Aggregate Architecture Simulation in Event-Sourcing Applications using Layered Queuing Networks

Gururaj Maddodi
Utrecht University
Utrecht, Netherlands
g.maddodi@uu.nl

Slinger Jansen
Utrecht University
Utrecht, Netherlands
slinger.jansen@uu.nl

Michiel Overeem
AFAS Software
Leusden, Netherlands
michiel.overeem@afas.nl

ABSTRACT

Workload intensity in terms of arrival rate and think-time can be used accurately simulate system performance in traditional systems. Most systems treat individual requests on a standalone basis and resource demands typically do not vary too significantly, which in most cases can be addressed as a parametric dependency. New frameworks such as *Command Query Responsibility Segregation* and *Event Sourcing* change the paradigm, where request processing is both parametric dependent and dynamic, as the history of changes that have occurred are replayed to construct the current state of the system. This makes every request unique and difficult to simulate. While traditional systems are studied extensively in the scientific community, the latter is still new and mainly used by practitioners. In this work, we study one such industrial application using *Command Query Responsibility Segregation* and *Event Sourcing* frameworks. We propose new workload patterns suited to define the dynamic behavior of these systems, define various architectural patterns possible in such systems based on *domain-driven design* principles, and create analytical performance models to make predictions. We verify the models by making measurements on an actual application running similar workloads and compare the predictions. Furthermore, we discuss the suitability of the architectural patterns to different usage scenarios and propose changes to architecture in each case to improve performance.

CCS CONCEPTS

• **Software and its engineering → Software performance**.

KEYWORDS

Software Performance Engineering, Software Architecture Variation, Command-Query Responsibility Segregation, Event Sourcing, Domain-Driven Design

ACM Reference Format:
Gururaj Maddodi, Slinger Jansen, and Michiel Overeem. 2020. Aggregate Architecture Simulation in Event-Sourcing Applications using Layered Queuing Networks. In *Proceedings of the 2020 ACM/SPEC International Conference on Performance Engineering (ICPE '20), April 20–24, 2020, Edmonton, AB, Canada.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3358960.3375797

ICPE '20, April 20–24, 2020, Edmonton, AB, Canada

ACM ISBN 978-1-4503-6991-6/20/04…$15.00
https://doi.org/10.1145/3358960.3375797

1 INTRODUCTION

Enterprise application workloads vary greatly, due to differences in the usage behavior of end-users [14]. Variation is typically caused by the types of business an organization conducts, e.g. a wholesale business will have large orders from the same customer while a supermarket has many single orders with a few items. For applications deployed in the cloud, the current approach is to simply scale the hardware. However, it may be profitable for the architects to use knowledge of the correlation between usage patterns and resource utilization to select an architecture for better performance.

Software Performance Engineering (SPE) [19], a methodology developed by the software engineering community, benefits software architects by incorporating performance objectives into the development process. SPE involves building *Software Execution Model*s that represent the control flow of the features of the software system using techniques such as UML sequence diagrams, execution graphs [19], transforming *software execution model* into *System Execution Model* representing the resource utilization of the software system in operation using techniques such as queuing networks [10, 15], stochastic petrinets [11], process algebras [8], etc., finally solving the *system execution model* models using analytical techniques to obtain performance predictions.

In this work, we investigate the effect of workload patterns on performance and resource metrics of component-based software using Event Sourcing (ES) and Command Query Responsibility Segregation (CQRS) [9, 24] patterns. CQRS and ES frameworks are popular in the industry, but require more academic study, as these patterns are becoming the standard in business systems development. *ES* pattern differs from other software design patterns, in that the processing of every request is different even for the same request types. This is because *ES* frameworks do not save the current state of the system, rather the state is rebuilt from all the history (or events) that have occurred in the system's life-cycle. As history is different for each request, the performance parameters are different for each request, which poses an interesting challenge to model such dynamic system behavior. We chose LQN to handle this as LQNs are expressive in defining system behavior [3, 10, 16, 18].

We use LQN to not only model the details of communications between the components in the software system but also to model replaying of the events to build the state. The three main aspects of modeling the *system execution model* for *ES* systems are: (1) the grouping of features into aggregates, where the features use same data in an aggregate, and features need to communicate for data access between the aggregates, (2) event store simulation with

new stream of events for every new instance of an aggregate is created, and (3) Event replay. The first aspect can be modeled by placing features in aggregates as single or separate stations in the LQN model so that features share a single queue or have separate queues for requests. The second aspect can also be modeled by using single or separate event store stations for the features in single aggregate or separate aggregates respectively. Lastly, the third aspect is modeled by multiplying the average number of events that can be occurring over a user session to the service demands at the identified stations and request types in the model.

The paper is organized as: In Sec. 2, we use the proven SPE methodology and describe the manner in which we position this research to align with the SPE methodology. We give a brief overview of the technical details of a CQRS and ES based software system (Sec. 3), and performance engineering concepts and scenarios in which the system is measured (Sec. 4).

The contributions of this work are mentioned in the sections:

Sec. 5 presents the performance model creation using LQN to model the system from the sequence diagram. We describe model creation and the calculation of the model parameters

Sec. 6 describes how the models are executed against the described scenarios and validate the models against measurements on an actual industrial application. We also discuss the choices architects can make based on the scenarios and propose improvements to the system to alleviate performance issues

We review related literature and discuss the differences and contributions to the existing literature that this work provides in Sec. 7. In In Sec. 8, we discuss the implications of the research, and conclude the paper with future work in the area.

2 RESEARCH METHOD

The research method we employed is as follows: (1) An experimental setup is designed to exercise workload on different *ES* system architectures, by analytical modeling and measurement, to illustrate how different architectures respond to particular workload patterns. (2) the experiment is conducted within the context of an exploratory theory-confirming case study. The case study adds context to the problems we are studying and enables us to discuss with experts the potential and challenges of the proposed solution.

Experimental Design - Two architecture scenarios in terms of the component organization were chosen in the experimental setup. We also define workload parameters to test system behavior as generated by end-users of an ERP software application specifically using *ES* pattern. The aim of the experimental setup is to evaluate the effects of workload patterns and test the accuracy of modeling the system. For measurements we use an existing *ES* implementation at a case-company, to closely match a realistic situation. Although the actual framework and application could not be shared due to proprietary reasons, the reader is referred to similar open-source CQRS and Event-Sourcing implementations such as Ncqrs for .Net[1], CQRS and ES for Windows Azure[2], and CQRS for .Net using Service Fabric[3]. In the research scenario, we have used several stable factors such as the hardware on which the measurements were done, and the OS and frameworks used to build the application. Local architects were consulted to validate the tests and results for the two identified architecture scenarios. For modeling, the SPE methodology was used by defining the *software execution model*, i.e the sequence diagram, and define *system execution model*s using LQN. The performance predictions are compared to the measurements obtained from the case company implementation. The measurements were repeated several times to eliminate spurious data caused by the operating system processes.

Case Study Context - The case study was done in an industry context. We have conveniently selected a company, anonymized ERPComp, in the Netherlands. ERPComp currently delivers a fully integrated ERP suite called ERPSoft, which is used daily by more than 1,000,000 professional users from more than 10,000 different End-User organizations. The case-company is developing a new version of their software, which is cloud-based, developed using the MDD approach, and implemented on a CQRS back-end. The work was performed on location at the company and we were given full access to the source code and the architects. The case study, experiment, and results were subjected to expert evaluation with two of the architects at ERPComp.

3 TECHNICAL CONCEPTS

CQRS is a distributed computing approach that advocates the paradigm - the requests to view the state of the system should not alter its current state. This basically divides the requests in the CQRS framework into (1) *Commands* defined as the requests that create or modify the state of the system, and (2) *Queries* that are the requests that access and present the current state.

CQRS uses Domain-driven Design (DDD) principles to specify the mechanism in which *commands* are handled and how events are generated for a specific *command*. DDD principles specify constructs called aggregates, which are groups of functional entities in a specific domain. These groups of functional entities can be processed together and have a consistent data boundary. An example is when a person places an order with several items in an online shopping application. Here, the entities are person, order, and order items and they belong to a single domain boundary of *Purchasing*. The aggregate has a hierarchical structure, i.e. the person first opens an order and then adds the items to purchase to the order.

Figure 1 shows a representation of CQRS. As the command-side handles the *commands*, the DDD aggregates exist on the command-model indicated by the circle in Figure 1. The root entity is called the Aggregate Root (AR). Entities that branch from the AR are Child Entities (CE) and can be accessed only through the AR. For the sake of simplicity, we distinguish entities as; (1) model-time entities are not instantiated, whereas (2) run-time entities are multiple instances of a model-time entity. This distinction helps to define the workload pattern for performance tests.

An *Aggregate* is registered to commands. There can be different command types for creating an AR and for adding instances of each type of CE. The commands also carry properties that have to map to the properties that are defined as part of the state of either the AR or the CE within the aggregate. Hence, an entity e is a set of properties, $e = \{pr_1, \ldots, pr_n \mid n \in \mathbb{N}\}$, where n is the number of properties in the entity. We assume that n is fixed is the same for

[1] https://github.com/pjvds/ncqrs
[2] https://github.com/abdullin/lokad-cqrs
[3] https://github.com/AFASResearch/CQRS-Playground

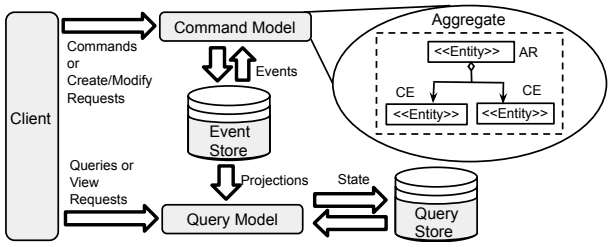

Figure 1: The Architecture of a typical CQRS Application with Command-model Showing the Structure of the Aggregates.

AR and CEs. For the purpose of simplicity, we also consider one CE under the AR. This model-time architecture is kept the same throughout the paper, and only the number of instances of CE at run-time are varied during tests. During run-time, the aggregate is defined as one instance of AR and k_{rt} instances of CE.

The CQRS pattern is frequently used with ES [5, 24]. In ES, new system states are recorded in the form of events, which are changes suggested to the aggregate properties using the business logic that is part of services within entities of the aggregate. Events are stored in an event store as depicted in figure 1. The event store enables a full replay of the events, leading to the reconstruction of the latest system state. ES store events as stream per aggregate instance. This means that for every instance of AR a new stream is created and events from the instances of CE goes into the same stream.

4 CASE DESCRIPTION

To illustrate, consider the case of an online order management application. The order is the main entity for the customer's purchase and one or more products could be present in the order represented as order items. Hence, an order entity (which is an AR) has several order items (CEs) under it. The order and each of the order items can have properties such as order ID, customer name, product name, and product cost. Using this example, we define two types of aggregate definitions. Please note that aggregates are similar to components in object-oriented engineering.

Single Aggregate Scenario - represented as $Aggr_s$, is where AR and CE are in the same domain boundary, and thereby end up in the same aggregate. The single aggregate scenario eliminates the need for communication as both the AR and the CEs are in the same domain boundary.

Multiple Aggregate Scenario - If the CE is defined as it's own domain boundary, then, in this case, we get two aggregates. Let us represent this case as $Aggr_m$. In our example system, we have two aggregates Order and Item where both are ARs. Since they are in their own domain boundary and do not share event streams, a communication mechanism might be required. An example is, in the online order management case, the item's price might have to be checked against a fixed purchase amount for the order. The implementation for this case is complex as data models for shared data for aggregates need to be consistent.

Figures 2[a] and 2[b] show the structure of a single and multiple aggregate respectively. In the scenario, the order is the AR and order items are the CEs. Figures 3[a] and 3[b] shows the sequence diagram of receiving and processing new order creation request and adding two items to the existing order for single and separate aggregate architecture respectively. The important things to be noticed in Figures 3[a] and 3[b] are: the communication between stations where there are more internal events in $Aggr_m$, and event store streams used where a new stream is used for every new item. The research question we want to answer is: given a workload pattern, which type of aggregate architecture suits the requirements.

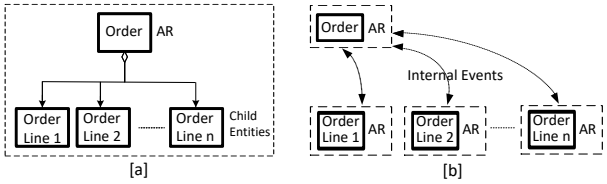

Figure 2: Aggregate Architecture Variations. [a] Single Aggregate, [b] Multiple Aggregate.

We define the workload pattern as the number of run-time instances of the CEs k_{rt}. The workload pattern characterizes the end-users that use the application as shown in our previous work [14].

Several performance metrics can be used to check the optimal architecture for a case. The time it takes to execute commands and generate events is relevant, and the throughput of the system. This is also proportional to total response time for a user session, as some business actions, such as ordering a product, requires real-time processing, while for bulk actions like salary calculations requires small response time. CPU and disk utilization are also interesting as hardware resources can be constrained.

We simulate different types of organizations on the order management system model by varying k_{rt} values. The variable n is fixed at 100 to get higher resource consumption scores. As k_{rt} is a continuous quantity, we performed simulation and measurements at an interval of 15 starting from 5 up to 500 in the tests. Hence, we generate one *OrderCreate* command and number of *AddItem* commands given by k_{rt}. In performance tests, the systems with the described aggregate architectures were subjected to the workload pattern using a synthetic workload generator [7], which was developed as part of research at ERPComp.

5 PERFORMANCE MODELING

Layerd Queing Networks (LQN) [6] is an extension of queuing networks that can simulate system operations with processing requests and queues as LQN allows interactions between the stations in the model. An LQN model is composed of tasks, entries, processors, calls, and activities. A task represents the servers with resources and a set of operations. Tasks have a queue with a discipline (d) and a multiplicity (m) for concurrency. A task has a processor modeling the physical resource (with m and d parameters) to perform operations. The operations are called entries, where each entry handles a specific request class. The parameters of entry are service demand (s), pure delay (Z), and the mean number of calls. Calls can be synchronous (y), asynchronous (z), or forwarding (f). The logic inside an entry can be specified using phases or activities. Reader is referred to [6] for more details of LQN and LQN Solver (LQNS).

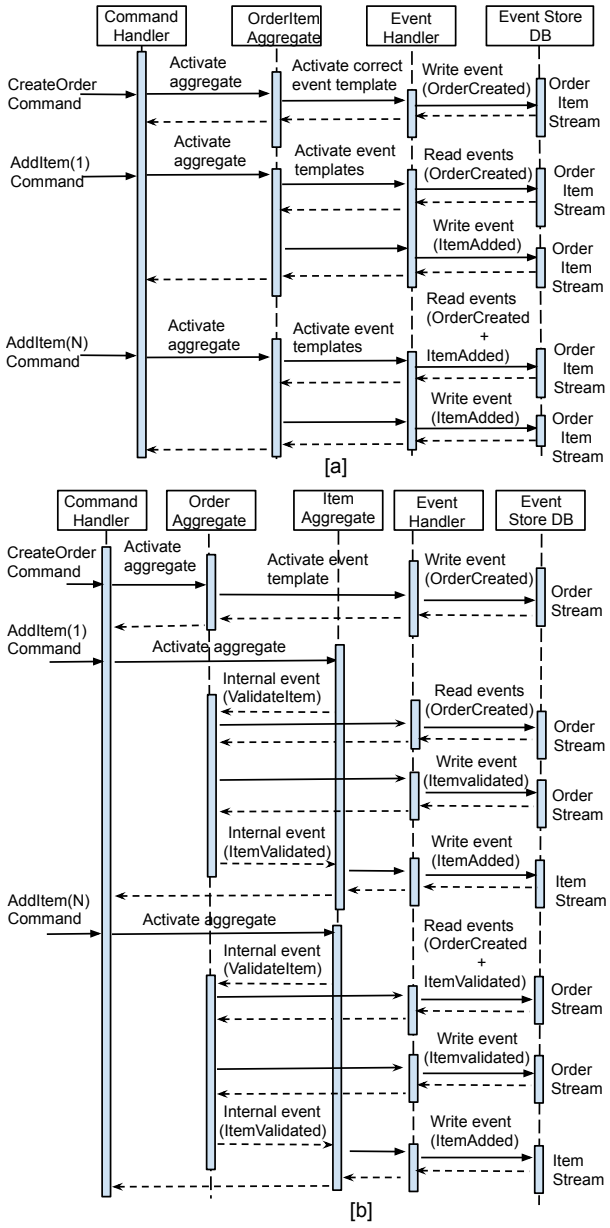

Figure 3: Sequence Diagram for Order Creation in an ES System with [a] Single Aggregate and [b] Separate Aggregate Architecture Types.

5.1 System Setup and Implementation

The service demands can be calculated from measurements on the system under test (SUT) for the request types. The application was written as a C# .NET web application on Windows 10 OS. The performance tests were run on a machine with Intel i5 3.4GHz 64-bit processor and 8 GB RAM. Microsoft SQL server with default max_connections parameter 32767 was used for event store.

For the application, the frameworks specified in the section 2 is used with JSON definitions for command and events. For CreateOrder and AddItem requests, 5 and 100 properties were defined respectively. In both the aggregate architectures, the properties are implemented using a Dictionary datatype containing key as name and value as a text string. In the single aggregate scenario, the entities were again implemented using Dictionary datatype with a key for representing the entity instance GUID and the value being Dictionary containing the property name and value as mentioned earlier. The properties were implemented as String with a length of 100 characters in the commands. The aggregate's function loops through the properties to generate resouce utilization.

5.2 LQN Models for Aggregate Architectures

The described architecture scenarios are modeled using LQN as shown in Figures 4 and 5 for $Aggr_s$ and $Aggr_m$ respectively. The sequence diagrams in Figures 3[a] and 3[b] show the three requests types: *CreateOrder*, *AddItem1*, and *AddItemN*, where N can be from 2 to k_{rt}. Due to the limitations in granularity of the measurements, i.e., we always measure events *CreateOrder* and *AddItem* together, we have had to separate them into two entries. We subtract the *OrderCreated* event replay execution from execution demand of *AddItemN* request to get only the execution demand for *ItemAdded* event. Therefore, we achieve event replay for all calls of *AddItemN* request class by multiplying the execution demands for *ItemAdded* event replay with $F_{mult} = [(k_{rt} - 1)/2]$. F_{mult}. In the model F_{mult} is used at read entries at event store and disk tasks and implementing events at aggregate tasks in the model.

In the model, we have the *tWorkloadGenerator* task with m=1 (as 1 user session is used as reasoned previously) and runs on processor *pWorkload* as we just look at the execution demands at the back-end of the system and the *tWorkloadGenerator* simulates an end-users' browser. As described in the execution scenario, is modeled by having an entry called *eLoad* with initial activity as *aCreateOrder*, followed by an *aAddItem1* activity, and *aAddItemN* in loop with iterations equal to k_{rt}-*1*. The Workload Generator's requests are handled by the Command Handler module, which are redirect to the aggregates. We include a *tCommandHandler* task to simulate the delay in redirection with its m=infinite and an infinite processor.

In the LQN models, for $Aggr_m$ multiple aggregates are modeled as separate tasks *tOrderAggregate* and *tItemAggregate*, whereas there is only one task *tOrderItemAggregate* for $Aggr_s$. The aggregate tasks have entries for the request types defined, i.e. *eCreateOrder*, *eAddItem1* and *eAddItemN*. Entries *eValidateItem1* and *eValidateItemN* are modeled in *tOrderAggregate* task in $Aggr_m$ for validating the items. These entries contain: *aGetData* activity representing validation code that iterates through all properties of the request, *aReadEvents* reads events from event store, and *aWriteEvents* writes events to the event store. It is to be noted that the *eCreateOrder* in both *eAddItem1* and *eAddItemN* in $Aggr_m$ do not contain activity for event read as both entries create a new stream in the event store and hence do not have any events. For event replay, the *aReadEventN* activity's service demand is multiplied by F_{mult}. The number of calls are set as deterministic with a value of '1'.

Event Handlers manage event read and write between aggregates and the event store, which is modeled by the *tEventHandler* task

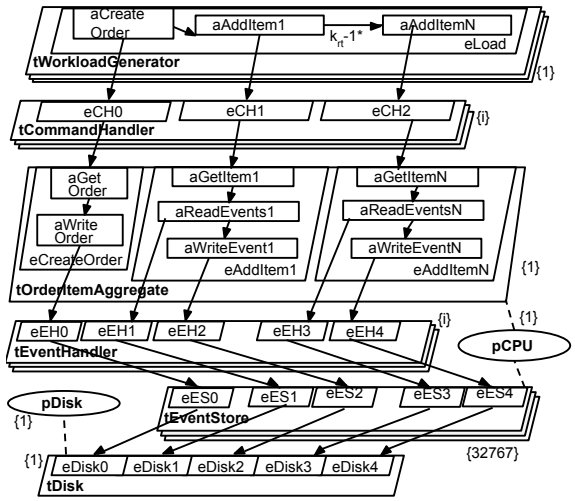

Figure 4: LQN Model for $Aggr_s$ Architectre with the Aggregate and the Event Store Represented as Single Tasks.

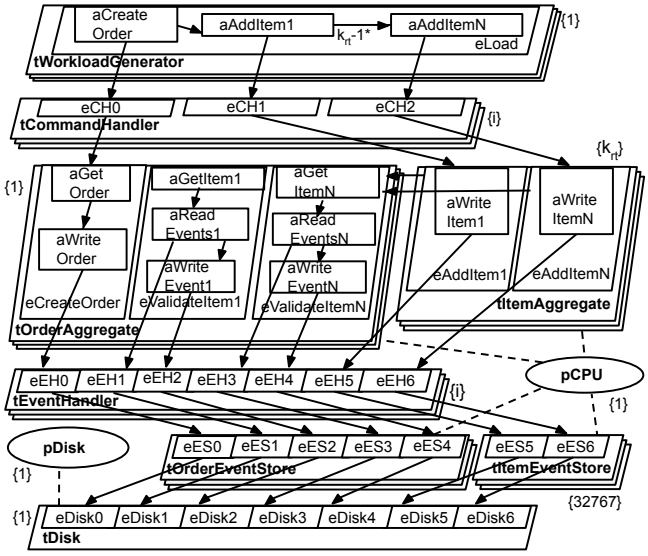

Figure 5: LQN Model for $Aggr_m$ Aggregate Architecture with the Aggregate and the Event Store Tasks Represented as Two Separate Tasks to Represent Separate Domain Boundary.

simulating the delay. It has a *m*=infinite and infinite processor. The event store is represented by an *tEventStore* for $Aggr_s$, and *tEventStoreOrder* and *tEventStoreItem* for $Aggr_m$. In the event store task/s for the request type *AddItemN*, two entries are modeled, one for read event (*eReadN* with service demand multiplied by F_{mult}) and one for write (*eWriteN*). The *tDisk* like event store task/s has entries for read and write entries with read using F_{mult}.

5.3 Service Demand Calculation

We use the Utilization Law and the Forced Flow Law, analogue to other works [4, 12, 13, 18]. Utilization Law states that for a request class, the demand at a station i in the network is $D_{c,i} = U_{c,i}/X_c$, where $U_{c,i}$ is the average utilisation at the station i and the X_c is the throughput of the entire system. Similarly, using Forced Flow Law, the number of visits at a station i is given by $V_{c,i} = X_{c,i}/X_c$, where $X_{c,i}$ is the throughput at i. Using the number of visits, $V_{c,i}$, the service demand can be calculated as $D_{c,i} = V_{c,i} * S_{c,i}$, where $S_{c,i}$ is service time per visit at the station i.

In order to measure the execution demand at the stations in the LQN model, measurements were run on the SUT with one user load for the three request types defined earlier in the LQN models. We ran only 5000 *CreateOrder* requests first, followed by 5000 *AddItem* requests on existing orders adding one item to each order, and finally ran another 5000 *AddItem* requests as second item to each order. At each stage, Performance Monitor utility on Windows was used to do the measurements. We used the following counters: CPU %idle (which also includes %iowait for Windows OS); %disk_time for read and write; %CPU_time, total tps, and write tps for SQL server instance. The SUT throughput and total execution time were obtained using the custom workload generator. Let us take an example for *CreateOrder* request on $Aggr_s$, where system overall CPU utilization is 0.238, event store CPU utilization 0.018, throughput is 246.8, event store total and write tps as 148.9 and 147.1, and request execution time is 3ms. By using the Utilization Law, we get the total SUT service demand as (0.238/246.8)*1000 = 0.964ms. This includes the service demands for aggregates, event store, and event handler. Again using the Utilization Law, Event Store's service demand on pCPU was calculated, i.e. 0.018/246.8=0.075ms. The Event Store read service demand is found by subtracting the proportion of write tps from the total tps, i.e. 0.075*[1-(147.1/148.9)]=0.0009ms. We subtract the event store service demand (0.075ms) by total pCPU demand to get demand for aggregate and event handler (0.887ms). From separate measurements, we calculate service demand for aGetOrder and aGetItem1/N as 0.1ms. To get disk service demand, either Utilization Law or Forced Flow Law can be used. Following the same example of *CreateOrder* request, the measured disk write tps is 152.1 and disk sec/write is 0.001sec. Using Forced Flow Law, the number of disk visits is calculated as 152.1/246.8=0.616. Multiplying the number of disk visits to disk sec/write, we get disk service demand for *CreateOrder* request as (0.616*0.001)*1000=0.616ms. Any delay is calculated as (3-0.964-0.616)=1.47ms as command handler delay. Similar calculations were used for the other two request types. The measurements and calculations were repeated several to get average value for the service demands. Table 1 shows the measured average service demands input to the model. It is to be noted that for $Aggr_m$ *eES5/6* and *eDisk5/6* demands are not specified as total event store and disk write is assigned to *eES2/4* and *eDisk2/4*

6 CASE RESULTS AND FINDINGS

The throughput and response time comparison of the simulation vs. measurement is shown in the Figure 6. Relative error for throughput excluding initial four points is 8% ($Aggr_s$) and 13%($Aggr_m$). Request response time error is 6% ($Aggr_s$) and 18%($Aggr_m$), which are acceptable according to [13] for model validation. An important point to be noted from the plots is that for smaller number of *AddItem* requests, Elastic Search reports the response times for individual requests as too high. When we consider the total system response

Table 1: Table of Service Demands in milliseconds for the Request Classes at the Stations Mentioned in Parentheses in LQN Model.

Request Class	Aggregate	Event Hdlr.	Command Hdlr.	DB Read	DB Write	Disk Read	Disk Write
CreateOrder ($Aggr_s$)	0.100 (aGetOrder)	0.930 (eEH0)	1.029 (eCH0)	-	0.063 (eES0)	-	0.818 (eDisk0)
AddItem1 ($Aggr_s$)	0.100 (aGetItem1)	2.178 (eEH1)	2.660 (eCH1)	0.003 (eES1)	0.187 (eES2)	0.537 (eDisk1)	8.318 (eDisk2)
AddItemN ($Aggr_s$)	0.100 * F_{mult} (aGetItemN)	1.908 (eEH3)	2.045 (eCH2)	0.0015 * F_{mult} (eES3)	0.180 (eES4)	0.911 * F_{mult} (eDisk3)	5.130 (eDisk4)
CreateOrder ($Aggr_m$)	0.100 (aGetOrder)	0.825 (eEH0)	1.087 (eCH0)	-	0.107 (eES0)	-	0.513 (eDisk0)
AddItem1 ($Aggr_m$)	0.100 (aGetItem1)	3.487 (eEH1)	2.506 (eCH1)	0.004 (eES1)	0.782 (eES2)	0.601 (eDisk1)	12.01 (eDisk2)
AddItemN ($Aggr_m$)	0.100 * F_{mult} (aGetItemN)	2.903 (eEH3)	2.048 (eCH2)	0.003 * F_{mult} (eES3)	0.641 (eES4)	0.923 * F_{mult} (eDisk3)	9.110 (eDisk4)

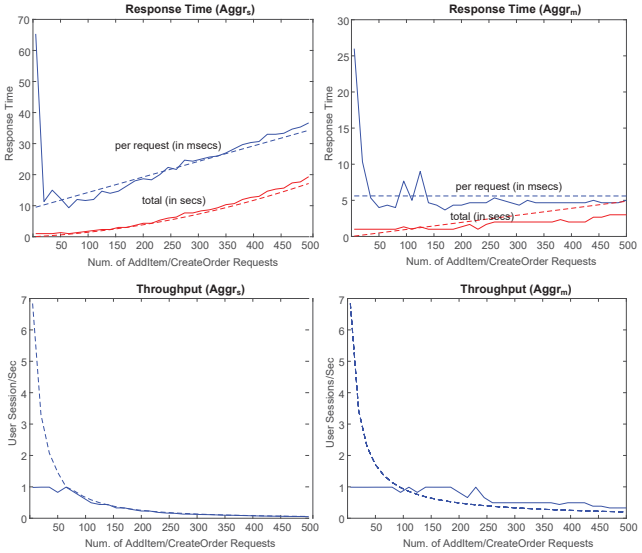

Figure 6: Plots of System Response Time for $Aggr_s$ (top left) and $Aggr_m$ (top right), and Throughput $Aggr_s$ (bottom left) and $Aggr_m$ (bottom right) in terms of User Sessions/Sec. The Simulation (dashed line) and Measurement (smooth line).

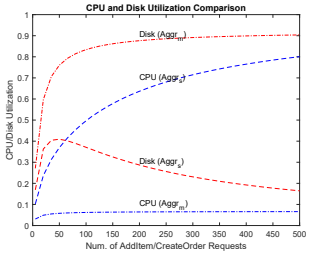

Figure 7: Plots of CPU (blue) and Disk (red) Utilization for $Aggr_s$ (dashed line) vs. $Aggr_s$ (dash-dotted line).

time by multiplying the individual request response time with the reported throughput, we see in the plots for response time in red lines that the total response time matches with predictions. This is reason why throughput values for initial points is low in throughput plots. This is an discrepancy reported by the monitoring tool.

From Figure 7, we can see that the *pCPU* and *pDisk* utilization scores for the two aggregate architectures have contrasting patterns. It can be seen that $Aggr_s$ utilizes more CPU cycles than the $Aggr_m$. This is because in $Aggr_m$ smaller events are handled, while more time is spent in disk IO. Contrary to CPU utilization, more disk utilization is observed in the $Aggr_m$ compared to $Aggr_s$ as the former has three times more events to write for every *AddItem* request processed, whereas in latter only one event is written. Since CPU is shared among several stations in the model, the response time increases as there is more waiting time for requests in $Aggr_s$ which leads to its steady rise.

6.1 Analysis of Findings

From the simulation and measurements on ES and CQRS based component-based software, one of the main observations is that *there is no one single architecture that performs best in all the scenarios that were envisioned in the research method.* Each architecture performs differently for different workloads.

While comparing the architectures, the following can be observed: in $Aggr_s$, CPU utilization keeps increasing with k_{rt}, whereas with $Aggr_m$ high disk utilization is observed. Similarly, we see that the throughput drops sharply in case of $Aggr_s$ dropping 10 times for k=5 to k=80 (o1), whereas as in $Aggr_m$ the throughput drop is gradual with 10 times drop from k_{rt}=5 to k_{rt}=290 (o2). Consequently, we also see that the per-request processing time is higher in $Aggr_s$ going up to 20 secs for the higher end of the k_{rt}, while in $Aggr_m$ the processing time is only a couple of seconds.

Based on the plots, the architects suggested - "It would better to go for $Aggr_m$ if there is no/less complex business logic or other communications between the ARs (as then there is no overhead and greater scalability). If there is complex business logic, $Aggr_s$ would be a better option". If there is both complex business logic and the need for scalability and greater response times, the following was mentioned - "It would be better to convince the designers to loosen the complex business logic. The application could be more forgiving and render the business logic as a warning after storing the input, instead of a blocking error before storing the input".

To summarize the findings, the guidelines that could be drawn from results shown in the graphs are:

(1) A separate aggregate architecture is better for scalability and response times, particularly if for a large number of child entities (o2).
(2) A separate aggregate architecture makes the life of programmers harder when they need to implement complex business logic (as seen from sequence diagram from separate aggregates being more complex).
(3) When 2. is in play, and 1. is not important, it would be advisable to go for a single aggregate architecture (o1).

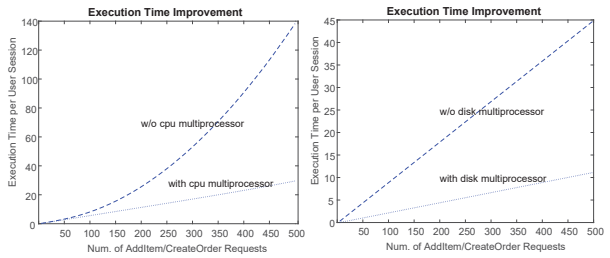

Figure 8: Plots of Total Execution Time (msecs) from Simulation for 10 User Load for w/o Multiprocessor for *pCPU* (dashed line) vs. with Multiprocessor (dotted line) for $Aggr_s$

(4) When 1. is a hard requirement, but 2. is also in play, trying to find other solutions for complex business logic is preferable such as to break up commands with a large number of properties into smaller commands as CPU contention in loading aggregate state is the bottleneck in Fig. 7.

From the observations and architects opinions,it can be concluded that the final choice depends on the business logic complexity, speed of execution, and workload patterns.

6.2 Tactics for Overcoming the Bottlenecks

Since the models were shown to have acceptable prediction accuracy, we can modify element parameters to improve performance.

In the $Aggr_s$ simulations, it is observed that the *pCPU* utilization of aggregate task increases from 8% to 72% for k=5 to k=500 while *pCPU* utilization of event store remains in and around 8%. This is an issue for large number of concurrent users where utilization starts from 17% at k=5, increases to 41% at k=35, and then drops to 16% for k=500. This means that the event store is mostly idle for larger k values. We ran the simulations with 10 users without and with multiple processors for *pCPU* with m=4. The execution time per user session is shown in Figure 8 (left). From the plots, we can see that response time with multiprocessor is drastically reduced as CPU bottleneck for concurrent users is alleviated.

For $Aggr_(m)$, *pCPU* utilization of stations is relatively low with the highest being at the event store at 6% (k=500). The event store's utilization itself goes from 29% to 96% for k=5 to k=500 and *pDisk* utilization is 27% to 90% for the same k range. This shows that the event store is waiting for disk IO as its CPU utilization is not high. Hence increasing the disk resource will improve performance. We simulated 10 concurrent users without and with multiprocessor for the disk with m=4. The Figure 8 (right) shows the response times per user session with and without a disk multiprocessor. With the introduction of a disk multiprocessor, the throughput of the disk task is improved even with the same utilization score for the event store task. This means that the disk, and hence the event store, is also processing more requests rather than waiting for disk.

7 RELATED WORK

Several research works have investigated the effects architectures and workload on resource utilization and performance. In [18], web application (MyBikeRoutes-OSM) performance modeling using LQN is performed. The authors compare the predictions to actual measurement results and also suggest improvements in terms of identifying the most utilized station in the model.

Tribastone [20] use LQN to model service-oriented systems. In [2], performance of CPU intensive containerized applications using LQN is studied. In [21], LQN is used to model a JavaEE component-based applications. The authors modeled ECPerf, which is a JavaEE benchmark for measuring the scalability and performance of J2EE applications. Urgaonkar et al. in [22] model multi-tier Internet applications where the number of tiers was arbitrary. The authors validated the approach on two applications and also suggested improvements to identified tiers by increasing the capacity of the tiers. In these works, the authors do not investigate the application architecture in terms of organization of features into components, but rather model the different stations of the system and suggest improvements. In contrast to this, we focus on the placement of the individual features of the overall application.

In [3, 17] investigate workload's effects in terms of interactions between software components and predicting the overall behavior of the system from individual components. Our work is different because the interaction between the components is modeled using events. Zúñiga-Prieto et al. [25] propose dynamic and automated reconfiguration of architecture in the cloud. The authors use meta-models and transformations to specify architecture patterns and reconfiguration to suit resource consumption. Our work is similar in that we have model-time component organization. Dynamic reconfiguration is seen as part of future work. Klock et al. [10], presents a workload based performance analysis on the query-side of the CQRS. The authors try to identify the optimum Microservice architecture to suit a workload. The authors look at the query-side of CQRS, while we look at run-time instantiation of components.

The authors in [1] investigate tools for architects to make architectural design decisions in large software projects. The solutions include reuse of past architectural decisions, alternate choices, the rationale behind a choice, and ask experts about decisions. In [23], the authors propose a framework for architects to evaluate quality attributes by utilizing software operation data in run-time termed as Architectural Intelligence. An e-Learning case-study is presented showing a framework for the development of information systems using process mining techniques. We propose a similar decision framework in architectural knowledge for a CQRS system, however, we define workload patterns and define hypothetical scenarios for workload, and use experts for architecture selection.

In several of the related works, researchers propose methods to capture resource requirements of component-based software systems at the individual component level [3, 17]. We use this principle, by defining architecture components that map to the workload at different levels. We then tested the component organization on different types of workloads and their effect on several performance metrics. An important distinction with the related works is that while many of these works consider workload intensity [10, 25], we consider in-depth workload patterns that help to map resource usage to different types of end-users of the software.

8 DISCUSSION AND CONCLUSION

The work in this paper was carried out on an enterprise software system that is under development at ERPComp. The system uses

component-based architecture, combining ES and CQRS. If we were to experiment on different hardware platforms, the results would vary in terms of absolute numbers, but the patterns observed from the tests would be similar as it is tied to architecture and how they run into bottlenecks (CPU vs disk resource). This was verified using analytical modeling using LQN. Additionally, the performance tests were conducted on a computer with a specific configuration and assumed to be a black box. During testing, we ensured that all processes were stopped, except for the application, the workload generator, and resource metrics logging. It is possible that OS processes could have introduced peaks at some stage in the measurements, but these points would be few and measurements were repeated several times and averaged out to smooth out the peaks. We only consider two features in the work, which helps to study the behavior of two possible aggregate architectures without complex modeling for larger systems. Studying larger systems with more features is seen as part of future work. In the future, we will do more complex modeling of aggregate and event store.

In this paper, a study of the effects of workload patterns on the performance of a component-based CQRS/ES based enterprise application is presented. We identified two main types of architecture, dependent on domain boundaries: the single aggregate architecture with all entities in one boundary, and the separate aggregate architecture with entities within their own boundaries. We evaluated how the two architectures would function in an industry case study. The LQN models were solved using analytical solvers and the results of simulation and measurements were compared to check the validity of the models. The relative error for throughput and response time were, 8% and 6% for single aggregate, and 13% and 18% for separate aggregates, which were within the acceptable error range as specified by Lazowska et al. [13].

We observe from both the simulations and measurements that the bottlenecks come from the definition of domain boundaries and this determines resource scaling and execution time. As the CPU resource is a bottleneck in the case of the single domain boundary, the throughput drops steeply and as a consequence, the response time steadily increases. Since with separate domain boundaries, the bottleneck is disk and it is not shared by several processes, the throughput drop is lower and response time is nearly constant. We proposed improvements to the single aggregate architecture to alleviate the CPU bottleneck by introducing a multiprocessor in the simulations, which resulted in a response time reduction. We also showed that multiprocessor for disk can improve response time for separate aggregate architecture.

In the near future, we will experiment with aggregate architectures for larger systems and we envision a self-adapting system that modifies the architecture dynamically to suit the workload.

REFERENCES

[1] Manoj Bhat, Klym Shumaiev, and Florian Matthes. 2017. Towards a framework for managing architectural design decisions. In *Proceedings of the 11th European Conference on Software Architecture: Companion Proceedings*. ACM, Canterbury, UK, 48–51.

[2] Emiliano Casalicchio. 2019. A study on performance measures for auto-scaling CPU-intensive containerized applications. *Cluster Computing* 22, 3 (2019), 995–1006.

[3] Merijn De Jonge, Johan Muskens, and Michel Chaudron. 2003. Scenario-based prediction of run-time resource consumption in component-based software systems. In *Proceedings: 6th ICSE Workshop on Component Based Software Engineering: Automated Reasoning and Prediction*. IEEE, Portland, Oregon, USA.

[4] Peter J Denning and Jeffrey P Buzen. 1978. The operational analysis of queueing network models. *Comput. Surveys* 10, 3 (1978), 225–261.

[5] Martin Fowler. 2005. Event sourcing. http://martinfowler.com/eaaDev/EventSourcing.html

[6] Greg Franks, Peter Maly, Murray Woodside, Dorina C. Petriu, and Alex Hubbard. 2005. Layered queueing network solver and simulator user manual. , 15–69 pages.

[7] Jan-Pieter Guelen, Slinger Jansen, and Jan-Martijn E. M. van der Werf. 2015. *Informed CQRS design with continuous performance testing*. Master's thesis. Dept. of Information and Computing Sciences, Utrecht University, Utrecht, the Netherlands.

[8] Holger Hermanns, Ulrich Herzog, and Joost-Pieter Katoen. 2002. Process algebra for performance evaluation. *Theoretical computer science* 274, 1-2 (2002), 43–87.

[9] Jaap Kabbedijk, Slinger Jansen, and Sjaak Brinkkemper. 2012. A case study of the variability consequences of the CQRS pattern in online business software. In *Proceedings of the 17th European Conference on Pattern Languages of Programs*. ACM, Irsee, Germany, 2:1–2:10.

[10] Sander Klock, Jan-Martijn EM Van Der Werf, Jan-Pieter Guelen, and Slinger Jansen. 2017. Workload-based clustering of coherent feature sets in microservice architectures. In *IEEE International Conference on Software Architecture (ICSA)*. IEEE, Gothenburg, Sweden, 11–20.

[11] Samuel Kounev and Alejandro Buchmann. 2003. Performance modelling of distributed e-business applications using queuing petri nets. In *2003 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, Austin, Texas, USA, 143–155.

[12] Samuel Kounev and Alejandro P Buchmann. 2003. Performance modeling and evaluation of large-scale J2EE applications. In *Proceedings of the Computer Measurement Group's International Conference (CMG)*, Vol. 11. Dallas, Texas, USA, 273–283.

[13] Edward D Lazowska, John Zahorjan, G Scott Graham, and Kenneth C Sevcik. 1984. *Quantitative system performance: computer system analysis using queuing network models*. Vol. 22. Prentice-Hall, Inc.

[14] Gururaj Maddodi, Slinger Jansen, and Rolf de Jong. 2018. Generating Workload for ERP Applications through End-User Organization Categorization using High Level Business Operation Data. In *International Conference on Performance Engineering*. ACM, Berlin, Germany, 200,210.

[15] Gianantonio Me, Giuseppe Procaccianti, and Patricia Lago. 2017. Challenges on the Relationship between Architectural Patterns and Quality Attributes. In *IEEE International Conference on Software Architecture (ICSA)*. IEEE, Gothenburg, Sweden, 141–144.

[16] Daniel A Menascé and Hassan Gomaa. 2000. A method for design and performance modeling of client/server systems. *IEEE transactions on software engineering* 26, 11 (2000), 1066–1085.

[17] Vibhu Saujanya Sharma, Pankaj Jalote, and Kishor S Trivedi. 2005. Evaluating performance attributes of layered software architecture. In *International Symposium on Component-Based Software Engineering*. Springer, Berlin, Heidelberg, 66–81.

[18] Y. Shoaib and O. Das. 2011. Web application performance modeling using layered queueing networks. *Electronic notes in theoretical computer science* 275 (2011), 123–142.

[19] Conie U Smith and Lloyd G Williams. 2002. *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*. Addison-Wesley Pearson Education.

[20] Mirco Tribastone, Philip Mayer, and Martin Wirsing. 2010. Performance prediction of service-oriented systems with layered queueing networks. In *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*. Springer, Berlin, Heidelberg, 51–65.

[21] Alexander Ufimtsev and Liam Murphy. 2006. Performance modeling of a JavaEE component application using layered queuing networks: revised approach and a case study. In *Proceedings of conference on Specification and verification of component-based systems*. ACM, Portland, Oregon, USA, 11–18.

[22] Bhuvan Urgaonkar, Giovanni Pacifici, Prashant Shenoy, Mike Spreitzer, and Asser Tantawi. 2007. Analytic modeling of multitier internet applications. *ACM Transactions on the Web (TWEB)* 1, 1 (2007), 2.

[23] Jan-Martijn E. M. van der Werf, Casper van Schuppen, Sjaak Brinkkemper, Slinger Jansen, Peter Boon, and Gert van der Plas. 2017. Architectural intelligence: a framework and application to e-learning. In *Evaluation and Modeling Methods for Systems Analysis and Development (EMMSAD)*. Springer, Essen, Germany, 95–102.

[24] Greg Young. 2010. CQRS and Event Sourcing. http://codebetter.com/gregyoung/2010/02/13/cqrs-and-event-sourcing

[25] Miguel Zúñiga-Prieto, Javier González-Huerta, Emilio Insfran, and Silvia Abrahão. 2018. Dynamic reconfiguration of cloud application architectures. *Software: Practice and Experience* 48, 2 (2018), 327–344.