2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP) | 979-8-3503-0037-6/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICSE-SEIP58684.2023.00011

A Model-based, Quality Attribute-guided Architecture Re-Design Process at Google

Qin Jia
Google LLC
qinjia@google.com

Yuanfang Cai
Drexel University
yuanfang.cai@drexel.edu

Onur C. Çakmak
Google LLC
onurcc@google.com

***Abstract*—Communicating and justifying design decisions are difficult, especially when the architecture design has to evolve. In this paper, we report our experiences of using formal but lightweight design models to communicate, justify, and analyze the quality trade-offs of an architecture revision plan for Monarch, a large-scale legacy system from Google. We started from a few critical user scenarios and their associated quality attribute scenarios, which makes these models lightweight and concise, expressing high-level abstractions only. We also separated static views from dynamic views so that each diagram can be precise and suitable for analyzing different types of quality attributes respectively. The combination of scenarios, quality attributes, and lightweight modeling was well accepted by the team as an effective way to analyze and communicate the trade-offs. A few days after we presented and shared this process, two new projects within the Monarch team adopted component and sequence diagrams in their design documents, and two other product areas within Google started to learn and to adopt the process as well. Our experience indicates that these architecture modeling and analysis techniques can be integrated into software development process to communicate and assess features, quality attributes, or design decisions continuously and iteratively.**

***Index Terms*—software architecture, software modeling, quality attribute**

I. Introduction

Software architecture modeling techniques, such as Unified Modeling Languages (UML) [1], quality attribute (QA) analysis, trade-off analysis [2], have been researched and taught in classrooms for decades. In practice, however, these techniques are seldom applied, as reported in recent surveys [3]–[6]. Most interviewed developers do not use formal models in practice, and consider that creating a "big picture view" of a large-scale system is difficult, costly, and not feasible [3], [4]. While it has been recognized that software projects could benefit from formal models if they can be applied within a limited context [3], [6], there is no guidance on how to leverage these models, especially when architecture changes are proposed and needed to be analyzed and justified for large-scale legacy systems. This is exactly the challenges faced by Monarch [7]—a large-scale in-memory database and one of the largest infrastructure software systems within Google.

To accommodate the rapid growth of users and workloads, the tech leads (TLs) of Monarch proposed to redesign the system to adopt a more modularized architecture. Given the potentially significant impact, the TLs encountered the challenges of convincing the team that the new design was worthwhile: due to the increased number of servers, the team was concerned about the potential degradation of several key qualities, especially performance, but could not precisely estimate the severity of the impact and the trade-offs among maintainability, performance, availability, etc..

In this paper, we report our experiences of *combining multiple lightweight models, which are abstract enough to model the specific scenarios only, and using these models to efficiently analyze and communicate trade-offs among multiple quality attributes* for the proposed re-architecting plan of Monarch in a rigorous and precise way. We first identified a few critical user scenarios and associated quality attributes that will be affected by the proposed re-design of the architecture, such as latency and availability. We further modeled each attribute using a *quality attribute scenario* (QAS), in which the concrete components (binaries) involved, responses expected, and the target measures of these responses were specified. After that, we modeled the static structure among these components using UML component diagrams, and modeled their runtime behaviors using UML sequence diagrams so that the *Remote Procedure Call* (RPC) routes needed to accomplish these scenarios can be visualized.

We modeled the current architecture and proposed new architecture using both component and sequence diagrams respectively. These models made it clear that in order to realize the same user scenario, how many new components will be added/changed, how the RPCs route through these components in the current design, and how the routes will change in the proposed new design. The paths of the RPCs in these sequence diagrams provided the concrete visualization for the team to assess how latency and availability will be affected in the proposed new design. The component diagrams made it clear how existing components will be decoupled to simplify the APIs and improve maintainability.

After quantifying the target measures in QAS, and modeling RPC routes in both designs, we were able to tell exactly how many more components and binary instances will be added on the RPC paths in the new design. Comparing with the actual performance data, the team recognized that even though the new design will increase query latency to some extent, the system can still meet performance service level objectives (SLO) [8]. On the other hand, the new design will greatly improve availability and maintainability, which are currently at risk and hindering the evolution of the overall system.

2832-7659/23/$31.00 ©2023 IEEE
DOI 10.1109/ICSE-SEIP58684.2023.00011

The team highly commended the scenario-based lightweight modeling and trade-off analysis techniques. The comments we received indicated that the UML component and sequence diagrams are effective ways to illustrate static and run-time structures of the system. The process was recommended to other Monarch projects and other product teams within Google. A few days after we presented and shared this process, two new projects within the Monarch team adopted component and sequence diagrams in their design documents, and two other product areas within Google started to learn and tried to adopt the modeling process. To summarize, there are two critical steps ensuring the success of this process:

First, *specifying concrete scenarios makes models concise*. Since a QAS lists the specific components involved in these scenarios, we were able to create lightweight models with these components only. It is critical that we limited the models to high-level abstractions, expressing architecture use cases only, to avoid unnecessary complexity of modeling the overall design [3], [4], [6].

Second, *separating static and run-time views makes models precise*. It has been long recognized that software architecture always has multiple structures [2], [9]: static module structures, run-time component-connector structures, and allocation structures. No single diagram can precisely capture all the structures. In practice, however, commonly used box-and-line style diagrams usually do not distinguish static or run-time views, and ambiguity is inevitable.

These two steps made it possible for us to create precise and self-explanatory diagrams, so that the whole team, including senior and junior developers, as well as managers, could understand the existing and newly proposed design without extensive prior knowledge on software modeling. This inspiring experience indicates that it is possible to integrate partial, lightweight modeling, quality attribute analysis, and trade-off analysis into software development processes, rather than creating a complex full picture at the beginning of a project. Since this process starts from user scenarios, it can be applied whenever a new feature is added, or a quality attribute needs to be evaluated.

According to Google's policy, we cannot report the actual measures from Monarch. Although the concrete QA numbers reported in this paper are for illustration purposes, their relative values are based on the actual data and estimation.

II. Challenges to Monarch

A. Background

Monarch [7], one of most widely used infrastructure software systems in Google, is used to monitor the availability, correctness, performance, load, and other aspects of applications and systems. For more than a decade, almost all Google global user-facing services (e.g., Youtube, GMail and Google Maps) and backend infrastructure for such services, including Colossus [10], Spanner [11], F1 [12] and Borg [13], have been replying on the reliable monitoring service provided by Monarch. Every second, the system accepts terabytes of time series data into memory and serves millions of queries. To ensure performance and availability, Monarch has a regionalized architecture. Global query and configuration planes integrate the regions into a unified system.

Each Monarch zone consists of a collection of clusters located in a strongly network-connected region. A zone stores local time series data based on geographical locations, and responses to the queries on those data. The locality ensures the high availability and low latency of zonal writes and queries. Component instances are deployed and replicated across the clusters to improve reliability. Each node within a cluster stores data in memory and avoids dependencies among each other, so that each zone can continuously provide services during transient outage of other zones, global components, or storage systems. Monarch's global components are replicated in each geographical zone, and can interact with all other zones. In order to communicate and analyze the architecture of Monarch, the team created an informal model as depicted in Figure 1, which shows the components involved in queries.

In response to a query, Monarch reads the time series data stored in various zones, and conducts various computations (e.g., aligns, aggregates and joins) on these data. For example, suppose a user issues a query to get the global queries-per-second (QPS) data for their services. Monarch reads the RPC count for each task (i.e., monitored *target*) of the service in different zones and aggregates them into a global QPS.

Once a Monarch user (e.g., a Google engineer, a dashboard, or an alerting pipeline) issues the query, the process starts from the *Root Mixer*, the entry point of the scenario. It first interacts with a *Root Index Server* to get the zones that potentially contain the requested data. After that, the *Root Mixer* sends the query to *Zone Mixers* in the selected zones. Each *Zone Mixer* interacts with a *Zone Index Server* to get a set of relevant *Leaves*, which potentially contain data needed for the query. A *Leaf* stores the time series data in memory, indexes, processes, and computes these data as well. The query processing within each zone contains two passes:

1) Replica Resolution: a *Zone Mixer* interacts with the relevant *Leaves* to get a summary of each relevant target to choose *Leaves* with the best data quality to process among *Leaf* replicas.

2) Query: these selected *Leaves* compute (e.g., align, aggregate and join) the time series data as much as they can and then send the output back to the *Zone Mixers* for further processing. The *Zone Mixers*, in turn, send the processed data back to the *Root Mixer*, which conducts final computation and aggregation, and returns to the end users.

In the meantime, the *Leaves* are also responsible for receiving new data points and storing them, as well as reading, processing, and writing time series data into a long-term historical data *Repository*. When a *Leaf* receives new time series data, or the time series data are moved between *Leaves* (for load-balancing), it also sends the indexing data (i.e., indices of the metadata of the time series, such as metric names and monitored targets) to the *Zone Index Server* to make sure

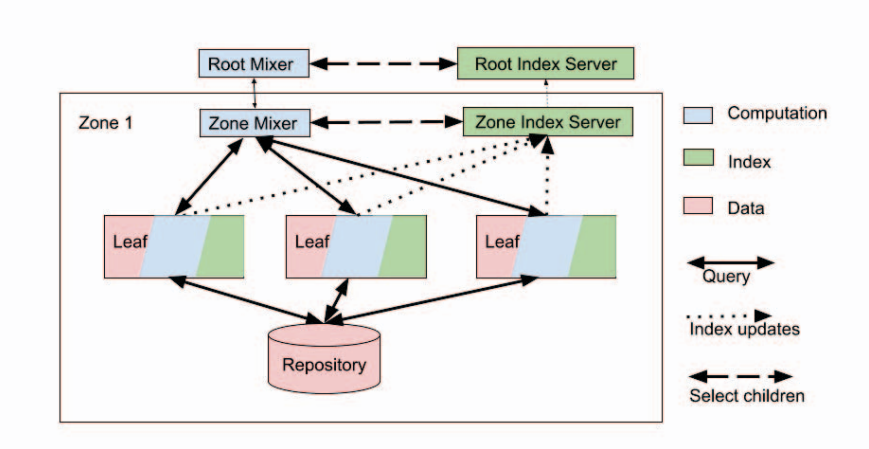


Fig. 1: Box-and-line model of the current architecture

it has the latest information about what data is stored within *Leaves*. The *Zone Index servers* then send the indexing data back to the *Root Index Servers*.

B. Challenges of rapid growth

During the past decade, Monarch experienced a rapid growth: the number and size of the stored time series data, as well as the queries-per-second (QPS), both experienced more than 2 times year-to-year growth. The number of *Leaves* within a zone increased from hundreds to tens of thousands over the years, and a query needs to retrieve a lot more data from more *Leaves*. The rapid growth greatly challenges the availability and scalability of Monarch queries. In addition, since a *Leaf* takes all the responsibilities of time series processing, indexing, and storage, failures on any of these responsibilities cannot be isolated, nor can the code and resources be decoupled. Therefore the reliability and maintainability become challenging. The developers complain that it is getting harder and harder to make changes to the *Leaf* component because multiple responsibilities are coupled with each other. We scanned the Monarch code base, and detected many anti-patterns [14] within the *Leaf* component. The system needs to be re-architected, in particular, to decouple these responsibilities, and, most importantly, to increase availability and maintainabiliy to keep up with the rapid growth.

C. Proposal to handle the challenges

In order to handle the challenges of availability and maintainability caused by the rapid growth, the team has proposed to decouple the overloaded *Leaf* component into multiple components, each responsible for storing, computing and indexing the data respectively. The tech leads (TLs) have created a box-and-line style diagram to represent the newly proposed architecture, as depicted in Figure 2.

D. Problems with informal modeling

The informal models depicted in Figure 1 and Figure 2 are then used to communicate and analyze the architecture revision proposal but presented a few communication obstacles:

First, the bidirectional arrows linking the components are ambiguous. For example, these is a solid line linking *Root mixer* and *Zone Mixer*, indicating that there is a query relation between them. But it's not clear which component starts the interaction, and which one responds next. The two separate passes, Replica Resolution and Query, can not be modeled either. The sequences of RPCs are critical for performance and latency analysis, but cannot be modeled in these diagrams.

Second, the semantic of each box is not clear: does a *Leaf* box represent a static *Leaf* component, or an instance of the same component deployed in multiple servers?

Third, the diagram of the proposed new architecture (Figure 2) appears to be more complicated than the one modeling the current architecture (Figure 1): two new components were added outside of the *Leaf* components, and many boxes were created within a *Leaf*. It is unclear if each box represents a new binary, or a module within the same binary.

Given that these diagrams are not sufficient to analyze trade-offs, there were concerns among team members that the new architecture would negatively impact performance and latency. In fact, it was not clear what the trade-offs among availability, maintainability, and latency were between the existing design and proposed new design. Without a more systematic way to conduct relatively rigorous and quantitative analysis, the team was not able to make this significant decision based on intuition and experiences only.

At this point, we recognized that one problem with the informal diagrams in Figure 1 and Figure 2 is that they mix static structures among these components and the run-time RPC passing sequences, where the canonical UML component and sequence diagrams could be used. In addition, these different dimensions of quality attributes can be modeled and analyzed rigorously using quality attributes scenarios [2].

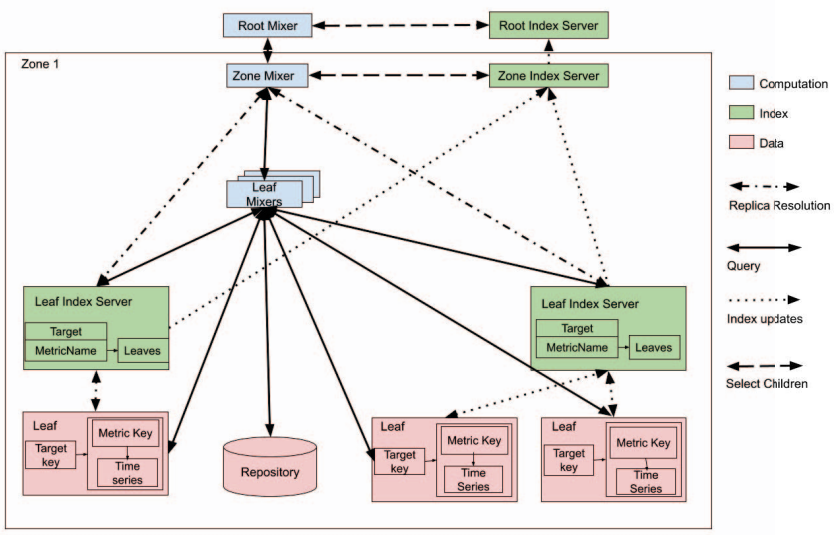


Fig. 2: Box-and-line model of the proposed architecture

In order to avoid creating overly complicated diagrams, we decided to start from just a few critical user scenarios. Next we elaborate each step of the process.

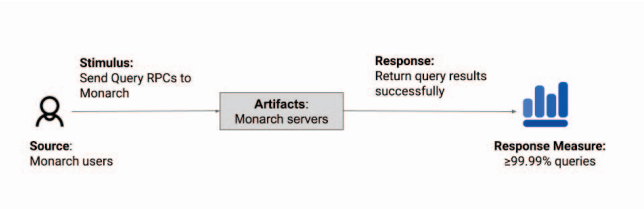


Fig. 3: A quality attribute scenario: Availability

III. Critical User Scenarios and Quality Attribute Scenarios

Monarch is a complicated system with dozens of components. A complete architecture model, either a static component diagram or a run-time sequence diagram, will be overly complicated. In order to keep the models concise, we decided to only model a part of the system related to the following critical user scenario:

QueryTimeSeries: A user sends a query to Monarch, and the system reads the time series data from various locations, computes them, and returns to the user successfully with low latency.

This user scenario combines both *functional requirements*, stating what the system must do and how it must behave, and *quality attribute* requirements that qualify these functions. Terminology such as *latency*, *availability* is widely used in industry and used to be called *non-functional* requirements. Bass et al. [2] pointed out that since *non-functional* requirements usually need to be accomplished through various *functions*, e.g., using login functions to ensure security, they should be called *quality attributes* to avoid the confusion.

Most functional requirements are not meaningful without qualifying attributes. Take the *QueryTimeSeries* scenario for example. If the system returns query results successfully, but only responds to the user after a few days, this functional requirement should be considered as failed. To specify qualities such as *"low latency"*, the Monarch team defines a number of *Service Level Objectives* (SLO) [8] that specify the detailed measures of query responses. For example, the availability SLO for *QueryTimeSeries* is defined as "*The system returns correct responses for* $\geq 99.99\%$ *of the queries.*"

Using these SLOs definitions, we can define a number of *Quality Attribute Scenarios* (QAS) [2] associated with the user scenario. As illustrated in Figure 3, a QAS consists of at least the following 5 parts[1]:

- *Source of Stimulus*: an entity, e.g., a Monarch user—a Google engineer, a dashboard, or an alerting pipeline, that initiated the stimulus.
- *Stimulus*: a condition or an event that requires a response when it arrives at a system, e.g., a query is initiated.
- *Artifacts*: the artifacts that are stimulated, which would be a collection of systems, the whole system, or some components of the system.
- *Response*: the activity undertaken as the result of the arrival of the stimulus.
- *Response measure*: when the response occurs, it should be measurable or testable.

A QAS starts from a concrete scenario, and allows the designer to specify (1) the artifacts involved in responding to an event, and (2) the quantifiable responses of each event. For example, the artifacts involved in the *QueryTimeSeries*

[1]All the concrete measures in this table are for illustration purposes, not actual Monarch metrics.

TABLE I: Target response measures of quality attributes

Quality Attributes	Target Measures
Availability	$\geq 99.99\%$ Monarch queries return successfully.
Maintainability	It takes less than X days to rollout a query change. Queries experience less than Y incidents a month. It takes less than Z hours to root cause an incident.
Latency	$\geq 99\%$ Monarch queries complete within M seconds
Resource efficiency	Less than 10% additional CPU and memory consumption while running a query

Note: All specific numbers in the table are for illustration purposes.

scenario in the current design are depicted in Figure 1, and its availability QAS is depicted in Figure 3: once the user initiated a query, the system should successfully return the results $\geq 99.99\%$ of the time.

For the Monarch *QueryTimeSeries* scenario, the TLs identified 4 quality attributes associated with it: 1) Availability, 2) Maintainability, 3) Latency, and 4) Resource efficiency. All these QASs share the same source, stimulus, and artifacts, and only differ in terms of response measures, as summarized in Table I. In order to analyze these quality attributes, we further modeled the static and run-time structures of the artifacts involved in these QAS, and analyzed how they will change in the proposed new architecture.

IV. Model Current Architecture

Figure 1 depicts the informal diagram the TLs created to model the current design, including all the components involved in the *QueryTimeSeries* scenario. The diagram has 8 boxes, but only 6 components are involved in the scenario: (1) *Root Mixer*, (2) *Root Index Server*, (3) *Zone Mixer*, (4) *Zone Index Server*, (5) *Leaf*, and (6) *Repository*. Both Figure 1 and 2 are ambiguous in that they mix static components (binaries) with binary instances deployed on multiple servers: these 3-color *Leaf* boxes meant to model *multiple* instances deployed on multiple servers, not that they are three components, or there are only three instances.

To address these problems, we created a UML component diagram, as shown in Figure 4, to model the current design. The key difference is that Figure 4 uses `Provided Interface` and `Required Interface` to model the APIs provided and expected by a component, rather than attempting to model how RPCs are passed among them. It only contains the component-level static information, i.e., there is only one box for one component, no matter how many instances are deployed in the production.

After identifying the components involved in each response, it becomes straightforward to model how RPCs route through these components to accomplish the scenario. Figure 5 depicts the sequence diagrams modeling the two stages of the *QueryTimeSeries* scenario, i.e., Replica Resolution and Query. Sequence diagram is an effective tool for latency and availability analysis because it visualizes which and how many hops there are on the RPC path of each scenario. Figure 5 indicates that, in the current design, there are 12 to 14 RPC hops needed to accomplish *QueryTimeSeries*. The longer the paths, the longer the latency. We also use the subscripts to denote multiple instances of the same component. For example, $Zone\ Mixer_{1..n}$ denotes that n instances of *Zone Mixer* are involved in the RPC sequences of the scenario. These sequence diagrams enable us to estimate the total number of RPCs needed. For example, a *Root Mixer* queries n *Zone Mixer*, so there are n RPCs between the two components.

As we introduced in Section II, the prominent risks of the current Monarch system is availability and maintainability. The main problem is that the *Leaf* component takes multiple responsibilities. Given the increased coupling among classes and arbitrary APIs, it takes a longer to make changes to *Leaf*. Handling multiple responsibilities on one component risks the system's availability given the rapid growth of users and requests.

V. Model Proposed New Architecture

In order to handle the challenges mentioned in Section II, The TLs proposed to change the architecture of Monarch. The main idea is to decouple the *Leaf* component into three components, each responsible for data computation, indexing, and storage respectively. They created a box-and-line style diagram to model the proposed new architecture as depicted in Figure 2. This diagram has similar problems of ambiguity, mixing static component structures with run-time message sequences, and mixing static components, their run-time instances, and classes within the components. Using this diagram, it is impossible to tell how the Replica Resolution and Query passes will be affected by the new design.

To avoid the ambiguity and illustrate how the newly proposed design will impact qualities, the team created component (Figure 6) and sequence diagrams (in Figure 7) to model static and run-time views separately in the new design. As shown in Figure 6, the *Leaf* component is decoupled into three components: *Leaf Mixer*, new *Leaf* and *Leaf Index Server*.

Leaf Index Server is the only component involved in the Replica Resolution pass because it contains all the data quality information of targets, and the addresses of *Leaves*. It stores the indexes and keys of the time series data and their hosting *Leaves*, and provides the keys to the *Zone Mixer*. *Leaf* is simplified and becomes a key-value store of time series data. *Leaf Mixer* is responsible for computing (e.g., alignment, aggregation and join) time series data, and uses the same, simplified *Read* API to interact with both *Leaf* and *Repository*.

Figure 7 depicts the sequence diagrams modeling how RPCs in the *QueryTimeSeries* scenario route through these new components. It shows that the Replica Resolution will end with the new *Leaf Index Server*, rather than *Leaf*. In the Query pass, two new hops to *Leaf Mixer* and *Leaf Index Server* are added to

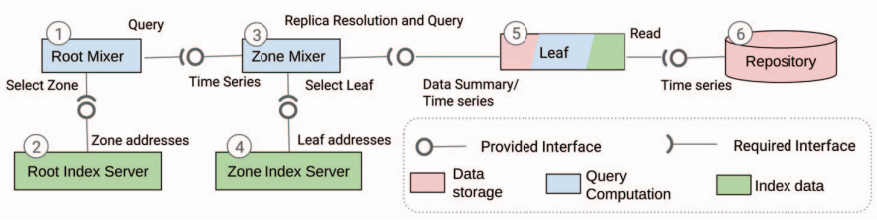
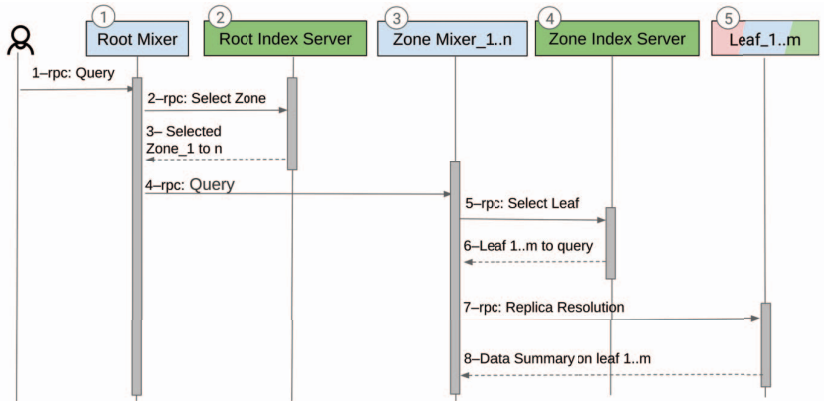
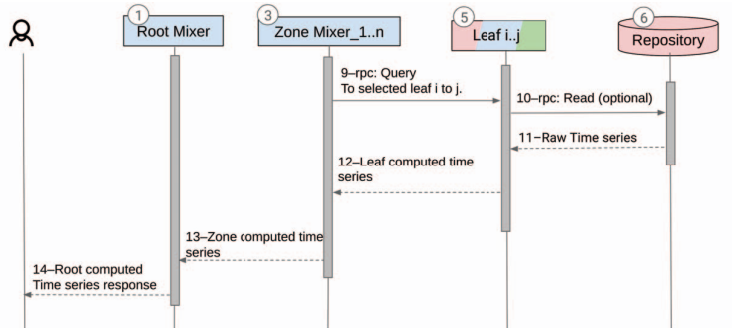


Fig. 4: Current design: scenario-based component diagram



(a) Sequence diagram: the Replica Resolution pass



(b) Sequence diagram: the Query pass

Fig. 5: Current design: scenario-based sequence diagrams

accomplish the query. The *Leaf Mixer* component will interact with the *Leaf* component and *Repository* separately, instead of relying on the *Leaf* to interact with *Repository*. As a result, 16 to 18 RPC hops are needed to accomplish this scenario.

VI. Quality Trade-off Analysis

Given the component diagrams and sequence diagrams of the current and proposed new architecture, it becomes easier to analyze the quality attributes, and how they will change in the new design. Table II presents the level of importance of each QA, their risk levels in the current design, and how these qualities will change in the new design based on the analysis in this section. A quality attribute with high risk means that the current system may not always meet the target response measures as the system scale grows, as listed in Table I. The risk level is determined based on how often the corresponding SLO is violated according to the execution data obtained in real-time. For example, if the system can't always meet its availability SLO, it means that its availability is at high risk.

A. Availability

Monarch has to return the results successfully for $\geq 99.99\%$ of queries, which is the most important quality attribute. We use both component and sequence diagrams to analyze how availability will change.

a) Current design: Currently availability is at risk for two reasons. First, as we discussed in the Section II, Monarch has experienced 2-5x year-to-year growth over the last several years. As the number of time series grows, the number of *Leaves* in a zone, and the number of *Leaves* involved in both Replica Resolution and Query RPCs grow drastically. Even if a small number of *Leaves* run into problems, the overall

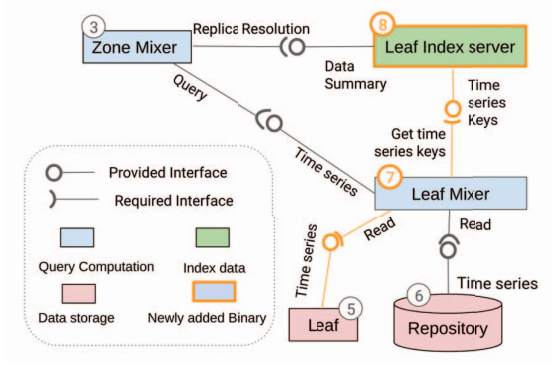


Fig. 6: Proposed new design: scenario-based component diagram

query availability will be affected. There are usually a certain (small) number of bad machines in a cluster. As the number of involved *Leaves* grows, the chance of reaching a problematic *Leaf* (on a bad machine) increases.

We model the availability of the *QueryTimeSeries* scenario to be the product of availability of all the *instances* along the RPCs path. This is just an approximation to illustrate how availability changes before and after the architecture change. In practice, individual node failures can be tolerated with various mechanisms (like data replication). From the sequence diagrams depicted in Figure 5, we can observe how many RPCs and how many component *instances* are involved. The availability of a query without touching *Repository* in the current architecture can be modeled as:

$$A_{current} = A_r A_{ri} A_z^n A_{zi}^n A_l^m \tag{1}$$

where A_r, A_{ri}, A_z, A_{zi} and A_l denote the availability of an instance of *Root Mixer*, *Root Index Server*, *Zone Mixer*, *Zone Index Server*, and *Leaf* respectively. Note that availability of the stateless components like *Root Mixer* and *Zone Mixer* is close to 100% because the failures could be tolerated by *retries*. The failures of *Root Index Server* and *Zone Index Server* are also negligible because the RPCs could be hedged to multiple instances in multiple clusters. So the major contributor to $A_{current}$ is A_l^m. The number of *Leaves* involved in Query is a small subset of *Leaves* involved in Replica Resolution. Our measurement shows that the ratio of *Leaves* involved in Query and Replica Resolution can be as low as 20% in huge zones [7]. So we only use A_l^m to represent the availability of the involved *Leaf* instances.

Second, since all three responsibilities are clustered into one *Leaf* component, if one function has a bug, or the node runs into resource problems due to a spike of requests (including writes and queries), all the queries on the *Leaf* are affected.

b) New design: The availability of the new design, as shown in Figure 7, can be modeled as:

$$A_{new} = A_r A_{ri} A_z^n A_{zi}^n (A_{li}^{m'} A_{lm}^k A_{l'}^{j-i+1}) \tag{2}$$

where the A_{li}, A_{lm} and $A_{l'}$ represent the availability of an instance of *Leaf Index Server*, *Leaf Mixer* and the new *Leaf*. A_{new} is significantly higher than $A_{current}$ for the following reasons:

First, A_l should be lower than the product of A_{li}, $A_{l'}$ and A_{lm}, although it is the combination of the three responsibilities. Because the *Leaf Mixer* is a stateless component, queries can utilize *retries* to tolerate failures. As a result, A_{lm} is negligible because it is close to 100%.

Second, m is much larger than m' because the new *Leaf Index Server* only stores the time series key information, so the data is more dense and the number of instances involved in the query will reduced.

Third, $j-i+1$ is much smaller than m because the number of *Leaves* involved in the Query is much smaller than the number of *Leaves* involved in Replica Resolution, i.e., $j-i+1$ can be as small as 20% of m on average in huge zones.

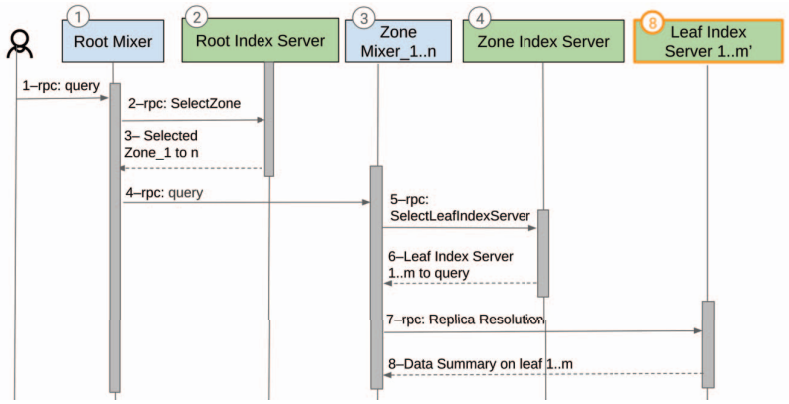
Moreover, as shown in the component diagram in Figure 6, since the computation and indexing functions are decoupled from the storage component, they are isolated from the failures caused by writing to the in-memory storage. This way the probability of failures can be significantly lowered. In addition, the two storage layers, i.e., *Leaf* and *Repository*, do not depend on each other any more. This significantly improves the availability of queries that only uses one of the storage layers. Furthermore, the system can also serve partial data to the query that combines data from both storage layers, and provides failure isolation between the two storage layers.

B. Maintainability

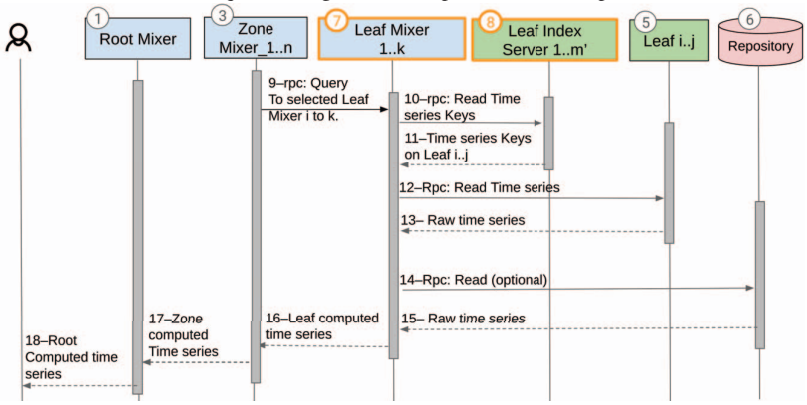
The team measures this attribute from three dimensions as listed in Table I: 1) the number of days to rollout a change, 2) the number of incidents allowed within a month on the query functions, and 3) the number of hours to root cause an incident. The team believes this attribute is of medium importance but with a high risk. We use component diagrams to analyze how the maintainability will change in the new design.

a) Current design: Since multiple responsibilities are currently aggregated within one component, *Leaf*, (as shown in Figure 4), the classes within it also couple with each other through complicated APIs. This makes it difficult to understand and harder to implement new features. When *Leaves* experience problems in production, it is also very hard to determine which part is the root-cause, or whether it is caused by the query scenarios or other activities. Currently *Leaves* are involved in 18 out of 20 Monarch performance dashboards. When there is a problem in *Leaves*, it's hard to root cause the problem based on the dashboards because most of them will be broken. In addition, problems in data storage would also make *Leaves* misbehave, which in turn, affects the performance of the query, leading to more production incidents.

b) New design: Decoupling the responsibilities into different components (as shown in Figure 6) enables the team to roll out different changes independently and in parallel, speeding up the code-to-production time. Currently, the rollout of *Zone Mixers* is around 30% faster than *Leaves*. In the new design, query related features no longer need to



(a) Sequence diagram: the Replica Resolution pass



(b) Sequence diagram: the Query pass

Fig. 7: Proposed new design: scenario-based sequence diagrams

be rolled out with Leaves, so the release time will also be reduced by 30% approximately. The rollback of one function won't affect the other functions anymore. Restricted APIs and decoupled components make it easy to identify root cause, and improved failure isolation will reduce the the number of incidents per month. With the new design, we can split the 18 dashboards into 5 write related dashboards and 13 query related dashboards, which can be used to pinpoint different problems more efficiently. It is even possible to shard the site reliability team into two shards, query and storage, each of which only responsible for the affected components.

C. Latency

This is another critical quality attribute and SLO: $\geq 99\%$ of queries must complete within a certain amount of time. We use sequence diagrams to analyze this attribute.

a) Current design: as depicted in Figure 5, there are 12 to 14 RPC hops needed to accomplish the *QueryTimeSeries* scenario. Based on the metrics collected from Monarch, the latency of queries is currently well below the target, and the team considers that this attribute is of low risk. In addition, from the team's experience, the number of hops on RPC paths is not the major cause of tail latency, which is mainly affected by data processing time.

b) New design: Figure 7 shows that in the new design, 16 to 18 RPC hops are needed to accomplish the query scenario. Our experiment revealed that the latency will increase to some extent, but the system can still meet the target measures without influencing user experiences. On the other hand, although the data volume output by *Leaves* has increased (because there is no computation including aggregations on the *Leaf* anymore), the added layer of k *Leaf Mixers* splits the majority of work on current Zone Mixers into k shards, which will increase the parallelism and mitigate the latency overhead. In addition, as there are fewer number of *Leaves* involved in the query (from m to $j - i + 1$, i.e., as much as a 80% reduction), the query tail latency could be further improved because a query is less likely to be impacted by the small number of hot-spotted leaves.

D. Resource efficiency

The target is that the query processing should take less than 10% of additional CPU and memory. We use both component and sequence diagrams to analyze the trade-offs.

a) Current design: the team marked this attribute as "low risk" because the current resource utilization of the system hasn't reached its limit and there are slack resources to use.

b) New design: As shown in Figure 7, the *QueryTimeSeries* user scenario now has 4 additional RPC hops. Introducing additional RPCs comes with the overhead of newtork I/O, CPU time to encode and decode the RPCs, and memory consumption to buffer those RPCs. Given that the current system still has some slack resources, it should be able to handle the additional resource overhead.

Furthermore, the system could utilize the resources better with the new design. Since the spiky load of query computation are moved out of data storage, and transferred into a stateless component, *Leaf Mixer*, as shown in Figure 6, both *Leaves* and *Leaf Mixers* can operate with higher resources (specifically memory) utilization because there is no need for a large headroom. This change does not sacrifice the reliability, because the storage workload is very stable and the failure of stateless component can be tolerated by retries.

Table II summarizes these trade-off analyses, indicating that for the quality attributes that are of medium or high importance, and at medium or high risk, the new design will significantly improve them and mitigate the risk. For the other quality attributes that will be degraded in the new design, they are either of low importance or low risk. The advantage of the new design clearly outweighs the current one. Note that none of the above analyses were possible before the component and sequence diagrams were created.

VII. Feedback and Impact

We presented the re-architecting plan to the team, using these scenario-based models to illustrate how key qualities of the system will change in the new designs. The overall process was well received. The team found that the canonical UML component and sequence diagrams are easy to understand, and acknowledged the advantage of differentiating static view from run-time views. Below are a few comments:

"I really like how this slide represents how the jobs interact with each other and what the responsibilities of each component are."

"Loved your presentation,... I also liked that you broke down the impact using the industry standard terms (availability, consistency, scalability etc)...."

"... I really liked the diagrams you used, particularly the -o (provided interface))- (required interface) notation. also, having legends on the diagrams. super useful."

The managers of Monarch commented that this process should be recommended to other Monarch teams, as well as other Google projects. A few days after we presented and shared the process, two new projects of Monarch adopted UML component and sequence diagrams in their design documents, and two other teams within Google started to learn and tried to adopt this process to support the refactoring of their systems.

VIII. Lessons Learned

In this section, we summarize the lessons learned from this experience, that is, how to support trade-off analysis among multiple quality attributes and justify re-architecting decisions, by creating lightweight models based on a few critical user and quality attribute scenarios. The most important lesson is to limit the models to high-level abstractions, expressing these scenarios only, and avoiding unnecessary complexity.

Specifying scenarios makes models lightweight, concise, and relevant. Monarch has dozens of components and binaries, and we just modeled a small part that is relevant to the query scenario. Each component we modeled provides many APIs, and we only modeled those that are involved in the scenario, and have impacts on these quality attributes, just enough to conduct the trade-off analysis. UML has the reputation of being overly complicated [3], [15], [16], but our experience indicates that it is possible to just use a small portion of the overall UML family, guided by concrete user and quality attribute scenarios. Only creating high-level models with sufficient abstraction is the key to make these models useful.

Separating static and run-time views makes models precise. Modeling static structures and run-time RPC sequences separately turns out to be an effective way to reduce ambiguity. The "`provided interface`" and "`required interface`" notations are well accepted by the developers since they merely model the existence of relevant APIs, rather than their interactions. Sequence diagrams, on the other hand, visualize RPC sequences using time lines, and these sequences are critical in terms of analyzing the trade-offs among availability, latency, and maintainability. Sequence modeling has been widely used in system designs, such as embedded systems and real-time systems [17]–[19], but are still not widely adopted in software design.

Creating self-explanatory diagrams. It is also important to have non-ambiguous legends and make the diagrams self-explanatory. We have to accept the fact that not all software developers are familiar with all the notations in formal models such as component and sequence diagrams. In order to communicate with both developers and management, we have to make sure that these diagrams are understandable and avoid using overly complicated notations that cannot be explained using legends or one or two sentences.

Simplified notations. Although we recommend the team to adopt component and sequence diagrams, we didn't follow standard UML notations strictly. For example, we didn't use stereotype or special icons to denote "*Component*". Rather, we just used a box to represent an independently executable, deployable component. Similarly, in these sequence diagrams, we also used a simple box to represent an instance of a binary at run-time, rather than using specialized UML notations. The assumption is that a box represents a static entity in a static view, and represents a run-time instance in a dynamic view. Our rationale is, again, keep the diagrams simple: not to introduce new notations until it is absolutely necessary.

Supporting customized notations. In these Monarch dia-

TABLE II: Risk and trade-off analysis

Quality Attributes	Importance	Current Architecture	Proposed New Architecture	
		Risk	Tradeoffs	Notes
Availability	High	High	+	Failure isolation between different responsibilities Fanout reduction leads to availability increase Move computation out from stateful components and allow retries
Maintainability	Medium	High	+	Code changes to the query, index and storage are now independent Simplify the API of leaf to a key-value store
Latency	High	Low	-	Four more RPC hops are needed to accomplish a query
Resource efficiency	Medium	Low	-	Additional RPC processing overhead

grams, we employed simple color coding to denote three types of responsibilities: computation, index generation, and data storage. In Figure 1, Figure 4, and Figure 5, it is clear that the *Leaf* component is taking all three responsibilities in the current design. Figure 6 and Figure 7 depict how they will be decoupled into three different components in the new design. This color-coding notation is not part of the standard UML, but is an innovative and effective way to demonstrate how the system will be refactored and why.

In summary, although we recommend UML modeling, we also recommend allowing variations, customization, and extension in term of notations used in diagrams, following the principles of usefulness, simplicity, and non-ambiguity. It is critical that the models are concise, precise, relevant to the tasks the developers are working on, and help developers address the challenges they are facing. We created these diagrams using Google Slides and a general graphic creation application, rather than UML-specific tools. It took us a few hours in total to create, revise, and refine these diagrams. The time cost is affordable.

In this study, we only used component and sequence diagrams. It is possible that in other situations, a different type of models will be needed. For example, a C4[2] context model can be useful to analyze how the components within a system interact with infrastructure components or third-party libraries. UML deployment diagrams can also be useful to demonstrate how different components should be installed in hardware devices. Again, we only create these models when needed, at the proper level of abstraction.

This is the first attempt to support architecture redesign for a planet-scale legacy system using scenario-based, lightweight models and multiple QA analysis within Google. The fact that these models were quickly adopted by other teams indicate the possibility to apply these techniques to other stages of a software development process, supporting feature analysis, quality attribute trade-off analysis, as well as refactoring decision-making.

In this experience, we have tackled the challenges of creating abstract and lightweight models that are just enough to express the scenarios in question, and avoided unnecessary complexity and details, which was the key to facilitate the quick adoption of these basic UML models and related concepts by multiple Google projects. The experience also provided an example and guidelines on how to leverage these models to facilitate design related decision-making. It is possible that more complicated, and more than one changes to the architecture are needed in the future. This experience is the first step for the team to adopt more formal and sophisticated models to provide quantitatively, automated, and more accurate predictions, as researchers have proposed in the domain of performance engineering [20], [20]–[23].

[2]https://c4model.com/

IX. Related Work

Modeling and documenting software architecture have been proposed and taught in classrooms for a few decades. Unified Modeling Language (UML) [1] is considered as the de facto standard of software design modeling techniques. Clements et al. [9] proposed that software systems should be modeled using different types of views. Bass et al. [2] also proposed that software architecture has different types of structures: module, component-connector, and allocation. C4 is another software modeling technique gaining popularity recently. It models software from four levels: Context, Containers, Components and Code. However, the application of these models are still limited in practice.

A recent survey [3] of 314 practitioners from 180 IT companies reveals that although most practitioners recognized the usefulness of UML in terms of improving system understandability and quality, as many as 74.8% of them did not use UML at all because of some reasons such as lack of knowledge about modeling, culture of the company, time constraints, as well as the difficulty of keeping the models updated. Researchers have conducted various surveys on the application of UML in practice [3], [15], [16], [24]–[26]. Basically most practitioners recognize the necessity and benefit of modeling, but they couldn't justify the cost of managing the complexity and evolution. This is consistent with our observations within Google and other companies we worked with.

Many other architecture description languages (ADL) [27], such as ACME [28], AADL [29] and Wright [30], have been proposed, but they face similar challenges for large-scale industrial projects. Researchers also proposed a number of architecture review and assessment approaches using various models [31]–[34]. Our report complements these prior works with a real experience of combining multiple abstract models to justify a significant architecture-level revision for an extra large legacy system within Google.

Service level objectives (SLO) and site reliability engineering (SRE) are widely used terms within Google and software industry to define important system attributes, such

as performance and reliability. These SLOs form a subset of quality attributes [2] (used to be called "non-functional requirements"), a term used in academic research and education. SLO and SRE usually rely on testing and continuous online monitoring to ensure the most important quality of the running system, and the practitioners have to count on their intuition and experiences to analyze the trade-offs among these attributes. Here we integrate user scenarios and lightweight models to enable rigorous trade-off analysis.

In the domain of software performance engineering, using UML and other models to analyze and predict software performance has been studied for more than a decade [20]–[23], [35]–[40]. In particular, Petriu's [22] recent work is most relevant to ours since we share similar purposes of using multiple models to analyze multiple quality attributes. Different from Petriu's work that employs an sophisticated ecosystem of heterogeneous modeling artifacts to support multiple features such as consistent co-evolution of the software models and cross-model traceability, we reported an experience of communicating and justifying the redesign of an extra large system using lightweight, basic UML models under the specific, realistic scenarios. We created abstract models just enough to analyze and communicate the trade-offs. We believe this is the first step towards the adoption of more sophisticated models. Most other SPE works focus on abstracting the interaction among software components to assess performance only, while in this study, we analyzed the trade-offs among multiple quality attributes, enabled by the combination of component and sequence models created based on critical user scenarios.

In the domain of Model-based engineering (MBE), models are widely used in the design and development of embedded systems. These models are usually used for the purpose of simulations, code generation, and documentation [17], [19], [41]–[43]. In software design, however, formal modeling is not part of the software development process yet. Most teams create design documents at the beginning of the project, usually containing informal diagrams that are not sufficient to support complicated decision-making or trade-off analysis. We expect that this positive experience inspires more effective adoption of software architecting and modeling into daily development processes.

X. Conclusion

In this paper, we report our experiences of using scenario-based formal models to conduct quality trade-off analysis, communicating and justifying a proposed re-architecting strategy for Monarch. We first identify a few critical user scenarios and their associated quality attribute scenarios, in which the concrete artifacts involved are specified. These scenarios enabled us to create precise and concise models with a proper level of abstraction: component diagrams with only scenarios-related components, and sequence diagrams only modeling how RPCs route through these components. We create these models for both the current design and the proposed new design, which made it easy to conduct trade-off analysis. This process was well-received by the team and recommended to other teams within Google. Soon after our presentation, two new projects adopted these diagrams in their design documents, and two other products within Google are adopting the process. This experience indicates that it is possible to integrate formal models and QA analysis into software development process continuously, communicating features, assessing quality attributes, or justifying design and architecture decisions.

XI. Acknowledgements

We thank Adam Tart, Ming Chen, Nick Sakharov and many other Monarch engineers for their contributions to the design. We would like to thank John Wilkes for the valuable feedback of the paper. This research was partially supported by the United States National Sciences Foundation grants 1835292, 1823177, and 2213764.

REFERENCES

[1] G. Booch, J. Rumbaugh, and I. Jacobson, *Unified Modeling Language User Guide. The 2nd Edition*. Addison-Wesley, 2005.

[2] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 4th ed. Addison-Wesley, 2021.

[3] E. Júnior, K. Farias, and B. Silva, "A survey on the use of UML in the brazilian industry," in *Proceedings of the XXXV Brazilian Symposium on Software Engineering*, ser. SBES '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 275–284. [Online]. Available: https://doi.org/10.1145/3474624.3474632

[4] E. Guimaraes, M. Manica, L. Gonçales, V. Bischoff, B. da Silva, and K. Farias, "On the UML use in brazilian industry: A state of the practice survey," 07 2018.

[5] H. Störrle, "How are conceptual models used in industrial software development? a descriptive survey," in *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering*, ser. EASE'17. New York, NY, USA: Association for Computing Machinery, 2017, p. 160–169. [Online]. Available: https://doi.org/10.1145/3084226.3084256

[6] A. M. Fernández-Sáez, M. R. Chaudron, and M. Genero, "An industrial case study on the use of UML in software maintenance and its perceived benefits and hurdles," *Empirical Softw. Engg.*, vol. 23, no. 6, p. 3281–3345, dec 2018. [Online]. Available: https://doi.org/10.1007/s10664-018-9599-4

[7] C. Adams, L. Alonso, B. Atkin, J. P. Banning, S. Bhola, R. Buskens, M. Chen, X. Chen, Y. Chung, Q. Jia, N. Sakharov, G. T. Talbot, A. J. Tart, and N. Taylor, Eds., *Monarch: Google's Planet-Scale In-Memory Time Series Database*, 2020.

[8] Y. Rastegari and F. Shams, "Optimal decomposition of service level objectives into policy assertions." *The Scientific World Journal*, vol. 2015 (2015): 465074.

[9] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, P. Merson, R. Nord, and J. Stafford, *Documenting Software Architectures: Views and Beyond. The 2nd Edition*. Addison-Wesley, 2010.

[10] A. Merchant, "Keynote address II: Optimal flash partitioning for storage workloads in google's colossus file system." Broomfield, CO: USENIX Association, Oct. 2014.

[11] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford, "Spanner: Google's Globally-Distributed database," in *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. Hollywood, CA: USENIX Association, Oct. 2012, pp. 261–264. [Online]. Available: https://www.usenix.org/conference/osdi12/technical-sessions/presentation/corbett

[12] B. Samwel, J. Cieslewicz, B. Handy, J. Govig, P. Venetis, C. Yang, K. Peters, J. Shute, D. Tenedorio, H. Apte, F. Weigel, D. G. Wilhite, J. Yang, J. Xu, J. Li, Z. Yuan, C. Chasseur, Q. Zeng, I. Rae, A. Biyani, A. Harn, Y. Xia, A. Gubichev, A. El-Helw, O. Erling, A. Yan, M. Yang, Y. Wei, T. Do, C. Zheng, G. Graefe, S. Sardashti, A. Aly, D. Agrawal, A. Gupta, and S. Venkataraman, "F1 query: Declarative querying at scale," 2018, pp. 1835–1848. [Online]. Available: http://www.vldb.org/pvldb/vol11/p1835-samwel.pdf

[13] A. Verma, L. Pedrosa, M. R. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at Google with Borg," in *Proceedings of the European Conference on Computer Systems (EuroSys)*, Bordeaux, France, 2015.

[14] R. Mo, Y. Cai, R. Kazman, L. Xiao, and Q. Feng, "Architecture anti-patterns: Automatically detectable violations of design principles," *IEEE Transactions on Software Engineering*, pp. 1–1, 2019.

[15] M. Petre, "UML in practice," in *2013 35th International Conference on Software Engineering (ICSE)*, 2013, pp. 722–731.

[16] W. J. Dzidek, E. Arisholm, and L. C. Briand, "A realistic empirical evaluation of the costs and benefits of UML in software maintenance," *IEEE Transactions on Software Engineering*, vol. 34, no. 3, pp. 407–432, 2008.

[17] G. Liebel, N. Marko, M. Tichy, A. Leitner, and J. Hansson, "Model-based engineering in the embedded systems domain: An industrial survey on the state-of-practice," *Softw. Syst. Model.*, vol. 17, no. 1, p. 91–113, feb 2018. [Online]. Available: https://doi.org/10.1007/s10270-016-0523-3

[18] T. Shailesh, A. Nayak, and D. Prasad, "An UML based performance evaluation of real-time systems using timed petri net," *Computers*, vol. 9, no. 4, 2020. [Online]. Available: https://www.mdpi.com/2073-431X/9/4/94

[19] J. Trowitzsch, A. Zimmermann, and G. Hommel, "Towards quantitative analysis of real-time UML using stochastic petri nets." vol. 2005, 01 2005.

[20] C. U. Smith, *Performance engineering of software systems*, ser. Software Engineering Institute series in software engineering. Addison-Wesley, 1990.

[21] D. C. Petriu, C. M. Woodside, D. B. Petriu, J. Xu, T. Israr, G. Georg, R. France, J. M. Bieman, S. H. Houmb, and J. Jürjens, "Performance analysis of security aspects in UML models," in *Proceedings of the 6th International Workshop on Software and Performance*, ser. WOSP '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 91–102. [Online]. Available: https://doi.org/10.1145/1216993.1217010

[22] D. Petriu, "Integrating the analysis of multiple non-functional properties in model-driven engineering," *Software and Systems Modeling*, vol. 20, 12 2021.

[23] C. Canevet, S. Gilmore, J. Hillston, L. Kloul, and P. Stevens, "Analysing UML 2.0 activity diagrams in the software performance engineering process," *SIGSOFT Softw. Eng. Notes*, vol. 29, no. 1, p. 74–78, jan 2004. [Online]. Available: https://doi.org/10.1145/974043.974055

[24] C. Lange, M. Chaudron, and J. Muskens, "In practice: UML software architecture and design description," *Software, IEEE*, vol. 23, pp. 40 – 46, 04 2006.

[25] M. Ozkaya and F. Erata, "A survey on the practical use of UML for different software architecture viewpoints," *Information and Software Technology*, vol. 121, p. 106275, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584920300252

[26] G. Scanniello, C. Gravino, M. Genero, J. A. Cruz-Lemus, and G. Tortora, "On the impact of UML analysis models on source-code comprehensibility and modifiability," *ACM Trans. Softw. Eng. Methodol.*, vol. 23, no. 2, apr 2014. [Online]. Available: https://doi.org/10.1145/2491912

[27] N. Medvidovic and R. Taylor, "A classification and comparison framework for software architecture description languages," *IEEE Transactions on Software Engineering*, vol. 26, no. 1, pp. 70–93, 2000.

[28] D. Garlan, R. Monroe, and D. Wile, "ACME: An architecture description interchange language," in *CASCON First Decade High Impact Papers*, ser. CASCON '10. USA: IBM Corp., 2010, p. 159–173. [Online]. Available: https://doi.org/10.1145/1925805.1925814

[29] P. H. Feiler, B. Lewis, S. Vestal, and E. Colbert, "An overview of the sae architecture analysis & design language (AADL) standard: A basis for model-based architecture-driven embedded systems engineering," in *Architecture Description Languages*, P. Dissaux, M. Filali-Amine, P. Michel, and F. Vernadat, Eds. Boston, MA: Springer US, 2005, pp. 3–15.

[30] R. Allen, R. Douence, and D. Garlan, "Specifying and analyzing dynamic software architectures," in *Proceedings of the 1998 Conference on Fundamental Approaches to Software Engineering (FASE'98)*, Lisbon, Portugal, March 1998, an expanded version of a the paper "Specifying Dynamism in Software Architectures," which appeared in the Proceedings of the Workshop on Foundations of Component-Based Software Engineering, September 1997.

[31] D. Sobhy, R. Bahsoon, L. Minku, and R. Kazman, "Evaluation of software architectures under uncertainty: A systematic literature review," *ACM Trans. Softw. Eng. Methodol.*, vol. 30, no. 4, aug 2021. [Online]. Available: https://doi.org/10.1145/3464305

[32] R. C. Soares, V. d. Santos, and E. Y. Nakagawa, "Continuous evaluation of software architectures: An overview of the state of the art," in *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*, ser. SAC '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1425–1431. [Online]. Available: https://doi.org/10.1145/3477314.3507318

[33] V.-P. Eloranta, U. van Heesch, P. Avgeriou, N. Harrison, and K. Koskimies, "Lightweight evaluation of software architecture decisions," in *Relating System Quality and Software Architecture*, I. Mistrik, R. Bahsoon, P. Eeles, R. Roshandel, and M. Stal, Eds. Boston: Morgan Kaufmann, 2014, pp. 157–179.

[34] P. Clements, P. Gordon, R. Kazman, and M. Klein, *Evaluating Software Architectures: Methods and Case Studies*. Addison-Wesley Professional; 1st edition, 2001.

[35] C. U. Smith, C. M. Lladó, V. Cortellessa, A. D. Marco, and L. G. Williams, "From UML models to software performance results: an SPE process based on XML interchange formats," in *Proceedings of the Fifth International Workshop on Software and Performance, WOSP 2005, Palma, Illes Balears, Spain, July 12-14, 2005*. ACM, 2005, pp. 87–98. [Online]. Available: https://doi.org/10.1145/1071021.1071030

[36] M. Melià, C. M. Lladó, C. U. Smith, and R. Puigjaner, "Experimentation and output interchange for petri net models," in *Proceedings of the 7th International Workshop on Software and Performance, WOSP 2008, Princeton, NJ, USA, June 23-26, 2008*, A. Avritzer, E. J. Weyuker, and C. M. Woodside, Eds. ACM, 2008, pp. 133–138. [Online]. Available: https://doi.org/10.1145/1383559.1383576

[37] M. Rapp, M. Scheerer, and R. Reussner, "Design-time performability optimization of runtime adaptation strategies," in *Companion of the 2022 ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 113–120. [Online]. Available: https://doi.org/10.1145/3491204.3527471

[38] J. I. Requeno, I. Gascón, and J. Merseguer, "Towards the performance analysis of Apache Tez applications," in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 147–152. [Online]. Available: https://doi.org/10.1145/3185768.3186284

[39] T. Altamimi and D. C. Petriu, "Incremental change propagation from uml software models to lqn performance models," in *Proceedings of the 27th Annual International Conference on Computer Science and Software Engineering*, ser. CASCON '17. USA: IBM Corp., 2017, p. 120–131.

[40] C. Li, T. Altamimi, M. Zargar, G. Casale, and D. Petriu, "Tulsa: A tool for transforming uml to layered queueing networks for performance analysis of data intensive applications," 08 2017, pp. 295–299.

[41] B. Anda, K. Hansen, I. Gullesen, and H. K. Thorsen, "Experiences from introducing UML-based development in a large safety-critical project," *Empirical Software Engineering*, vol. 11, no. 4, pp. 555–581, 2006.

[42] M. Brambilla, J. Cabot, and M. Wimmer, "Model-driven software engineering in practice," *Synthesis lectures on software engineering*, vol. 3, no. 1, pp. 1–207, 2017.

[43] J. Hutchinson, J. Whittle, and M. Rouncefield, "Model-driven engineering practices in industry: Social, organizational and managerial factors that lead to success or failure," *Science of Computer Programming*, vol. 89, pp. 144–161, 2014.