Partitioning Microservices: A Domain Engineering Approach

Munezero Immaculée Josélyne
Makerere University
Kampala, Uganda
munejosy@gmail.com

Doreen Tuheirwe-Mukasa
Makerere University
Kampala, Uganda
tdmk4jc@gmail.com

Benjamin Kanagwa
Makerere University
Kampala, Uganda
bkanagwa@gmail.com

Joseph Balikuddembe
Makerere University
Kampala, Uganda
jbalikis@gmail.com

ABSTRACT

Architecture styles in the software world continue to evolve driven by the need to present easier and more appealing ways of designing and building software systems to meet stakeholder needs. One of the popular trends at the moment is microservices. Microservice architecture is gaining the market of software development architecture due to its capability to scale. It separates independent small services of a system to perform one business capability at a time. However, determining the right size of business capability that could be called a microservice is still a challenge. Current practices of partitioning microservice rely on personal practice within industry which is prone to bias by practitioners. Based on the ambiguity of determining the optimum size of a microservice, in this paper, we propose a conceptual methodology to partition a microservice based on domain engineering technique. Domain engineering identifies the information needed by a microservice, services needed for microservice functionality and provides description for workflows in the service. We demonstrate the usage of this methodology on the weather information dissemination domain as a confirmatory case study. We show how to split the weather information dissemination system sub-domain into different microservices that accomplish the weather information dissemination business capability.

KEYWORDS

Sizing microservice, DDD pattern, weather domain

ACM Reference Format:
Munezero Immaculée Josélyne, Doreen Tuheirwe-Mukasa, Benjamin Kanagwa, and Joseph Balikuddembe. 2018. Partitioning Microservices: A Domain Engineering Approach. In *SEIA '18: SEIA '18: Symposium on Software Engineering in Africa , May 27–28, 2018, Gothenburg, Sweden.* ACM , 7 pages. https://doi.org/10.1145/3195528.3195535

SEIA '18, May 27–28, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5719-7/18/05…$15.00
https://doi.org/10.1145/3195528.3195535

1 INTRODUCTION

Designing good software requires proper planning for understanding what the software is all about and determining the best architecture the software should rely on [29]. Microservice architecture is the new buzz word around software architecture patterns today. Microservices provide several advantages over monolithic systems. They include the ability to make rapid functional changes which contributes to achieving high integrity factors such as maintainability and scalability; continuous software delivery; and delivering software into production [16, 33]. Microservices are mainly used in the cloud to deploy large and medium applications as a set of small independent services that can be developed, tested, deployed, scaled, operated and upgraded independently [25]. Those services are partitioned in the way that they can register themselves, be discovered by other services, record their configuration, and be generally orchestrated in their deployment and update processes [34].

A major challenge is on how to introduce microservices and arrive at appropriate size [22]. The question to be answered is in establishing where component boundaries should lie [13].
Some suggestions have been proposed to this effect, including among others, aspects on if the microservice will be a user service, and therefore a decision made based on the tooling (with leaning towards the usage of lightweight tools); size being determined by the number of lines of code (with recommendations of not exceeding a couple thousand lines of code); and functionality in terms of the microservice accomplishing specifically only one task [33]. Authors in [22] propose partitioning services by use case. Other strategies are to partition by verbs, nouns, or resources, and the scaling cube [2]. According to Newman [23], independent services should focus service boundaries on business boundaries, so as to avoid the difficulties introduced when the service becomes too large. He also postulates a microservice as something that could be rewritten in two weeks, with proper alignment to team structures.

Size for a microservice is important, because the granularity of microservices influences the quality of service (QoS). This is because the granularity of the service is highly dependent on the appropriateness of service tailoring [27]. Even though in microservice architecture the design of small services is encouraged, too fine-grained services cause an ineffectively high amount of interactions necessary to fulfill a one request [23].

The challenges due to the ambiguity around the right size of a microservice and lack of good guidelines for designing a microservice

in terms of scope or size, encompass how to partition a microservice into the optimal tailored size, to ensure loose coupling, such that the service can easily be changed to keep up with business and technical demands. That not withstanding, the microservice approach must contend with some issues such as integration between communication of applications [33], and complexities that arise from creating a distributed system. These include testing, deployment and increased memory consumption [22].

Domain Driven Design (DDD) provides a number of useful patterns for dealing with the kind of complexity encountered in designing distributed systems and with large and complex domains, by breaking the domain into a series of bounded contexts [12]. In general, this method has been suggested for use when designing a microservice [26]. DDD provides a way to set a well designed microservice in terms of choosing the appropriate boundary. In this paper, we propose a detailed methodology that uses DDD patterns to partition microservices, by respecting microservice architecture design and characteristics. In addition we exemplify the proposed methodology on the weather domain.

The rest of this paper is organized as follows; Section 2 explores work done on microservice partitioning and the relationship of microservices to DDD pattern. Section 3 describes our approach of partitioning using the DDD pattern. Section 4 illustrates the use of our result on a weather case study. Section 5 concludes the paper.

2 RELATED WORK

Researchers continue to seek for solutions to microservice problems particularly concerning performance, deployment, cloud computing, and the general structure of the architecture itself including the boundary of microservices [4]. Microservices are a new research area, and some of the research concerning the size of microservice shows that the size of a microservice should be small and focused on achieving one functionality [35]. In [3], the decomposition of microservices is done by considering system requirements, security and scalability, however these are not the only factors that should be considered when decomposing microservices. Hassan et. al., [7] propose an ambient granularity which is the modeling concept that treats microservice boundaries as an adaptable first-class entity. Some researches propose a metric policy to measure the size of a microservice by counting the sum of resources and clients which are responsible for interactions between microservices or external services [5]. Therefore, decomposing an application into an appropriate microservice is a crucial task because of the ambiguity around how small it could be.

2.1 Partitioning Strategies

A number of strategies have been used by practitioners to determine the appropriate size for a microservice, and how to partition a microservice.

Line of code Some developers relate the size of a microservice to the number of lines of code (LOC). They recommend that a service should not exceed 10 to 100 LOC [30]. The idea behind counting LOC is to keep track of the lines of code a microservice should not exceed. Thus, fewer lines of microservice code increases the flexibility of scaling a microservice and eases the practice of altering or removing a microservice. However, the LOC strategy is not appropriate because microservices are built using different technology stacks, which differ on LOC. Also, secondary services differ on minimum LOC depending on the type of service. For example, process services that coordinate calls between multiple tasks can have 100 to 1000 LOC, and a data service can be implemented by 10 to approximately 100 LOC.

Deployment unit A microservice is defined as one kind of development and deployment unit. This service is mainly exhibited in the cloud as Infrastructure as a service (IaaS), Platform as a service (PaaS) and Security as a service (SaaS) to deploy a large application as a set of small services that can be deployed, tested, scaled, operated and updated independently. These services are partitioned in the way they register themselves, are discovered by other services and can be generally orchestrated in their deployment and update process [34].

Business capability Business capability defines what a system does in enabling an organization to successfully perform a unique business activity [17]. In agile development, it is an important way of combining data that have something in common such as functionality, rather than using collections of data entities that expose CRUD-style methods. This helps to understand the demand impact on application architecture at an early stage. In literature, microservices are built based on the need to address one business capability, or one business functionality at a time. However, developers face a challenge in using the business capability as a boundary for microservices; on defining which level of granularity a business capability should fit, so that it is not too small or too large. A small business capability leads a service to depend on other services, and requires service orchestration, or if too large will have the impact of turning the microservice into heavyweight SOA, with a lot of confusion and complexity [29]. Also, if the service is too big, by having coarse functionality, the benefits of the microservice architecture pattern, such as scalability, loose coupling and independent deployment, will not be seen [29].

Some have experienced the challenge of not partitioning a microservice correctly in its functionality. During design, if the microservice is too fine-grained there is a need for making orchestration from within user interface layer or in API layer. Moreover it implies doing inter-communication among services to process a single customer request which increases the complexity in design [29].

2.2 Domain Driven Design Pattern

Domain driven design enables inflexible architectures to create a large-scale structure across bounded contexts aimed at providing a better understanding of the high-level concepts, which is the core goal of microservices. Some research shows that knowing where to draw the boundaries is the key task when designing and defining a microservice [21].

The domain driven design (DDD) approach helps system developers to reduce the complexity of a business or of the domain by involving domain experts in the system development process [12]. The main goal of DDD is to systematically group all requirements in a realistic domain model and implement feasible code of that domain model [10].

The general structure of DDD presented in Figure 1 shows that the main goal of DDD is to design a domain model which represents the solution in all aspects. This encompasses all of the domain knowledge and the relationships of the different objects. A domain is composed of different subdomains in the given bounded context, with the bounded context applicable from ubiquitous language. However, it is not always obvious that each subdomain has only one bounded context. As Figure 1 shows, different bounded contexts can constitute a subdomain.

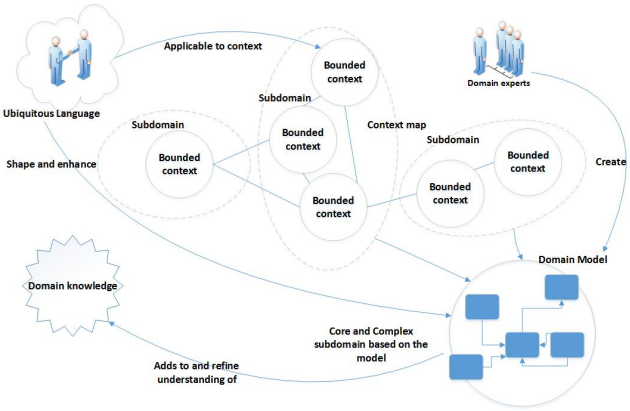

Figure 1: Domain Driven Design overview

There are patterns of domain driven design that can be used to size a microservice including; identifying bounded contexts, aggregate dependency injection, making concepts explicit, applying object oriented strategies, ubiquitous language, and separation of entities and value [6].

Context map and bounded contexts The context map is the tool used to make boundaries between domains explicit. A bounded context encapsulates the details of a single domain. A well designed bounded context in a context map will end up being a microservice. Hence, in microservice architecture, services should be organized around business capability by organizing cross-functional team(s) of experts to explicitly define the boundary of the domain. Also microservices should be autonomous and have the capability of making changes without affecting other services. In expecting this, the microservice should fit only one bounded context. This also implies the implementation of single database per service. If there is a need to scale up microservices to a high level, a bounded context may contain multiple microservices, but a microservice that is cross-cutting different bounded contexts should be avoided [11].

Aggregate The aggregate is a logical boundary for things that can change in a business transaction of a given context. To ensure the consistency among parallel operation changes, aggregates play a big role to coordinate activities that are considered as one unit in regard to change [12]. Since aggregates have a root, which is the reference of objects that compose the aggregate, it is advisable to choose the aggregate root as entity and other aggregate objects as value objects, because any change in the objects will not affect the integrity of the aggregate. Also the creation of aggregates should be atomic.

Repository In DDD, a repository has the responsibility of storing and retrieving data. The repository can be in any format: SQL server, Oracle, XML file, and so on [15]. The main goal and priority of the DDD approach is to design a domain model form, domain objects, identify objects relationship, their interaction, and find out if the business goals are achieved. This creates a persistence ignorance of a database in the application, which will make an application free from coupling with any data store [15].

Applying Object Oriented Object Oriented Programming (OOP) is suitable for implementing the domain model of DDD. OOP encapsulates conceptual connections between objects and classes, then makes mapping between the domain and code very easy [6]. In OOP UI, database, and other support code often gets written directly into the business objects which easily facilitate the implementation of bounded context to the extent of independent deployment of bounded context. Since microservice architecture has decentralized governance [20], microservices that compose a system can be built using different tools. This can be done using tools that follow OOP, because procedural languages are characterized by functional calls, which are burdensome in implementing the autonomy of bounded context.

Ubiquitous language This is a communication language expressed in speech, diagram, and writing by the domain expert, system architect and system developer. Its rationale is to share knowledge and straighten the model of the system [6]. Microservices that constitute a system need ubiquitous language to ensure that they are in line with the business perspective, by implementing the systems that use the same terms and same operations as the business do.

Separation of entities and value type Entities are necessary objects of DDD, but not all objects should be entities because this will cause performance problems, due to the increase of instances that have to be created for each object. Value objects are an alternative for those objects used to describe the domain aspect where identity is not necessary. Value objects have a significance in designing objects because these can be immutable and shared among objects.

Obtaining the right size of microservice that obeys all rules, can be achieved by using the DDD standard. DDD provides the information needed by a microservice, a way to identify the services needed and a description of the workflows for the identified services. This will aid in determining the appropriate size of a microservice. However, there are other factors that could influence the size of microservices, notably the way a microservice will interact with other microservices, known as the messaging route (AMQP, Kafka), using either the synchronous or asynchronous mechanisms; the way to publish and connect to event choice of API gateway protocol (REST/http); and the way to interact with the database. DDD approach offers the ability to separate a system into loosely coupled parts that handle a well known domain activity. Since microservices should be small, separately deployed units that are distributed in nature [24], there is no common size for all services, and the right size cannot be measured in number of lines of code. However, the size of a microservice should be in line with what the system should deliver [31], by properly separating bounded contexts to fit in the

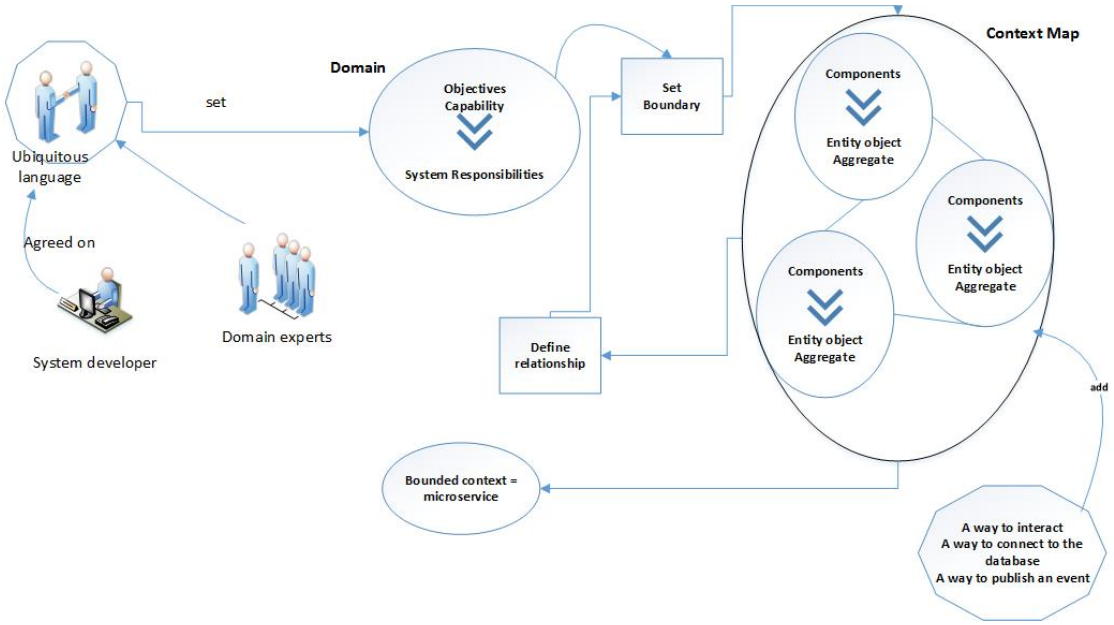

Figure 2: Microservice partitioning procedure overview

corresponding domain functionality.

3 DDD PROCEDURE OF PARTITIONING INTO MICROSERVICES

Based on the DDD pattern described in section 2.2, and an exploration study of research on how to design a good microservice, we take the steps illustrated in the figure 2 to design the right size:

- Having a well defined domain, start by indicating all the responsibilities of the system. Using ubiquitous language, define what the system does, and the different domain functionalities it should implement. Note that if the domain is wide it is advisable to split it into sub-domains.
- Find the boundary of each responsibility and make it as a business capability. Each business capability is a microservice. In the process of setting the boundary, the focus should be on the relationships among different microservices.
- For each domain responsibility, break down coarse grained responsibility.
- Define the microservice components, then the objects of a component, design the objects of an aggregate to be included in one component. Avoid aggregates whose objects are cross-cutting different components or microservices. Design entity objects to be aggregate roots and value objects to be aggregates. If it is required that an object is to be used in different components or microservices, this should be a value object.
- Analyze relationships of the microservice, which must be as independent as possible. In case there is a distributed transaction across multiple microservices, this design could be avoided by setting the boundary and minimizing cross-cutting transactions. Moreover, if one microservice is responsible for more than one responsibility, its components have to be reviewed to form more than one microservice.
- Design a microservice that has as minimum operation as possible.

The partitioning procedure of sizing microservices explained is the first step to designing a successful microservice. There is however the need to still design techniques on the messaging format, protection of the service, API gateway, monitoring, repository and different ways to find the service.

The figure 2 illustrates the domain driven microservice partitioning procedure which begins with people involved in the system. These include the system developers, stakeholders and domain experts who agree on a ubiquitous language, which sets the system responsibility. System responsibility composes a domain according to the objectives and capabilities of the system. The boundary of the domain is then set. Coarse system capability is broken down into different components in respect to its context map, in the same context map study well how components are going to interact with each other in loose coupling possible. With context map that can store data in its own database, play a big role in establishing good design of a microservice. However, if the partitioning of a microservice results into more coupling architecture, it is revisable to set again relationship and set the context map again. In other case it means that microservices are not a good choice for all user cases, in some scenarios, other software architecture designs may result in better performance than microservices [32, 34].

4 ILLUSTRATION

We apply the strategies of section 3 to demonstrate how to possibly design a microservice for a fresh domain. We exemplify this using the weather information domain. In this section, we illustrate how to partition the weather information system into appropriate microservices. We begin by defining the weather domain so as to provide a vocabulary (ubiquitous language) for the domain, and then apply the procedure of partitioning microservices.

Weather is one of the factors that affects the livelihood of people everywhere in one way or another. Accessibility to reliable weather information is therefore vital for informed decision-making in various socio-economic sectors such as agriculture, disaster management, aviation, fishing, energy, mining, construction, defense, water resources and health, among others [1]. However, access to reliable and timely weather information remains a challenge for the African context, particularly considering Uganda [18]. Weather services are therefore critical for the activities in these sectors to support sector-related activities of stakeholders.

For instance, stakeholders from the agricultural sector are farmers; either crop production farmers, livestock farmers or fishermen. These farmers are interested in making decisions that will boost and maximize their productivity. Considering crop production farmers, these decisions will range from when to prepare land for tilling, when to plant, what type of crop to plant, depending on the weather season, all the way to harvesting and post-harvest matters such as storage, and market pricing, among others. For this type of farmer, therefore, weather information that is readily accessible and timely is a crucial service.

We elicited this domain knowledge by interaction with domain experts. Our domain experts included the weather information service providers, and the weather information consuming stakeholders, who were farmers in Uganda. The interaction was conducted in the form of guided questionnaires with the service providers, and focus group discussion sessions with the farmers. The results from the interviews were transcribed as narratives.

4.1 Weather Domain

Take the weather domain to be a well-defined domain, with weather described as the present conditions of the elements that compose the climate, and their variations over short periods [19]. Weather is characterized by parameters such as temperature, humidity, rain, wind direction and speed, atmospheric pressure, etc [18].

To achieve the objective of access to reliable weather information by stakeholders, weather information has to be collected, archived, processed and disseminated to relevant stakeholders. Weather information refers to weather data that characterizes the weather (weather parameters) or weather forecast (a prediction of the state of the atmosphere for a given location and time or time interval). A weather information system will therefore be necessary to provide weather information that is vital for informed decision-making by various stakeholders. This information is necessary for increased productivity (in the agricultural, energy, water resources and construction sectors) and safety (in the aviation, disaster management, fishing, health, mining, and defense sectors) [28].

The weather domain is a wide domain, and therefore, to achieve the objective of improving weather information management, the domain is split into four main sub-domains. Collectively the functionality in each sub-domain will contribute to the overall objective of improving weather information management. We identify the weather data collection; weather data archiving; weather forecasting/prediction; and weather information dissemination subdomains, as shown in Figure 3. The weather system should implement this functionality. We are particularly interested in the weather information dissemination sub-domain, which we describe further.

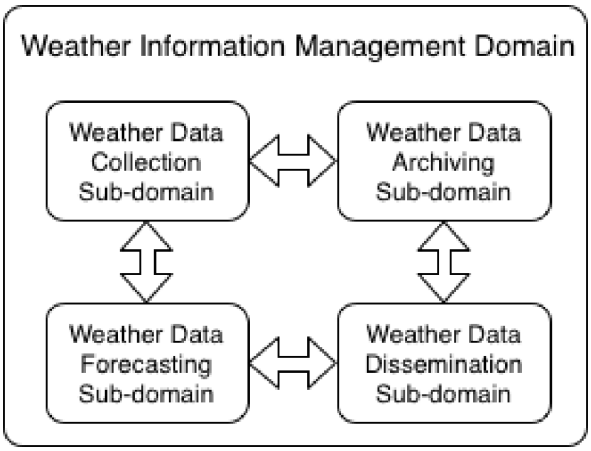

Figure 3: Weather Domain and Sub-domains

Weather Information Dissemination After weather data is processed and a forecast produced, the weather forecast information is disseminated to relevant stakeholders. An efficient dissemination system should allow easy and timely access of weather information. This information is to enable stakeholders make informed decisions for improved productivity and safety related to their sector activities.

The sub-domains in Figure 3 describe the boundaries and therefore the business capability of each aspect of the weather information management system. Each sub-domain/business capability is focused on doing one thing. Taking the business capability as a microservice, we can clearly see the relationship between the different microservices. Weather data is firstly collected, then archived, then processed and finally disseminated. Decomposing the weather domain into sub-domains adheres with the principles of componentization, collaboration, reliable connections and controls, which are key to successful microservices [14]. This componentization also allows for making of rapid and independent functional changes between the sub-domains. Each component can be maintained and scaled independently.

4.2 Partitioning into Microservice Components

We have generally described the weather information dissemination domain as the domain on which we apply the procedure of partitioning into microservices. The weather information dissemination system's main functionality is to provide timely and easy

access of weather information to stakeholders to enhance their decision-making process.

The weather forecast information prepared by the weather information service provider is disseminated to stakeholders using various channels including mass media (such as TV and radio), print media (such as newspapers and magazines), via Internet (email, websites, social media), word-of-mouth, among others. As a part of this domain, we identify and define the core concepts that should be implemented as part of the domain functionality. These concepts are categorized as entities which may have values, properties and types, or functions over entities or events and behaviors of the entities [8]. We abstract these concepts as a result of interaction with domain experts. Concepts in this case represent knowledge about objects with certain properties, and characterize a domain [9]. Through brainstorming, sketching rough descriptions of the domain and informal analysis of the transactions of the domain, we abstract the following entities and their vocabulary.

Weather Service A weather service is a composition of different weather information.

Weather Information This can either be raw, processed or predictive. For instance, weather information in Uganda, may be in the form of alerts (warnings), dekadals (ten-day forecast) or seasonal (three months forecast).

Stakeholder This includes both the weather service provider and weather information consumer. Examples of consumers include the stakeholders from the different sectors.

Dissemination Channel Medium through which weather information is conveyed to various stakeholders. Main channels are mass media, Internet, mobile telephones, Non-Governmental Organizations (NGOs) and government ministries, departments and agencies, among others.

Decision Refers to what the action the stakeholder will make on processing provided weather information. Considering crop production farmers, possible decisions may be when to plant, what type of crop to plant, when to weed, or harvest.

Format Form of presentation of weather information for dissemination. Possible formats are audio, text, graphic, animation. Aspects include language to be used.

The unique business activity of this component is weather information dissemination as shown in Figure 4. The objects are organized around this business capability, each representing a possible boundary. Each of them presents a business capability and can be implemented as a microservice. The microservices are dependent on a database for their information. The services have the ability to register themselves, record their configuration and be orchestrated in the domain's transactions.

4.3 Relationships and Operations

We identify some relationships among the objects in the microservices (Figure 4) and use natural language to represent them. They include:

- Weather Information *forms* weather services
- Weather Information *informs* decisions
- Weather Information *has* format
- Weather Services *support* stakeholder

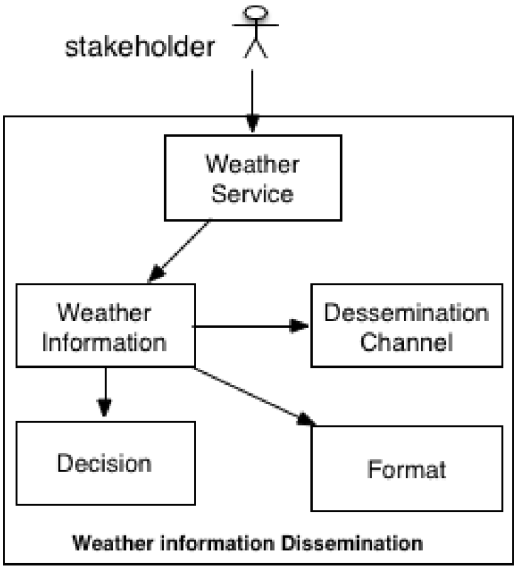

Figure 4: Weather Information Dissemination Microservices

- Stakeholder *provides/consumes* weather information (depending on if they are providing weather information or consuming the weather information)
- Stakeholder *makes* decision
- Dissemination channel *conveys* weather information

All of the identified relationships are independent and particular to the weather dissemination microservice. They demonstrate the interactions between the different entities. The operation within this microservice is minimal. Examples of operations include:

- *Weather service = Weather Information x Stakeholder x Dissemination Channel x Format x Location*
  A weather service is structured by taking weather information for a particular stakeholder, this information being provided through a specific channel and in a specified format. The weather information can be specific or general weather parameters. Language is a part of this information, the information may be tailored to a specific language
- *Dissemination Channel = Stakeholder x Format*
  the dissemination channel can be appropriately chosen by considering the stakeholder, and the format of message to be relayed
- *Gregorian-Time-Point = Year x Month x Day x Hour x {00,15,30,45}*
  for 15 min intervals

Note: The notation '*x*' used for the operations implies a combination of an element (as a Cartesian product) from each set.

DDD breaks down a domain into a series of bounded contexts, each of which, if autonomous constitutes a microservice. For each of the identified services, a single database can be implemented.

5 CONCLUSION

In this paper, we provide a method for partitioning microservices following the DDD pattern. In particular, we have demonstrated

this method on the weather information dissemination domain. The domain is decomposed into sub-domains and objects to make microservices that perform one functionality at a time. Sizing microservices in this way allows for the creation of small and independent components, which have few operations that can be easily altered if possible. We posit that the system designed in this way is able to make rapid functional changes, which domain engineering supports, and are crucial for microservices. DDD breaks down a domain into a series of bounded contexts, each of which, if autonomous, constitutes a microservice. For each of the identified services, a single database can be implemented.

As a result of the weather information dissemination service, the stakeholder is able to receive weather information that enhances their decision-making. The stakeholder can choose to receive raw, processed or predictive weather information. On the one mode, the stakeholder initiates a request for weather information through a medium such as a mobile phone, or Internet platform (web-portal, social media) and receives a message entailing their interested information. In this mode, the weather information is tailored to suit the requirements of the stakeholder, creating a demand-driven service. In the second mode, the stakeholder receives general weather information that is provided by the weather service providers. This information is pushed to the stakeholder over mass media (TV, radio, print media). This mode supports the PUSH mechanism where the stakeholder simply receives predetermined weather information. With the information, the stakeholder is then able to make an informed decision on how to proceed to carry out their economic activity.

Next steps are to implement the identified components as microservices, focusing on loose coupling to enable changes to be made independently, interaction with other services and allowing easy scale up through a combination of multiple microservices. Future work will entail an evaluation of the approach to provide validation.

REFERENCES

[1] [n. d.]. WIMEA-ICT. http://wimea.mak.ac.ug/. ([n. d.]). Accessed: 2017-04-15.

[2] Martin L Abbott and Michael T Fisher. 2009. *The art of scalability: Scalable web architecture, processes, and organizations for the modern enterprise.* Pearson Education.

[3] Mohsen Ahmadvand and Amjad Ibrahim. 2016. Requirements Reconciliation for Scalable and Secure Microservice (De) composition. In *IEEE 24th International Requirements Engineering Conference Workshop*. 6. https://doi.org/10.1109/REW.2016.14

[4] Ali Nour Evans Roger Alshuqayran Nuha. 2016. A Systematic Mapping Study in Microservice Architecture. *IEEE 9th International conference on Service Oriented Computing and Application* (2016), 8. https://doi.org/10.1109/SOCA.2016.15

[5] Tugrul Asik. 2017. Policy Enforcement upon Software Based on Microservice Architecture. *IEEE computer society* (2017), 283–287.

[6] Abel Avram. 2006. *Domain-Driven Design Quickly* (first edit ed.). InfoQ.

[7] Rami bahsoon Hassan Sara, Ali Nour. 2017. Microservice Ambients : An Architectural Meta-modelling Approach for Microservice Granularity. In *2017 IEEE International Conference on Software Architecture*. 1–10. https://doi.org/10.1109/ICSA.2017.32

[8] Dines Bjørner. 2006. *Software Engineering 3: Domains, requirements, and software design.* Springer Science & Business Media.

[9] Krzysztof Czarnecki, Ulrich W Eisenecker, G Goos, J Hartmanis, and J van Leeuwen. 2000. Generative programming. *Edited by G. Goos, J. Hartmanis, and J. van Leeuwen* 15 (2000).

[10] Nguyen Viet-ha Duc-Minh Dang. 2016. Domain-Driven Design Patterns : A Metadata-Based Approach. *IEEE International Conference on Computing and Communication Technology, Research Innovation, and Vision for the Future* 102 (2016), 247–252.

[11] Eberhard Wolff. 2016. *Microservices: Flexible Software Architecture* (1st editio ed.). InformIT. http://www.informit.com/articles/article.aspx?p=2738465

[12] Eric Evans. 2003. *Domain-Driven Design* (2nd editio ed.). Vol. 7873. Addison Wesley.

[13] Martin Fowler and James Lewis. 2014. Microservices. *Viittattu* 28 (2014), 2015.

[14] Owen Garrett. [n. d.]. Three keys to successful microservices. http://www.infoworld.com/article/2936148/application-development/three-keys-to-successful-microservices.html. ([n. d.]).

[15] Mahmud Hasan. [n. d.]. Domain Driven Design - Clear Your Concepts Before You Start - CodeProject. ([n. d.]). https://www.codeproject.com/articles/339725/domain-driven-design-clear-your-concepts-before-yo Accessed: 2017-07-15.

[16] Peter Jarman. 2015. Microservices âĂŞ A New Application Paradigm. *Infosys* (2015).

[17] Ulrich Kalex. [n. d.]. Business Capability Management: Your Key to the Business Board Room. http://www.opengroup.org/johannesburg2011/Ulrich%20Kalex%20-%20Business%20Capability%20Management.pdf. ([n. d.]).

[18] B Kanagwa, D Tuheirwe-Mukasa, and K Muwembe. 2015. The Need for An Integrated Effective Weather Dissemination System for Uganda. In *Proceedings of the 2015 International Conference on Frontiers in Education: Computer Science and Computer Engineering*. CSREA Press, 330.

[19] Simone Leao. 2014. Mapping 100 Years of Thornthwaite Moisture Index: Impact of Climate Change in Victoria, Australia. *Geographical Research* 52, 3 (2014), 309–327.

[20] Martin Fowler Lewis James. 2014. Microservices. (2014). file:///G:/Newfolder/micro2/webfile/Microservices.html

[21] Eisele Markus. 2016. *Developing Reactive Microservices* (1st editio ed.). O'Reilly Media Inc.

[22] Dmitry Namiot and Manfred Sneps-Sneppe. 2014. On micro-services architecture. *International Journal of Open Information Technologies* 2, 9 (2014).

[23] Sam Newman. 2015. *Building Microservices.* " O'Reilly Media, Inc.".

[24] Sam Newman. 2015. Microservices. In *Building Microservices*. O'Reilly, 1–11.

[25] Srikanta Patanjali, Benjamin Truninger, Piyush Harsh, and Thomas Michael Bohnert. 2015. CYCLOPS : A Micro Service based approach for dynamic Rating , Charging & Billing for cloud. (2015). https://doi.org/10.1109/ConTEL.2015.7231226

[26] Roland Petrasch. 2017. Model-based Engineering for Microservice Architectures using Enterprise Integration Patterns for inter-service Communication. *IEEE* (2017), 5–8.

[27] Florian Rademacher, Sabine Sachweh, and Z Albert. 2017. Differences Between Model-driven Development of Service-oriented and Microservice Architecture. In *IEEE international conference on Software Architecture Workshop*. https://doi.org/10.1109/ICSAW.2017.32

[28] Joachim Reuder and Julianne Sansa-Otim. 2013. WIMEA-ICT: Improving Weather Information Management in East Africa for effective service provision through the application of suitable ICTs. (November 2013).

[29] Mark Richards. 2015. *Software Architecture Patterns* (first edit ed.). O'Reilly Media Inc.

[30] Gerald Schermann, Jürgen Cito, and Philipp Leitner. 2015. All the Services Large and Micro: Revisiting Industrial Practice in Services Computing. In *International Conference on Service-Oriented Computing*. Springer, 36–47.

[31] Carlos M Ferreira Shahir Daya Nguyen Van Duy, Kameswara Eati. 2015. *Microservices from Theory to Practice Creating Applications in IBM Bluemix Using the Microservices Approach* (first edit ed.). International Technical Support Organization. 170 pages.

[32] Rion Dooley Stubbs Joe, Walter Moreira. 2015. Distributed Systems of Microservices Using Docker and Serfnode. *7th International Workshop on Science Gateways* (2015). https://doi.org/10.1109/IWSG.2015.16

[33] Johannes Thönes. 2015. Microservices. *IEEE Software* 32, 1 (2015), 116–116.

[34] Mario Villamizar, Oscar Garces, Lina Ochoa, Harold Castro, Lorena Salamanca, Mauricio Verano, Rubby Casallas, Santiago Gil, Carlos Valencia, Angee Zambrano, and Mery Lang. 2016. Infrastructure Cost Comparison of Running Web Applications in the Cloud Using AWS Lambda and Monolithic and Microservice Architectures. In *Proceedings - 2016 16th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing, CCGrid 2016*. 179–182. https://doi.org/10.1109/CCGrid.2016.37

[35] Eberhard Wolff. [n. d.]. What Are Microservices _ 3. ([n. d.]). http://www.informit.com/articles/article.aspx?p=2738465 Accessed: 2017-06-15.