*, vol. 45, no. 2, 2012, pp. 37–42.
7. J. Shute, R. Vingralek, B. Samwel, et al., “F1: A Distributed SQL Database That Scales,” *Proc. VLDB Endowment*, vol. 6, no. 11, 2013, pp. 1068–1079.
8. M. Fowler, “Polyglot Persistence,” Nov. 2011. http://www.martinfowler.com/bliki/PolyglotPersistence.html

9. M. Finnegan, “Boeing 787s to Create Half a Terabyte of Data per Flight,” *ComputerWorld UK*, Mar. 2013. http://www.computerworlduk.com/news/infrastructure/3433595/boeing-787s-to-create-half-a-terabyte-of-data-per-flight-says-virgin-atlantic
10. P. Groves, B. Kayyali, D. Knott, et al., *The ‘Big Data’ Revolution in Healthcare*, McKinsey & Company, 2013. http://www.mckinsey.com/insights/health_systems_and_services/the_big-data_revolution_in_us_health_care
11. X. Amatriain and J. Basilico, “System Architectures for Personalization and Recommendation,” Netflix Techblog, Mar. 2013. http://techblog.netflix.com/2013/03/system-architectures-for.html
12. M. Armbrust, “A View of Cloud Computing,” *Comm. ACM*, vol. 53, no. 4, 2010, pp. 50–58.
13. P.B. Kruchten, “The 4+1 View Model of Architecture,” *IEEE Software*, vol. 12, no. 6, 1995, pp. 42–50.
14. J. Gray and A. Reuter. *Transaction Processing*, Kaufmann, 1993.
15. L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice* (3rd ed.), Addison-Wesley, 2012.

This article will appear in an upcoming issue of *IEEE Software*.

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

DM-0000810