UNIVERSITAS INDONESIA
Veritas, Probitas, Iustitia

FAKULTAS ILMU KOMPUTER

Module 8: Software Architectures

Advanced Programming

Compiled and chosen by: Ade Azurat
Editor: Teaching Team (Lecturer and Teaching Assistant)
Advanced Programming
Semester Genap 2023/2024
Fasilkom UI

Ade Azurat
Any question, feedback and critics can be sent by email to: ade@cs.ui.ac.id

Table of Contents

Learning Objectives

1. Students understand the importance of software architectures.
2. Students understand the microservice architecture
3. Students understand the event-driven architecture style
4. Students are able to implement an architecture in their project.
5. Students are able to design and draw an architecture based on specific needs.

References

1. Chapter 1,2,14,17,20, 21: *Mark Richards and Neal Ford,* Fundamental of Software Architectures: An engineering approach, O'Reilly, 2021.
2. Chapter 1: *Adam Bellemare*, Building Event-Driven Microservices, O'Reilly, 2020
3. https://medium.com/geekculture/event-driven-programming-with-rust-and-rabbitmq-using-crosstown-bus-c39c50ce6c98

1. Motivation

We need architect!

- We need architect for building a great building!
- What a about building a great software? Do we need architect?
- What is software architect? (person)
- What is software architecture? (concept)

FACULTY OF COMPUTER SCIENCE

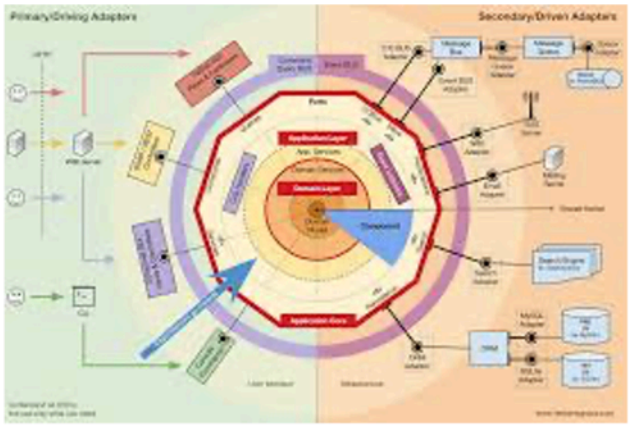
We need an architect for building great building. We may need only contractor to build simple building. The same thing with software. If it is a very big or a great software, we may need software architect to build it, while if it is just a simple application, we may only need a few programmer only.
The software architect is an important person in enterprise system or in social media where it usually handles a big case either in the size of user or in the size of transaction. Ignoring this need may end up in an application which is not appropriate to use or not appropriate for continuous changes which is a must is nowadays software engineering.
But what is a software architect? It is quite know as a prospective career and a well paid one. How would i can be one. Before going to answer it, we need to know first, what is software architecture. How software architectures relates with software development? Do they only focus on requirement? Or They involves even deeper into implementation?
These are the things we will study. In short, It is actually an advanced level of a programmer who also understand the business issue in software development. Let's start to go deeper.

Introduction to Software Architecture

Chapter 1,2, Mark Richards and Neal Ford, Fundamental of Software Architectures: An engineering approach, O'Reilly, 2021

What is software architecture?

- The industry doesn't have a good definition of software architecture
- Many developments in computer science are derived from industry development
- The need for software architecture is clear!
- It is to hire a smart person in software development!



FACULTY OF COMPUTER SCIENCE

How to become a software architect?

- Some job position has a clear carrier path:
  - Lieutenant -> Captain -> Major -> Colonel -> General
  - Cost Analyst -> Cash Manager -> Treasurer -> CFO

- But, not in software architect! why?
  - No clear definition of software architecture
  - The role embodies massive amount and scope of responsibility that continues to expand
  - The rapidly evolving of software development ecosystem
  - Absolute very quickly

FACULTY OF COMPUTER SCIENCE

The job "software architect" appears near the top of numerous lists of best jobs across the world. Yet when readers look at the other jobs on those lists (like nurse practitioner or finance manager), there's a clear career path for them. Why is there no path for software architects?
First, the industry doesn't have a good definition of software architecture itself. When we teach foundational classes, students often ask for a concise definition of what a software architect does, and we have adamantly refused to give one. And we're not the only ones. In his famous whitepaper "Who Needs an Architect?" Martin Fowler famously refused to try to define it, instead falling back on the famous quote:
Architecture is about the important stuff…whatever that is. --Ralph Johnson

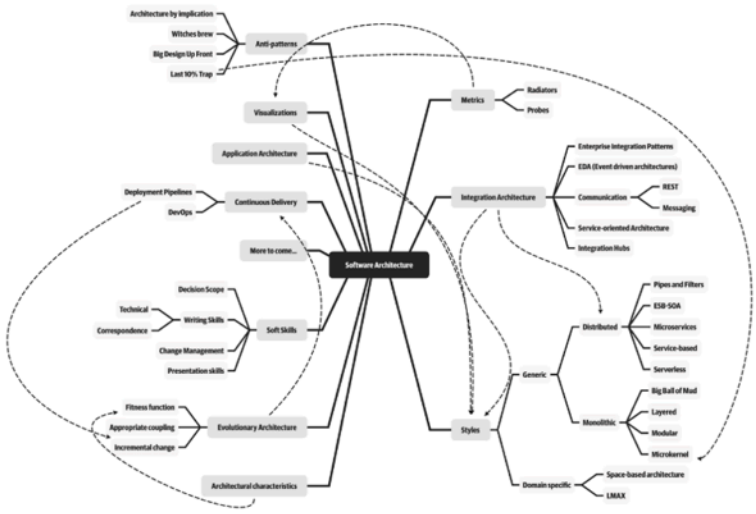


Figure 1-1. The responsibilities of a software architect encompass technical abilities, soft skills, operational awareness, and a host of others

When pressed, we created the mind map shown in Figure 1-1, which is woefully incomplete but indicative of the scope of software architecture. We will, in fact, offer our definition of software architecture shortly.

Second, as illustrated in the mind map, the role of software architect embodies a massive amount and scope of responsibility that continues to expand. A decade ago, software architects dealt only with the purely technical aspects of architecture, like modularity, components, and patterns. Since then, because of new architectural styles that leverage a wider swath of capabilities (like microservices), the role of software architect has expanded.

Understood in context

- Software Architecture must be studied in the context!
- For example, before 2000an, proposing a microservice architecture will be very expansive. The ideas will be rejected right away. Because at that time the current technology is still mainframe and data-center.
- With the emerging of cloud computing, it bring a lot of changes.
- Not to mention now a days with the emerging of social media application!

When studying architecture, readers must keep in mind that, like much art, it can only be understood in context. Many of the decisions architects made were based on realities of the environment they found themselves in. For example, one of the major goals of late 20th-century architecture included making the most efficient use of shared resources, because all the infrastructure at the time was expensive and commercial: operating systems, application servers, database servers, and so on. Imagine strolling into a 2002 data center and telling the head of operations "Hey, I have a great idea for a revolutionary style of architecture, where each service runs on its own isolated machinery, with its own dedicated database (describing what we now know as microservices). So, that means I'll need 50 licenses for Windows, another 30 application server licenses, and at least 50 database server licenses." In 2002, trying to build an architecture like microservices would be inconceivably expensive. Yet, with the advent of open source during the intervening years, coupled with updated engineering practices via the DevOps revolution, we can reasonably build an architecture as described. Readers should keep in mind that all architectures are a product of their context.

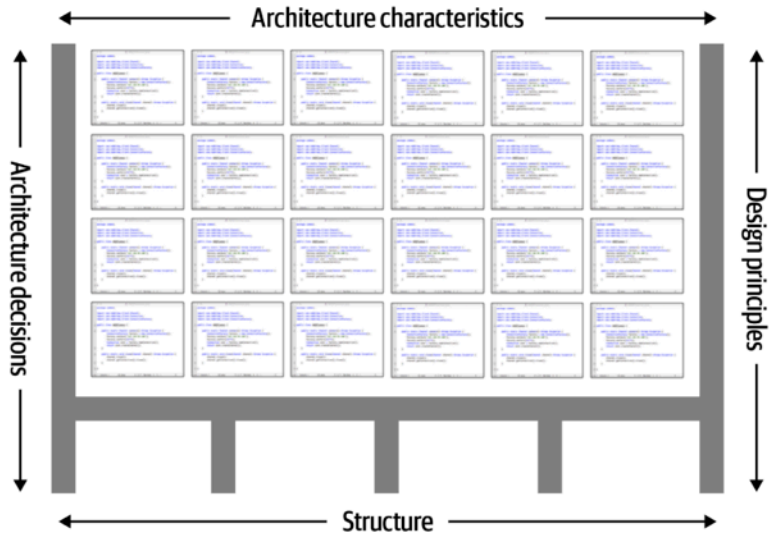
Defining software architecture

- We still have to try to define it and have a clear target to learn!
- Some architects refer to software architecture:
  - as the blue print of the system
  - as the roadmap for developing a system.
- The common thing of both:
  - How the architect analyzed the system before building the software architecture (in either reference)

FACULTY OF COMPUTER SCIENCE

The industry as a whole, has struggled to precisely define "software architecture." Some architects refer to software architecture as the blueprint of the system, while others define it as the roadmap for developing a system. The issue with these common definitions is understanding what the blueprint or roadmap contains. For example, what is analyzed when an architect analyzes an architecture?

Defining software architecture



Architecture consists of the structure combined with architecture characteristics ("-ilities"), architecture decisions, and design principles

The Figure illustrates a way to think about software architecture. In this definition, software architecture consists of the structure of the system (denoted as the heavy black lines supporting the architecture), combined with architecture characteristics ("-ilities") the system must support, architecture decisions, and finally design principles.

The Structure

- A microservice architecture is not actually really describe the whole architecture.
- It only describe the structure of the system.

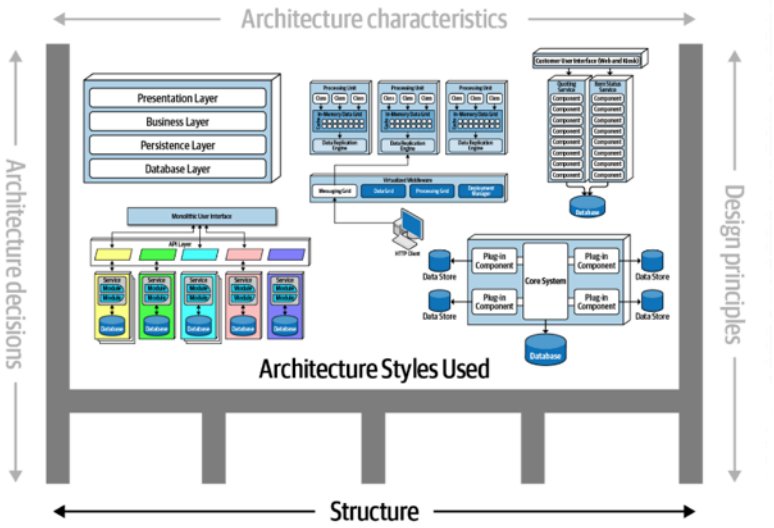


Figure 1-3. Structure refers to the type of architecture styles used in the system

The structure of the system, as illustrated in Figure 1-3, refers to the type of architecture style (or styles) the system is implemented in (such as microservices, layered, or microkernel). Describing an architecture solely by the structure does not wholly elucidate an architecture. For example, suppose an architect is asked to describe an architecture, and that architect responds "it's a microservices architecture." Here, the architect is only talking about the structure of the system, but not the architecture of the system. Knowledge of the architecture characteristics, architecture decisions, and design principles is also needed to fully understand the architecture of the system.

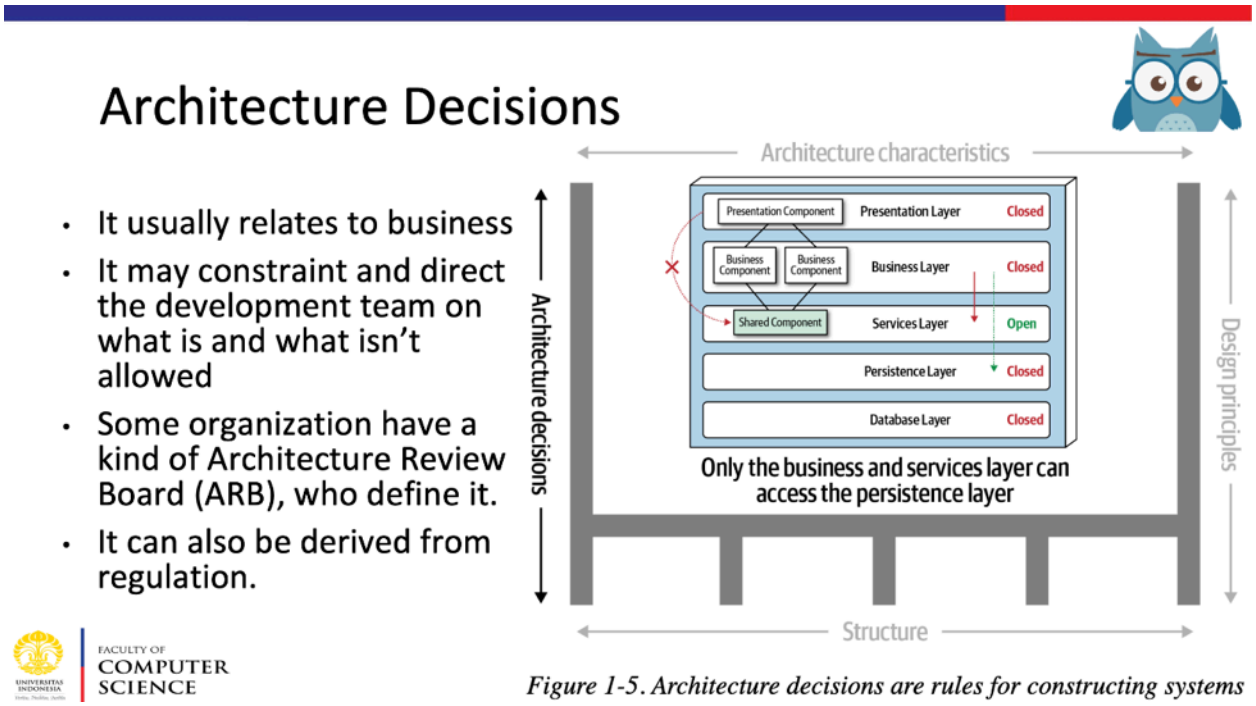


Figure 1-5. Architecture decisions are rules for constructing systems

The next factor that defines software architecture is architecture decisions. Architecture decisions define the rules for how a system should be constructed. For example, an architect might make an architecture decision that only the business and services layers within a layered architecture can access the database (see Figure 1-5), restricting the presentation layer from making direct database calls. Architecture decisions form the constraints of the system and direct the development teams on what is and what isn't allowed.

If a particular architecture decision cannot be implemented in one part of the system due to some condition or other constraint, that decision (or rule) can be broken through something called a variance. Most organizations have variance models that are used by an architecture review board (ARB) or chief architect. Those models formalize the process for seeking a variance to a particular standard or architecture decision. An exception to a particular architecture decision is analyzed by the ARB (or chief architect if no ARB exists) and is either approved or denied based on justifications and trade-offs.

Design Principles

- It is a system design principles to differentiate it from OO Design principles
- It is a generic guidelines of designing a system.
- For example, suggest the use of gRPC for communication protocol.

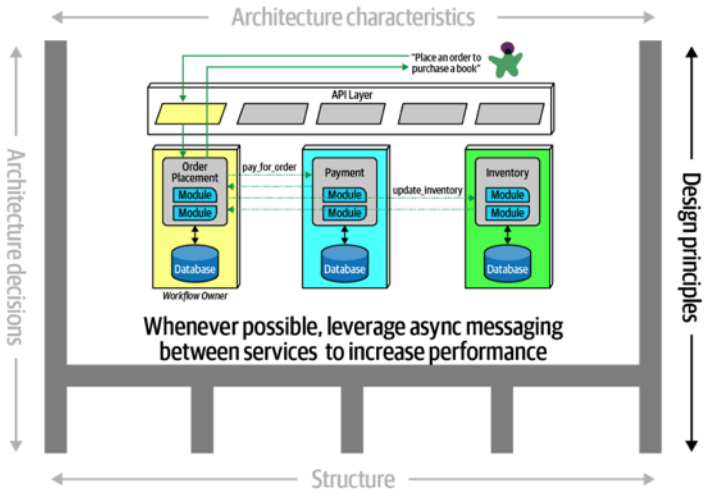


Figure 1-6. Design principles are guidelines for constructing systems

The last factor in the definition of architecture is design principles. A design principle differs from an architecture decision in that a design principle is a guideline rather than a hard-and-fast rule. For example, the design principle illustrated in Figure 1-6 states that the development teams should leverage asynchronous messaging between services within a microservices architecture to increase performance. An architecture decision (rule) could never cover every condition and option for communication between services, so a design principle can be used to provide guidance for the preferred method (in this case, asynchronous messaging) to allow the developer to choose a more appropriate communication protocol (such as REST or gRPC) given a specific circumstance.

Expectations of an Architect

1. Make architecture decisions
2. Continually analyze the architecture
3. Keep current with latest trends
4. Ensure compliance with decisions
5. Diverse exposure and experience
6. Have business domain knowledge
7. Possess interpersonal skills
8. Understand and navigate politics

FACULTY OF COMPUTER SCIENCE

Image source: https://www.khaleejtimes.com/wellness/are-your-expectations-ruining-your-life
Defining the role of a software architect presents as much difficulty as defining software architecture. It can range from expert programmer up to defining the strategic technical direction for the company. Rather than waste time on the fool's errand of defining the role, we recommend focusing on the expectations of an architect.
There are eight core expectations placed on a software architect, irrespective of any given role, title, or job description:

Make Architecture Decisions

An architect is expected to define the architecture decisions and design principles used to guide technology decisions within the team, the department, or across the enterprise.

FACULTY OF COMPUTER SCIENCE

Guide is the key operative word in this first expectation. An architect should guide rather than specify technology choices. For example, an architect might make a decision to use React.js for frontend development. In this case, the architect is making a technical decision rather than an architectural decision or design principle that will help the development team make choices. An architect should instead instruct development teams to use a reactive-based framework for frontend web development, hence guiding the development team in making the choice between Angular, Elm, React.js, Vue, or any of the other reactive-based web frameworks.
Guiding technology choices through architecture decisions and design principles is difficult. The key to making effective architectural decisions is asking whether the architecture decision is helping to guide teams in making the right technical choice or whether the architecture decision makes the technical choice for them. That said, an architect on occasion might need to make specific technology decisions in order to preserve a particular architectural characteristic such as scalability, performance, or availability. In this case it would be still considered an architectural decision, even though it specifies a particular technology. Architects often struggle with finding the correct line, so Chapter 19 is entirely about architecture decisions.

Continually Analyze the Architecture

An architect is expected to continually analyze the architecture and current technology environment and then recommend solutions for improvement.

FACULTY OF COMPUTER SCIENCE

This expectation of an architect refers to architecture vitality, which assesses how viable the architecture that was defined three or more years ago is today, given changes in both business and technology. In our experience, not enough architects focus their energies on continually analyzing existing architectures. As a result, most architectures experience elements of structural decay, which occurs when developers make coding or design changes that impact the required architectural characteristics, such as performance, availability, and scalability.
Other forgotten aspects of this expectation that architects frequently forget are testing and release environments. Agility for code modification has obvious benefits, but if it takes teams weeks to test changes and months for releases, then architects cannot achieve agility in the overall architecture.

An architect must holistically analyze changes in technology and problem domains to determine the soundness of the architecture. While this kind of consideration rarely appears in a job posting, architects must meet this expectation to keep applications relevant.

Keep Current with Latest Trends

An architect is expected to keep current with the latest technology and industry trends.

FACULTY OF COMPUTER SCIENCE

Developers must keep up to date on the latest technologies they use on a daily basis to remain relevant (and to retain a job!). An architect has an even more critical requirement to keep current on the latest technical and industry trends. The decisions an architect makes tend to be long-lasting and difficult to change. Understanding and following key trends helps the architect prepare for the future and make the correct decision.
Tracking trends and keeping current with those trends is hard, particularly for a software architect. In Chapter 24 we discuss various techniques and resources on how to do this.

Diverse Exposure and Experience

An architect is expected to have exposure to multiple and diverse technologies, frameworks, platforms, and environments.

FACULTY OF COMPUTER SCIENCE

This expectation does not mean an architect must be an expert in every framework, platform, and language, but rather that an architect must at least be familiar with a variety of technologies. Most environments these days are heterogeneous, and at a minimum an architect should know how to interface with multiple systems and services, irrespective of the language, platform, and technology those systems or services are written in.

One of the best ways of mastering this expectation is for the architect to stretch their comfort zone. Focusing only on a single technology or platform is a safe haven. An effective software architect should be aggressive in seeking out opportunities to gain experience in multiple languages, platforms, and technologies. A good way of mastering this expectation is to focus on technical breadth rather than technical depth. Technical breadth includes the stuff you know about, but not at a detailed level, combined with the stuff you know a lot about. For example, it is far more valuable for an architect to be familiar with 10 different caching products and the associated pros and cons of each rather than to be an expert in only one of them.

Have Business Domain Knowledge

An architect is expected to have a certain level of business domain expertise.

FACULTY OF COMPUTER SCIENCE

Effective software architects understand not only technology but also the business domain of a problem space. Without business domain knowledge, it is difficult to understand the business problem, goals, and requirements, making it difficult to design an effective architecture to meet the requirements of the business. Imagine being an architect at a large financial institution and not understanding common financial terms such as an average directional index, aleatory contracts, rates rally, or even nonpriority debt. Without this knowledge, an architect cannot communicate with stakeholders and business users and will quickly lose credibility.
The most successful architects we know are those who have broad, hands-on technical knowledge coupled with a strong knowledge of a particular domain. These software architects are able to effectively communicate with C-level executives and business users using the domain knowledge and language that these stakeholders know and understand. This in turn creates a strong level of confidence that the software architect knows what they are doing and is competent to create an effective and correct architecture.

Possess Interpersonal Skills

An architect is expected to possess exceptional interpersonal skills, including teamwork, facilitation, and leadership.

FACULTY OF COMPUTER SCIENCE

Having exceptional leadership and interpersonal skills is a difficult expectation for most developers and architects. As technologists, developers and architects like to solve technical problems, not people problems. However, as Gerald Weinberg was famous for saying, “no matter what they tell you, it’s always a people problem.” An architect is not only expected to provide technical guidance to the team, but is also expected to lead the development teams through the implementation of the architecture. Leadership skills are at least half of what it takes to become an effective software architect, regardless of the role or title the architect has.
The industry is flooded with software architects, all competing for a limited number of architecture positions. Having strong leadership and interpersonal skills is a good way for an architect to differentiate themselves from other architects and stand out from the crowd. We’ve known many software architects who are excellent technologists but are ineffective architects due to the inability to lead teams, coach and mentor developers, and effectively communicate ideas and architecture decisions and principles. Needless to say, those architects have difficulties holding a position or job.

Understand and Navigate Politics

An architect is expected to understand the political climate of the enterprise and be able to navigate the politics.

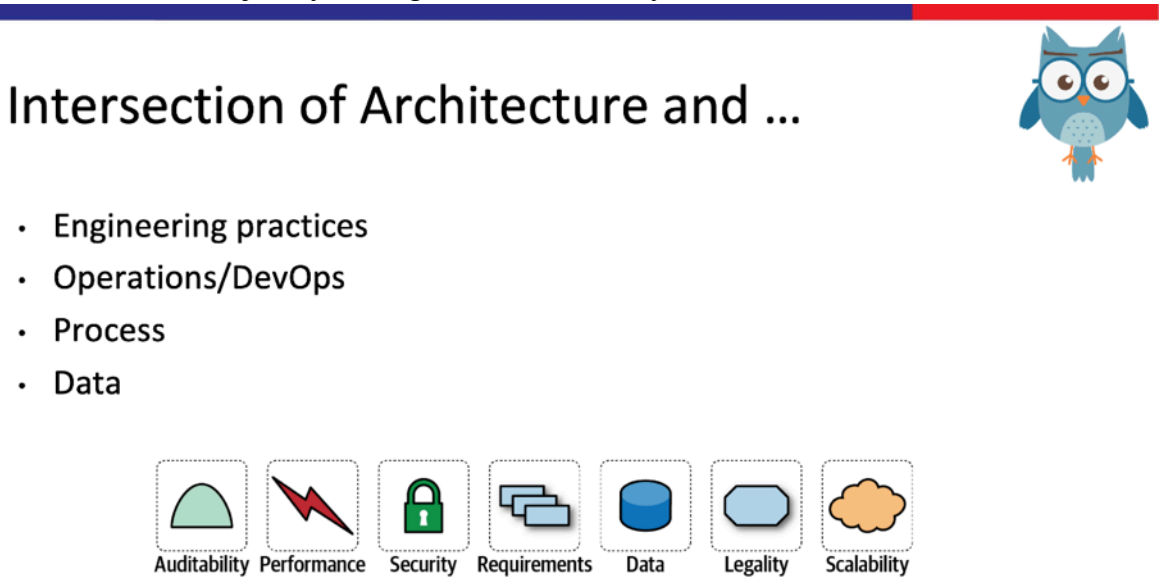
FACULTY OF COMPUTER SCIENCE

It might seem rather strange talk about negotiation and navigating office politics in a book about software architecture. To illustrate how important and necessary negotiation skills are, consider the scenario where a developer makes the decision to leverage the strategy pattern to reduce the overall cyclomatic complexity of a particular piece of complex code. Who really cares? One might applaud the developer for using such a pattern, but in almost all cases the developer does not need to seek approval for such a decision.

Now consider the scenario where an architect, responsible for a large customer relationship management system, is having issues controlling database access from other systems, securing certain customer data, and making any database schema change because too many other systems are using the CRM database. The architect therefore makes the decision to create what are called application silos, where each application database is only accessible from the application owning that database. Making this decision will give the architect better control over the customer data, security, and change control. However, unlike the previous developer scenario, this decision will also be challenged by almost everyone in the company (with the possible exception of the CRM application team, of course). Other applications need the customer management data. If those applications are no longer able to access the database directly, they must now ask the CRM system for the data, requiring remote access calls through REST, SOAP, or some other remote access protocol.

The main point is that almost every decision an architect makes will be challenged. Architectural decisions will be challenged by product owners, project managers, and business stakeholders due to increased costs or increased effort (time) involved. Architectural decisions will also be challenged by developers who feel their approach is better. In either case, the architect must navigate the politics of the company and apply basic negotiation skills to get most decisions approved. This fact can be very frustrating to a software architect, because most decisions made as a developer did not require approval or even a review. Programming aspects such as code structure, class design, design pattern selection, and sometimes even language choice are all part

of the art of programming. However, an architect, now able to finally be able to make broad and important decisions, must justify and fight for almost every one of those decisions.



FACULTY OF COMPUTER SCIENCE

The scope of software architecture has grown over the last decade to encompass more and more responsibility and perspective. A decade ago, the typical relationship between architecture and operations was contractual and formal, with lots of bureaucracy. Most companies, trying to avoid the complexity of hosting their own operations, frequently outsourced operations to a third-party company, with contractual obligations for service-level agreements, such as uptime, scale, responsiveness, and a host of other important architectural characteristics. Now, architectures such as microservices freely leverage former solely operational concerns. For example, elastic scale was once painfully built into architectures (see Chapter 15), while microservices handled it less painfully via a liaison between architects and DevOps.

3. Law of Software Architectures

Law of Software Architecture

Everything in software architecture is a trade-off.

-- First Law of Software Architecture

FACULTY OF COMPUTER SCIENCE

Nothing exists on a nice, clean spectrum for software architects. Every decision must take into account many opposing factors.
"If an architect thinks they have discovered something that isn't a trade-off, more likely they just haven't identified the trade-off yet." --Corollary 1

Law of Software Architecture

Why is more important than how.

-- Second Law of Software Architecture

FACULTY OF COMPUTER SCIENCE

We define software architecture in terms beyond structural scaffolding, incorporating principles, characteristics, and so on. Architecture is broader than just the combination of structural elements.

Understanding the why is more important than knowing how to build it. An architect should understand why such approach need to be taken, and why some are not. The effect of the approach needs to be considered in depth. It may influence other aspect as well.

4. Architectural Thinking

Architectural Thinking

FACULTY OF COMPUTER SCIENCE

Image Source:
https://www.univariety.com/blog/seven-ways-to-architect-your-future/https://robogarden.cn/es/blog/six-unique-characteristics-for-all-programmers

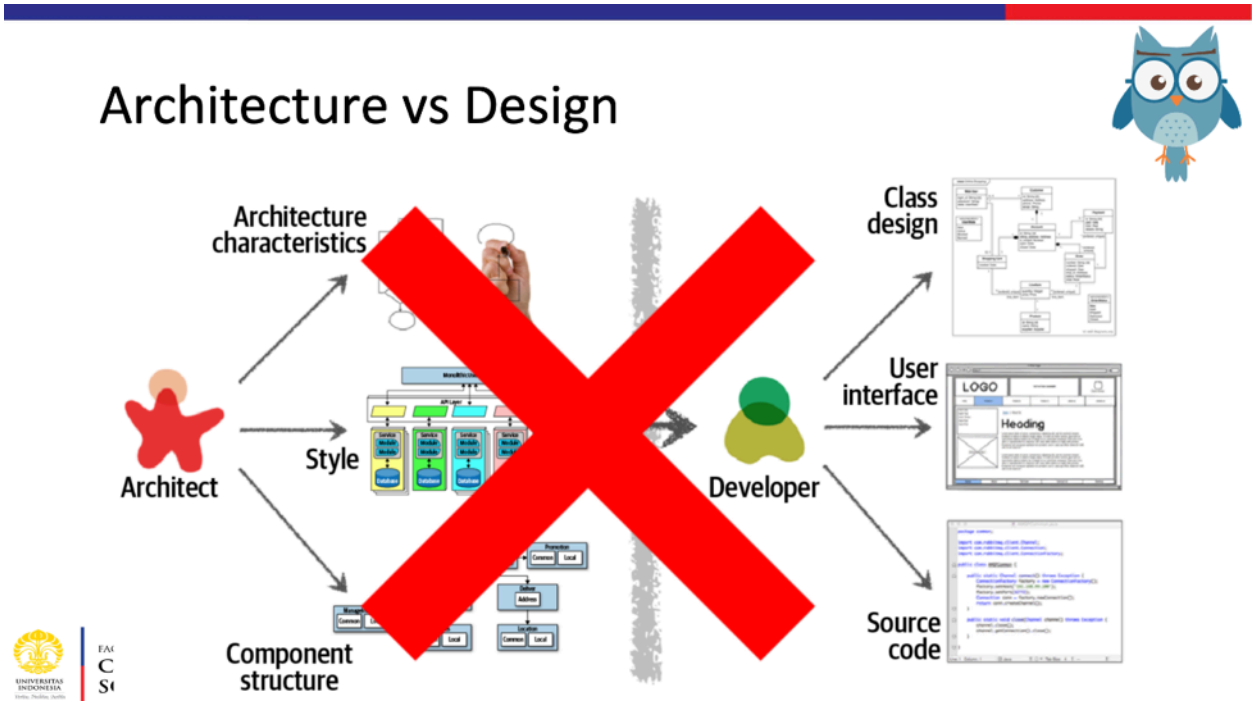
Architectural Thinking

1. It's understanding the **difference between architecture and design** and knowing how to collaborate with development teams to make architecture work.
2. It's about having a **wide breadth of technical knowledge** while still maintaining a certain level of technical depth, allowing the architect to see solutions and possibilities that others do not see.
3. It's about understanding, **analyzing, and reconciling trade-offs** between various solutions and technologies.
4. It's about **understanding the importance of business drivers** and how they translate to architectural concerns.

FACULTY OF COMPUTER SCIENCE

An architect sees things differently from a developer's point of view, much in the same way a meteorologist might see clouds differently from an artist's point of view. This is called architectural thinking. Unfortunately, too many architects believe that architectural thinking is simply just "thinking about the architecture."

Architectural thinking is much more than that. It is seeing things with an architectural eye, or an architectural point of view. There are four main aspects of thinking like an architect. First, it's understanding the difference between architecture and design and knowing how to collaborate with development teams to make architecture work. Second, it's about having a wide breadth of technical knowledge while still maintaining a certain level of technical depth, allowing the architect to see solutions and possibilities that others do not see. Third, it's about understanding, analyzing, and reconciling trade-offs between various solutions and technologies. Finally, it's about understanding the importance of business drivers and how they translate to architectural concerns.



The difference between architecture and design is often a confusing one. Where does architecture end and design begin? What responsibilities does an architect have versus those of a developer? Thinking like an architect is knowing the difference between architecture and design and seeing how the two integrate closely to form solutions to business and technical problems.

Consider Figure 2-1, which illustrates the traditional responsibilities an architect has, as compared to those of a developer. As shown in the diagram, an architect is responsible for things like analyzing business requirements to extract and define the architectural characteristics ("-ilities"), selecting which architecture patterns and styles would fit the problem domain, and creating components (the building blocks of the system). The artifacts created from these activities are then handed off to the development team, which is responsible for creating class diagrams for each component, creating user interface screens, and developing and testing source code.

There are several issues with the traditional responsibility model illustrated in Figure 2-1. As a matter of fact, this illustration shows exactly why architecture rarely works. Specifically, it is the unidirectional arrow passing though the virtual and physical barriers separating the architect from the developer that causes all of the problems associated with architecture. Decisions an architect makes sometimes never make it to the development teams, and decisions development teams make that change the architecture rarely get back to the architect. In this model the

architect is disconnected from the development teams, and as such the architecture rarely provides what it was originally set out to do.

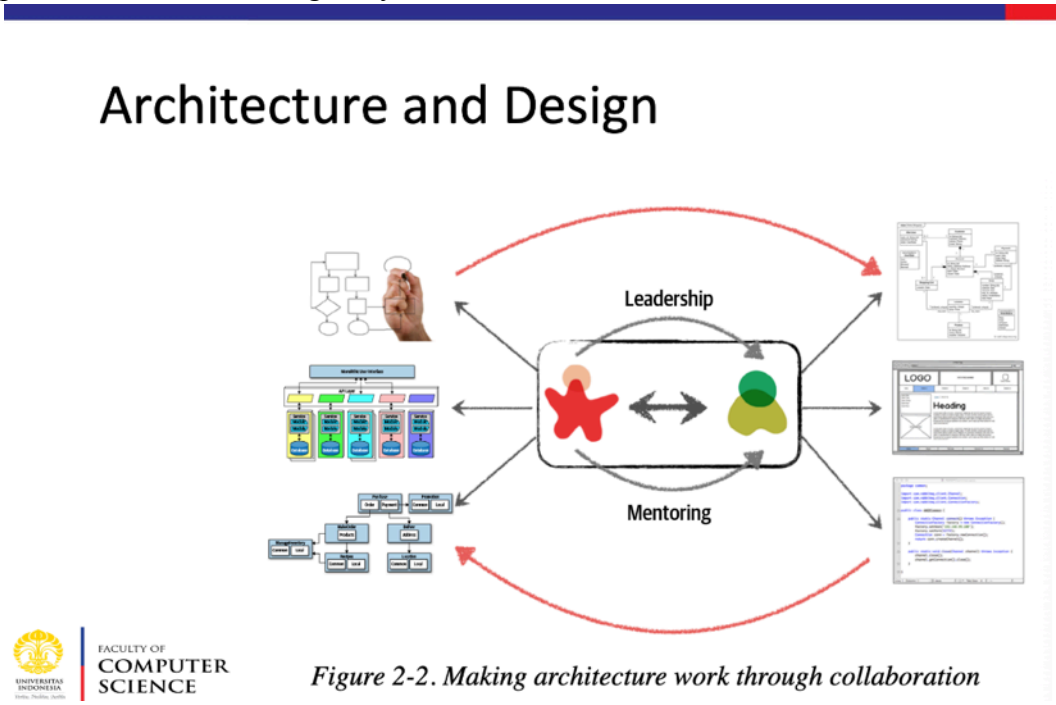


Figure 2-2. Making architecture work through collaboration

To make architecture work, both the physical and virtual barriers that exist between architects and developers must be broken down, thus forming a strong bidirectional relationship between architects and development teams. The architect and developer must be on the same virtual team to make this work, as depicted in Figure 2-2. Not only does this model facilitate strong bidirectional communication between architecture and development, but it also allows the architect to provide mentoring and coaching to developers on the team.

Unlike the old-school waterfall approaches to static and rigid software architecture, the architecture of today's systems changes and evolves every iteration or phase of a project. A tight collaboration between the architect and the development team is essential for the success of any software project. So where does architecture end and design begin? It doesn't. They are both part of the circle of life within a software project and must always be kept in synchronization with each other in order to succeed.

Technical Breadth

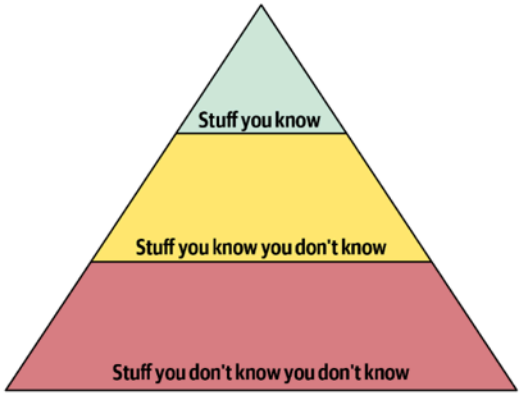


Figure 2-3. The pyramid representing all knowledge

FACULTY OF COMPUTER SCIENCE

The scope of technological detail differs between developers and architects. Unlike a developer, who must have a significant amount of technical depth to perform their job, a software architect must have a significant amount of technical breadth to think like an architect and see things with an architecture point of view. This is illustrated by the knowledge pyramid shown in Figure 2-3, which encapsulates all the technical knowledge in the world. It turns out that the kind of information a technologist should value differs with career stages.

As shown in Figure 2-3, any individual can partition all their knowledge into three sections: stuff you know, stuff you know you don't know, and stuff you don't know you don't know.

Stuff you know includes the technologies, frameworks, languages, and tools a technologist uses on a daily basis to perform their job, such as knowing Java as a Java programmer. Stuff you know you don't know includes those things a technologist knows a little about or has heard of but has little or no expertise in. A good example of this level of knowledge is the Clojure programming language. Most technologists have heard of Clojure and know it's a programming language based on Lisp, but they can't code in the language. Stuff you don't know you don't know is the largest part of the knowledge triangle and includes the entire host of technologies, tools, frameworks, and languages that would be the perfect solution to a problem a technologist is trying to solve, but the technologist doesn't even know those things exist.

A developer's early career focuses on expanding the top of the pyramid, to build experience and expertise. This is the ideal focus early on, because developers need more perspective, working knowledge, and hands-on experience. Expanding the top incidentally expands the middle section; as developers encounter more technologies and related artifacts, it adds to their stock of stuff you know you don't know.

Technical Breadth

- Architect should have sufficiently strong technical depth and continue broaden its technical breadth
- Once he/she become a software architect, he/she should focus on broaden its technical breadth.

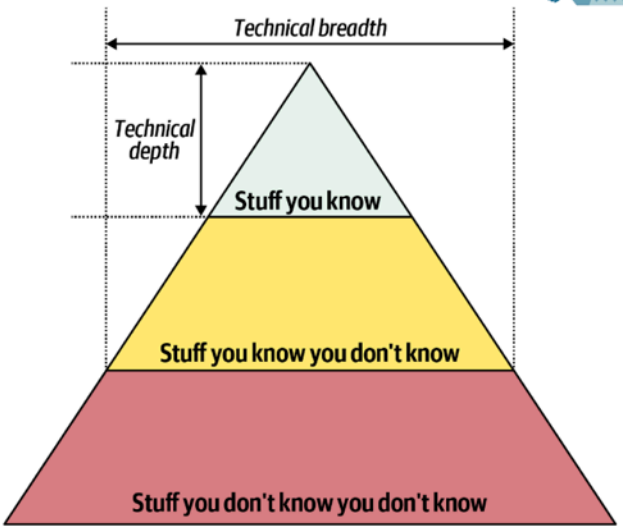


Figure 2-5. What someone knows is technical depth, and how much

However, the stuff you know is also the stuff you must maintain—nothing is static in the software world. If a developer becomes an expert in Ruby on Rails, that expertise won't last if they ignore Ruby on Rails for a year or two. The things at the top of the pyramid require time investment to maintain expertise. Ultimately, the size of the top of an individual's pyramid is their technical depth.

However, the nature of knowledge changes as developers transition into the architect role. A large part of the value of an architect is a broad understanding of technology and how to use it to solve particular problems. For example, as an architect, it is more beneficial to know that five solutions exist for a particular problem than to have singular expertise in only one. The most important parts of the pyramid for architects are the top and middle sections; how far the middle section penetrates into the bottom section represents an architect's technical breadth, as shown in Figure 2-5.

As an architect, breadth is more important than depth. Because architects must make decisions that match capabilities to technical constraints, a broad understanding of a wide variety of solutions is valuable. Thus, for an architect, the wise course of action is to sacrifice some hard-won expertise and use that time to broaden their portfolio, as shown in Figure 2-6. As illustrated in the diagram, some areas of expertise will remain, probably in particularly enjoyable technology areas, while others usefully atrophy.

Architects should focus on technical breadth so that they have a larger quiver from which to draw arrows. Developers transitioning to the architect role may have to change the way they view knowledge acquisition. Balancing their portfolio of knowledge regarding depth versus breadth is something every developer should consider throughout their career.

Analyzing Trade-Offs

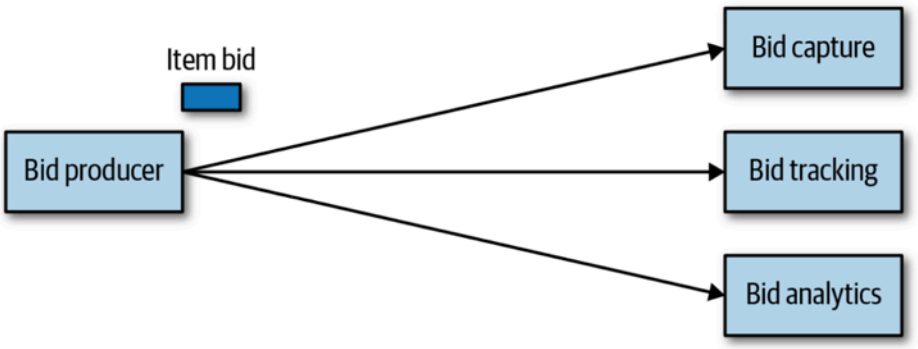
- Everything in architecture is a trade-off.
- So, you cannot answer a software architecture by googling, because the answer will depend on the context and how the trade-off is decided.

There are no right or wrong answers in architecture—only trade-offs.

FACULTY OF COMPUTER SCIENCE

Thinking like an architect is all about seeing trade-offs in every solution, technical or otherwise, and analyzing those trade-offs to determine what is the best solution. To quote Mark (one of your authors): “Architecture is the stuff you can’t Google.”
Everything in architecture is a trade-off, which is why the famous answer to every architecture question in the universe is “it depends.” While many people get increasingly annoyed at this answer, it is unfortunately true. You cannot Google the answer to whether REST or messaging would be better, or whether microservices is the right architecture style, because it does depend. It depends on the deployment environment, business drivers, company culture, budgets, timeframes, developer skill set, and dozens of other factors. Everyone’s environment, situation, and problem is different, hence why architecture is so hard. To quote Neal (another one of your authors): “There are no right or wrong answers in architecture—only trade-offs.”

Analyzing Trade-Offs: Example



FACULTY OF COMPUTER SCIENCE

For example, consider an item auction system, as illustrated in Figure, where someone places a bid for an item up for auction.
The Bid Producer service generates a bid from the bidder and then sends that bid amount to the Bid Capture, Bid Tracking, and Bid Analytics services. This could be done by using queues in a point-to-point messaging fashion or by using a topic in a publish-and-subscribe messaging fashion. Which one should the architect use? You can't Google the answer. Architectural thinking requires the architect to analyze the trade-offs associated with each option and select the best one given the specific situation.

Analyzing Trade-Offs: Example

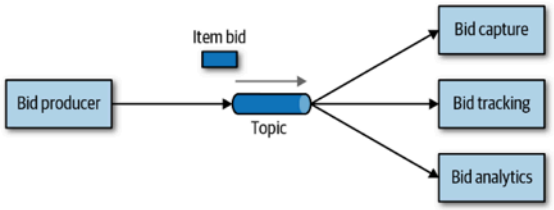


Figure 2-8. Use of a topic for communication between services

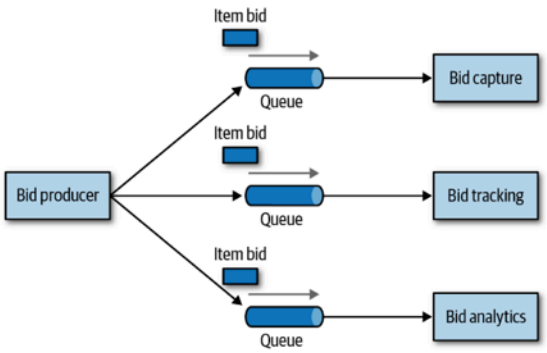


Figure 2-9. Use of queues for communication between services

FACULTY OF COMPUTER SCIENCE

The two messaging options for the item auction system are shown in Figures 2-8 and 2-9, with Figure 2-8 illustrating the use of a topic in a publish-and-subscribe messaging model, and Figure 2-9 illustrating the use of queues in a point-to-point messaging model.

The clear advantage (and seemingly obvious solution) to this problem in Figure 2-8 is that of architectural extensibility. The Bid Producer service only requires a single connection to a topic, unlike the queue solution in Figure 2-9 where the Bid Producer needs to connect to three different queues. If a new service called Bid History were to be added to this system due to the requirement to provide each bidder with a history of all the bids they made in each auction, no changes at all would be needed to the existing system. When the new Bid History service is created, it could simply subscribe to the topic already containing the bid information. In the queue option shown in Figure 2-9, however, a new queue would be required for the Bid History service, and the Bid Producer would need to be modified to add an additional connection to the new queue. The point here is that using queues requires significant change to the system when adding new bidding functionality, whereas with the topic approach no changes are needed at all in the existing infrastructure. Also, notice that the Bid Producer is more decoupled in the topic option—the Bid Producer doesn't know how the bidding information will be used or by which services. In the queue option the Bid Producer knows exactly how the bidding information is used (and by whom), and hence is more coupled to the system.

With this analysis it seems clear that the topic approach using the publish-and-subscribe messaging model is the obvious and best choice.

Thinking architecturally is looking at the benefits of a given solution, but also analyzing the negatives, or trade-offs, associated with a solution. Continuing with the auction system example, a software architect would analyze the negatives of the topic solution. In analyzing the differences, notice first in Figure 2-8 that with a topic, anyone can access bidding data, which introduces a possible issue with data access and data security. In the queue model illustrated in Figure 2-9, the data sent to the queue can only be accessed by the specific consumer receiving that message. If a rogue service did listen in on a queue, those bids would not be received by the corresponding service, and a notification would immediately be sent about the loss of data (and hence a possible security breach). In other words, it is very easy to wiretap into a topic, but not a queue.

In addition to the security issue, the topic solution in Figure 2-8 only supports homogeneous contracts. All services receiving the bidding data must accept the same contract and set of bidding data. In the queue option in Figure 2-9, each consumer can have its own contract specific to the data it needs. For example, suppose the new Bid History service requires the current asking price along with the bid, but no other service needs that information. In this case, the contract would need to be modified, impacting all other services using that data. In the queue model, this would be a separate channel, hence a separate contract not impacting any other service.

Which one to choose? It depends!

Programmers know the benefits of everything
and the trade-offs of nothing.
Architects need to understand both.

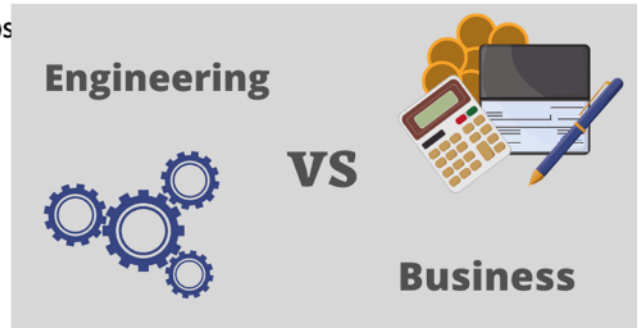
-- Rich Hickey, the creator of the Clojure programming language

FACULTY OF COMPUTER SCIENCE

Another disadvantage of the topic model illustrated in Figure 2-8 is that it does not support monitoring of the number of messages in the topic and hence auto-scaling capabilities. However, with the queue option in Figure 2-9, each queue can be monitored individually, and programmatic load balancing applied to each bidding consumer so that each can be automatically scaled independency from one another. Note that this trade-off is technology specific in that the Advanced Message Queuing Protocol (AMQP) can support programmatic load balancing and monitoring because of the separation between an exchange (what the producer sends to) and a queue (what the consumer listens to).
Given this trade-off analysis, now which is the better option?

Understanding Business Drivers

- Translating business requirements into architecture characteristics (such as scalability, performance, and availability).
- This is a challenging task
  - business domain knowledge
  - healthy collaborative relationships



FACULTY OF COMPUTER SCIENCE

Thinking like an architect is understanding the business drivers that are required for the success of the system and translating those requirements into architecture characteristics (such as scalability, performance, and availability). This is a challenging task that requires the architect to have some level of business domain knowledge and healthy, collaborative relationships with key business stakeholders.

The challenges:
Balancing Architecture and Hands-On Coding

- Every architect should code!
- Every architect should be able to maintain a certain level of detail.

- How can a software architect still remain hands-on and maintain some level of technical detail?

FACULTY OF COMPUTER SCIENCE

One of the difficult tasks an architect faces is how to balance hands-on coding with software architecture. We firmly believe that every architect should code and be able to maintain a certain level of technical depth (see “Technical Breadth”). While this may seem like an easy task, it is sometimes rather difficult to accomplish.

The first tip in striving for a balance between hands-on coding and being a software architect is avoiding the bottleneck trap. The bottleneck trap occurs when the architect has taken ownership of code within the critical path of a project (usually the underlying framework code) and becomes a bottleneck to the team. This happens because the architect is not a full-time developer and therefore must balance between playing the developer role (writing and testing source code) and the architect role (drawing diagrams, attending meetings, and well, attending more meetings).

One way to avoid the bottleneck trap as an effective software architect is to delegate the critical path and framework code to others on the development team and then focus on coding a piece of business functionality (a service or a screen) one to three iterations down the road. Three positive things happen by doing this. First, the architect is gaining hands-on experience writing production code while no longer becoming a bottleneck on the team. Second, the critical path and framework code is distributed to the development team (where it belongs), giving them ownership and a better understanding of the harder parts of the system. Third, and perhaps most important, the architect is writing the same business-related source code as the development team and is therefore better able to identify with the development team in terms of the pain they might be going through with processes, procedures, and the development environment.

The challenges: Balancing Architecture and Hands-On Coding

- There are four basic ways an architect can still remain hands-on at work
  1. Do frequent proof-of-concepts or POCs. The architect should write the best production-quality code they can.
     1. Throwaway proof-of-concept code goes into the source code repository and becomes the reference architecture or guiding example for others to follow.
     2. Writing production-quality proof-of-concept code, the architect gets practice writing quality, well-structured code rather than continually developing bad coding practices.
  2. Working on bug fixes within an iteration. Tackle some of the technical debt stories or architecture stories, freeing the development team up to work on the critical functional user stories.
  3. Leveraging automation by creating simple command-line tools and analyzers
  4. Do frequent code reviews.

FACULTY OF COMPUTER SCIENCE

There are four basic ways an architect can still remain hands-on at work without having to "practice coding from home" (although we recommend practicing coding at home as well).
The first way is to do frequent proof-of-concepts or POCs. This practice not only requires the architect to write source code, but it also helps validate an architecture decision by taking the implementation details into account. For example, if an architect is stuck trying to make a decision between two caching solutions, one effective way to help make this decision is to develop a working example in each caching product and compare the results. This allows the architect to see first-hand the implementation details and the amount of effort required to develop the full solution. It also allows the architect to better compare architectural characteristics such as scalability, performance, or overall fault tolerance of the different caching solutions.
Our advice when doing proof-of-concept work is that, whenever possible, the architect should write the best production-quality code they can. We recommend this practice for two reasons. First, quite often, throwaway proof-of-concept code goes into the source code repository and becomes the reference architecture or guiding example for others to follow. The last thing an architect would want is for their throwaway, sloppy code to be a representation of their typical work. The second reason is that by writing production-quality proof-of-concept code, the architect gets practice writing quality, well-structured code rather than continually developing bad coding practices.
Another way an architect can remain hands-on is to tackle some of the technical debt stories or architecture stories, freeing the development team up to work on the critical functional user stories. These stories are usually low priority, so if the architect does not have the chance to complete a technical debt or architecture story within a given iteration, it's not the end of the world and generally does not impact the success of the iteration.
Similarly, working on bug fixes within an iteration is another way of maintaining hands-on coding while helping the development team as well. While certainly not glamorous, this

technique allows the architect to identify where issues and weakness may be within the code base and possibly the architecture.

Leveraging automation by creating simple command-line tools and analyzers to help the development team with their day-to-day tasks is another great way to maintain hands-on coding skills while making the development team more effective. Look for repetitive tasks the development team performs and automate the process. The development team will be grateful for the automation. Some examples are automated source validators to help check for specific coding standards not found in other lint tests, automated checklists, and repetitive manual code refactoring tasks.

Automation can also be in the form of architectural analysis and fitness functions to ensure the vitality and compliance of the architecture. For example, an architect can write Java code in ArchUnit in the Java platform to automate architectural compliance, or write custom fitness functions to ensure architectural compliance while gaining hands-on experience. We talk about these techniques in Chapter 6.

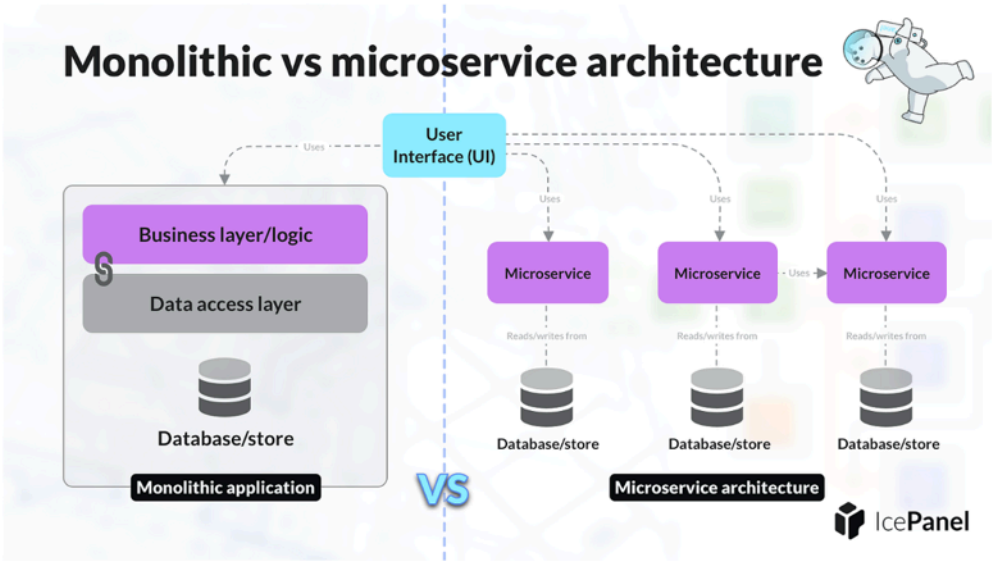
A final technique to remain hands-on as an architect is to do frequent code reviews. While the architect is not actually writing code, at least they are involved in the source code. Further, doing code reviews has the added benefits of being able to ensure compliance with the architecture and to seek out mentoring and coaching opportunities on the team.

5. Microservices Architecture

Microservice Architecture

Chapter 17, Mark Richards and Neal Ford, Fundamental of Software Architectures: An engineering approach, O'Reilly, 2021

And various resources



FACULTY OF COMPUTER SCIENCE

Image Source:
https://icepanel.medium.com/monolithic-vs-microservices-architectures-e71c75b252d1

Microservice: Background

- It is popularized by a famous blog entry by Martin Fowler and James Lewis, entitled "Microservices," published in March 2014.
- It is heavily inspired by the ideas in domain-driven design (DDD), a logical design process for software projects.
- One concept in particular from DDD, bounded context, decidedly inspired microservices.
- The concept of bounded context represents a decoupling style. Prefers duplication to coupling.
- When a developer defines a domain, that domain includes many entities and behaviors, identified in artifacts such as code and database schemas.

FACULTY OF COMPUTER SCIENCE

Microservices is an extremely popular architecture style that has gained significant momentum in recent years. In this chapter, we provide an overview of the important characteristics that set this architecture apart, both topologically and philosophically.

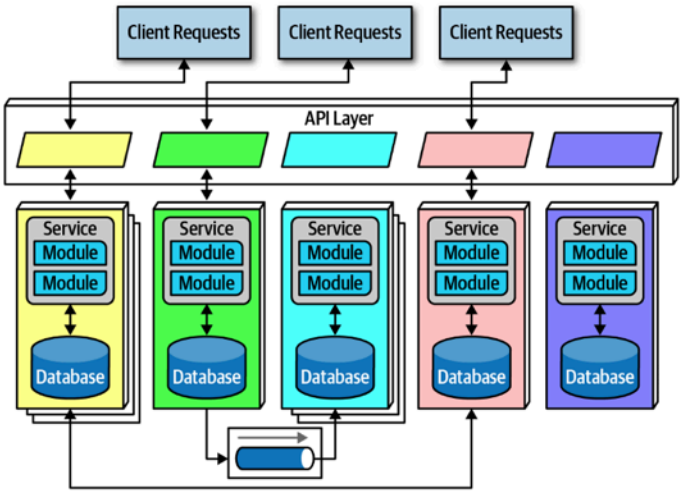
History

Most architecture styles are named after the fact by architects who notice a particular pattern that keeps reappearing—there is no secret group of architects who decide what the next big movement will be. Rather, it turns out that many architects end up making common decisions as the software development ecosystem shifts and changes. The common best ways of dealing with and profiting from those shifts become architecture styles that others emulate.

Microservices differs in this regard—it was named fairly early in its usage and popularized by a famous blog entry by Martin Fowler and James Lewis entitled "Microservices," published in March 2014. They recognized many common characteristics in this relatively new architectural style and delineated them. Their blog post helped define the architecture for curious architects and helped them understand the underlying philosophy.

Microservices is heavily inspired by the ideas in domain-driven design (DDD), a logical design process for software projects. One concept in particular from DDD, bounded context, decidedly inspired microservices. The concept of bounded context represents a decoupling style. When a developer defines a domain, that domain includes many entities and behaviors, identified in artifacts such as code and database schemas.

Microservice Architecture: Topology



FACULTY OF COMPUTER SCIENCE

Figure 17-1. The topology of the microservices architecture style

As illustrated in Figure 17-1, due to its single-purpose nature, the service size in microservices is much smaller than other distributed architectures, such as the orchestration-driven service-oriented architecture. Architects expect each service to include all necessary parts to operate independently, including databases and other dependent components. The different characteristics appear in the following sections.

Microservice: Distributed

- Each service runs in its own process, which originally implied a physical computer but quickly evolved to virtual machines and containers.
- With cloud resources and container technology, teams can reap the benefits of extreme decoupling, both at the domain and operational level.
- Performance is often the negative side effect of the distributed nature of microservices.
- Determining the granularity of services the key to success in this architecture -> Scaling

FACULTY OF COMPUTER SCIENCE

Microservices form a distributed architecture: each service runs in its own process, which originally implied a physical computer but quickly evolved to virtual machines and containers. Decoupling the services to this degree allows for a simple solution to a common problem in

architectures that heavily feature multitenant infrastructure for hosting applications. For example, when using an application server to manage multiple running applications, it allows operational reuse of network bandwidth, memory, disk space, and a host of other benefits. However, if all the supported applications continue to grow, eventually some resource becomes constrained on the shared infrastructure. Another problem concerns improper isolation between shared applications. Separating each service into its own process solves all the problems brought on by sharing. Before the evolutionary development of freely available open source operating systems, combined with automated machine provisioning, it was impractical for each domain to have its own infrastructure. Now, however, with cloud resources and container technology, teams can reap the benefits of extreme decoupling, both at the domain and operational level.
Performance is often the negative side effect of the distributed nature of microservices. Network calls take much longer than method calls, and security verification at every endpoint adds additional processing time, requiring architects to think carefully about the implications of granularity when designing the system. Because microservices is a distributed architecture, experienced architects advise against the use of transactions across service boundaries, making determining the granularity of services the key to success in this architecture.

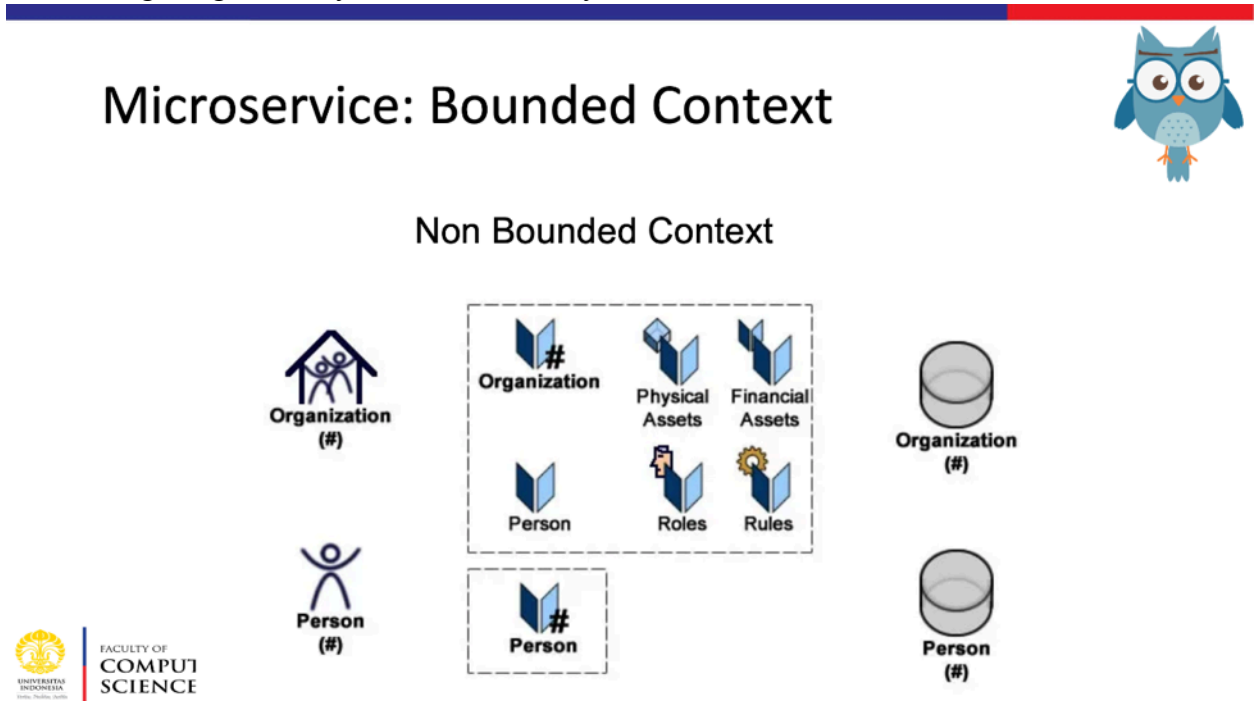


Image source: https://caminao.blog/2016/09/13/focus-bctx-ocpt/

Microservice: Bounded Context

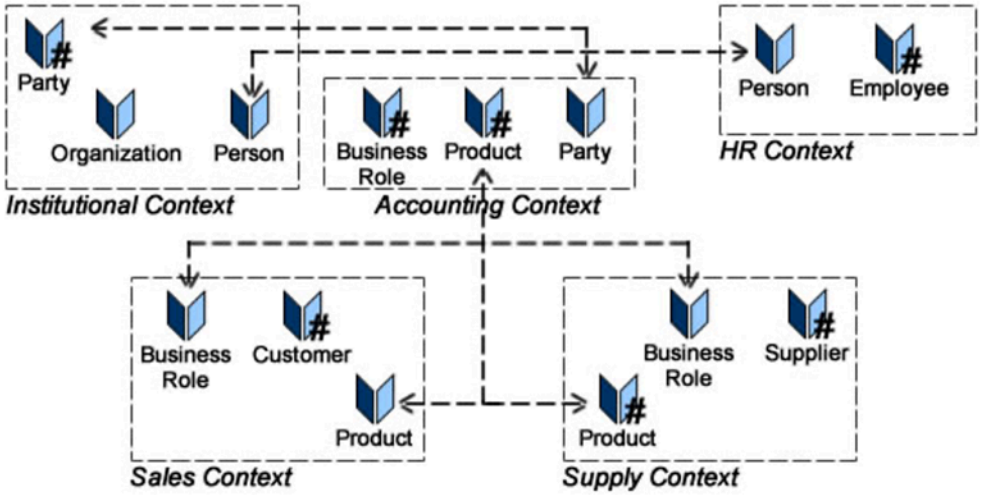


Image source: https://caminao.blog/2016/09/13/focus-bctx-ocpt/

Microservice: Bounded Context

- Each service models a domain or workflow.
- Contexts are introduced to:
  - conciliate continuity and integrity,
  - managed through aggregates, and
  - semantics and functional accesses,
  - managed through contexts;
- bounded contexts (BCs) are ones with shared business entities.

The driving philosophy of microservices is the notion of bounded context: each service models a domain or workflow. Thus, each service includes everything necessary to operate within the application, including classes, other subcomponents, and database schemas. This philosophy drives many of the decisions architects make within this architecture. For example, in a monolith, it is common for developers to share common classes, such as Address, between disparate parts of the application. However, microservices try to avoid coupling, and thus an architect building this architecture style prefers duplication to coupling.

Microservices take the concept of a domain-partitioned architecture to the extreme. Each service is meant to represent a domain or subdomain; in many ways, microservices is the physical embodiment of the logical concepts in domain-driven design.

Microservice: Granularity

- ***Purpose***: The most obvious boundary relies on the inspiration for the architecture style, a domain. Ideally, each microservice should be extremely functionally cohesive, contributing one significant behavior on behalf of the overall application.
- ***Transactions***: Bounded contexts are business workflows, and often the entities that need to cooperate in a transaction show architects a good service boundary.
- ***Choreography***: If an architect builds a set of services that offer excellent domain isolation yet require extensive communication to function, the architect may consider bundling these services back into a larger service to avoid the communication overhead.

FACULTY OF COMPUTER SCIENCE

Architects struggle to find the correct granularity for services in microservices, and often make the mistake of making their services too small, which requires them to build communication links back between the services to do useful work.
"The term "microservice" is a label, not a description." -- Martin Fowler
In other words, the originators of the term needed to call this new style something, and they chose "microservices" to contrast it with the dominant architecture style at the time, service-oriented architecture, which could have been called "gigantic services". However, many developers take the term "microservices" as a commandment, not a description, and create services that are too fine-grained.
The purpose of service boundaries in microservices is to capture a domain or workflow. In some applications, those natural boundaries might be large for some parts of the system—some business processes are more coupled than others. Here are some guidelines architects can use to help find the appropriate boundaries as mentioned in the slide.
Iteration is the only way to ensure good service design. Architects rarely discover the perfect granularity, data dependencies, and communication styles on their first pass. However, after iterating over the options, an architect has a good chance of refining their design.

Microservices: Data Isolation

- Data is isolated for each service
- Microservices avoid any kind of coupling, including in databases
- It is basically driven by the bounded context concept.
- The data is defined within the context of each service.
- Differentiate a database may seem strange and complicated at first, but it has positive aspect:
  - The teams are not forced to unify around a single database
  - Each service can choose appropriate representation/tools of database
  - Easy for changing

FACULTY OF COMPUTER SCIENCE

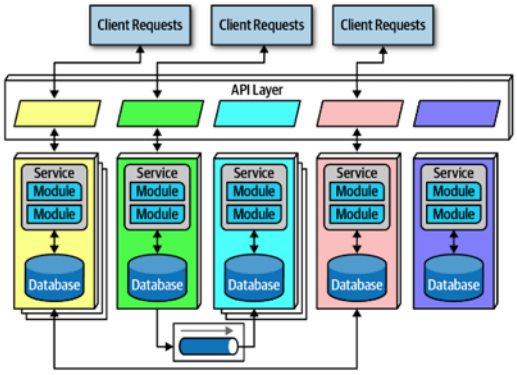
Another requirement of microservices, driven by the bounded context concept, is data isolation. Many other architecture styles use a single database for persistence. However, microservices tries to avoid all kinds of coupling, including shared schemas and databases used as integration point. Data isolation is another factor an architect must consider when looking at service granularity. Architects must be wary of the entity trap (discussed in “Entity trap”) and not simply model their services to resemble single entities in a database.

Architects are accustomed to using relational databases to unify values within a system, creating a single source of truth, which is no longer an option when distributing data across the architecture. Thus, architects must decide how they want to handle this problem: either identifying one domain as the source of truth for some fact and coordinating with it to retrieve values or using database replication or caching to distribute information.

While this level of data isolation creates headaches, it also provides opportunities. Now that teams aren’t forced to unify around a single database, each service can choose the most appropriate tool, based on price, type of storage, or a host of other factors. Teams have the advantage in a highly decoupled system to change their mind and choose a more suitable database (or other dependency) without affecting other teams, which aren’t allowed to couple to implementation details.

Microservices: API Layer

- Usually, it use other service.
- Many commercial products has added additional support, make the name become less standard, such as:
  - API Gateway
  - Service Discovery
  - Etc
- But it should not be added with capability to do mediator nor orchestration



FACULTY OF COMPUTER SCIENCE

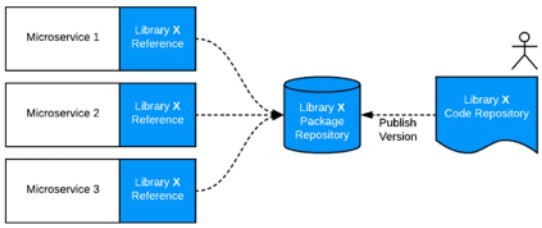
Most pictures of microservices include an API layer sitting between the consumers of the system (either user interfaces or calls from other systems), but it is optional. It is common because it offers a good location within the architecture to perform useful tasks, either via indirection as a proxy or a tie into operational facilities, such as a naming service or service discovery or API Gateways.

While an API layer may be used for variety of things, it should not be used as a mediator or orchestration tool if the architect wants to stay true to the underlying philosophy of this architecture: all interesting logic in this architecture should occur inside a bounded context, and putting orchestration or other logic in a mediator violates that rule. This also illustrates the difference between technical and domain partitioning in architecture: architects typically use mediators in technically partitioned architectures, whereas microservices is firmly domain partitioned.

Architects use service discovery as a way to build elasticity into microservices architectures. Rather than invoke a single service, a request goes through a service discovery tool, which can monitor the number and frequency of requests, as well as spin up new instances of services to handle scale or elasticity concerns. Architects often include service discovery in the service mesh, making it part of every microservice. The API layer is often used to host service discovery, allowing a single place for user interfaces or other calling systems to find and create services in an elastic, consistent way.

Microservices: Operational Reuse

- Prefers duplication to coupling
- Separate the concern of ***domain*** and ***operational***.
- **reusing code** between microservices isn't a prominent practice.
  But as we all know, duplicating code doesn't help either.
- **reusing code** within a Microservice boundary is not a question at all. The dilemma is about reusing code between Microservices.



FACULTY OF COMPUTER SCIENCE

Source: https://blog.bitsrc.io/the-dilemma-of-code-reuse-in-microservices-a925ff2b9981
In the world of Microservices, reusing code between microservices isn't a prominent practice. But as we all know, duplicating code doesn't help either. Besides, reusing code within a Microservice boundary is not a question at all. The dilemma is about reusing code between Microservices.
When we consider several Microservice teams, we want each team to operate with minimal dependencies between each other. Even when there are dependencies, we don't want them to change often, affecting each teams' autonomy. However, when we reuse code across Microservices, we fall into the trap of dependence, which could potentially disrupt the independence and agility of Microservice teams. Therefore, the challenge is to find the right balance between reuse and reliance.

Microservices: Frontend

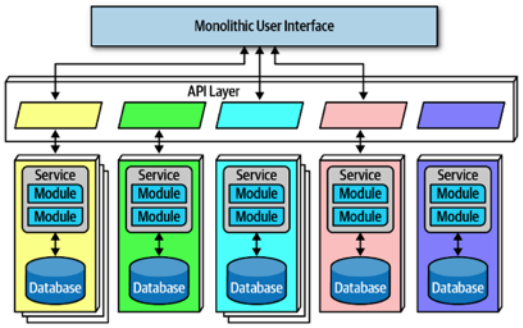


Figure 17-5. Microservices architecture with a monolithic user interface

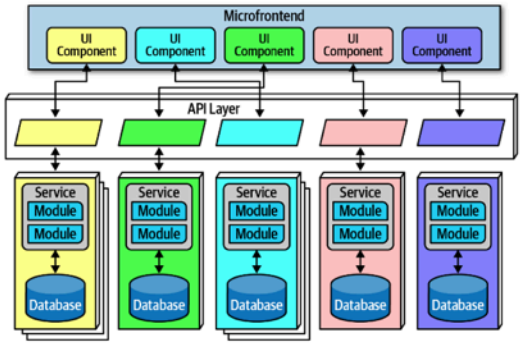


Figure 17-6. Microfrontend pattern in microservices

FACULTY OF COMPUTER SCIENCE

Microservices favors decoupling, which would ideally encompass the user interfaces as well as backend concerns. In fact, the original vision for microservices included the user interface as part of the bounded context, faithful to the principle in DDD. However, practicalities of the partitioning required by web applications and other external constraints make that goal difficult. Thus, two styles of user interfaces commonly appear for microservices architectures; the first appears in Figure 17-5.

In Figure 17-5, the monolithic frontend features a single user interface that calls through the API layer to satisfy user requests. The frontend could be a rich desktop, mobile, or web application. For example, many web applications now use a JavaScript web framework to build a single user interface. The second option for user interfaces uses microfrontends, shown in Figure 17-6.

In Figure 17-6, this approach utilizes components at the user interface level to create a synchronous level of granularity and isolation in the user interface as the backend services. Each service emits the user interface for that service, which the frontend coordinates with the other emitted user interface components. Using this pattern, teams can isolate service boundaries from the user interface to the backend services, unifying the entire domain within a single team. Developers can implement the microfrontend pattern in a variety of ways, either using a component-based web framework such as React or using one of several open source frameworks that support this pattern.

Microservices: Communication

- Architect must decide on synchronous or asynchronous communication.
- Microservices architecture typically utilize ***protocol-aware heterogeneous interoperability***.
  - Protocol-aware: REST, pRPC, sync or async
  - Heterogeneous: different technology stack, support different platform
  - Interoperability: provide support to call one another.
- For asynchronous communication, architects can use event and messages. (combining/utilizing event-driven architecture)

FACULTY OF COMPUTER SCIENCE

In microservices, architects and developers struggle with appropriate granularity, which affects both data isolation and communication. Finding the correct communication style helps teams keep services decoupled yet still coordinated in useful ways.
Fundamentally, architects must decide on synchronous or asynchronous communication. Synchronous communication requires the caller to wait for a response from the callee.
Microservices architectures typically utilize protocol-aware heterogeneous interoperability.

- Protocol-aware: Because microservices usually don't include a centralized integration hub to avoid operational coupling, each service should know how to call other services. Thus, architects commonly standardize on how particular services call each other: a certain level of REST, message queues, and so on. That means that services must know (or discover) which protocol to use to call other services.
- Heterogeneous: Because microservices is a distributed architecture, each service may be written in a different technology stack. Heterogeneous suggests that microservices fully supports polyglot environments, where different services use different platforms.
- Interoperability: Describes services calling one another. While architects in microservices try to discourage transactional method calls, services commonly call other services via the network to collaborate and send/receive information.

For asynchronous communication, architects often use events and messages, thus internally utilizing an event-driven architecture, covered in a couple next sessions; the broker and mediator patterns manifest in microservices as choreography and orchestration.

Microservices: Choreography and Orchestration

- Some services may need to call or communicate with other services. There are two approaches:
- ***Choreography*** utilizes the same communication style as a broker event-driven architecture. In other words, no central coordinator exists in this architecture, respecting the bounded context philosophy. This approach is preferable.
- ***Orchestration / Mediator*** is another block of code that handle how each service communicate with each other.

FACULTY OF COMPUTER SCIENCE

Choreography utilizes the same communication style as a broker event-driven architecture. In other words, no central coordinator exists in this architecture, respecting the bounded context philosophy. Thus, architects find it natural to implement decoupled events between services. Domain/architecture isomorphism is one key characteristic that architects should look for when assessing how appropriate an architecture style is for a particular problem. This term describes how the shape of an architecture maps to a particular architecture style.

Microservices: Choreography

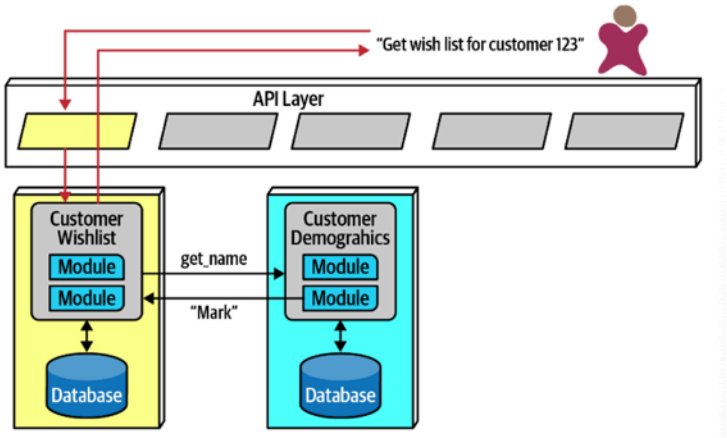


Figure 17-7. Using choreography in microservices to manage coordination

FACULTY OF COMPUTER SCIENCE

In choreography, each service calls other services as needed, without a central mediator. For example, consider the scenario shown in Figure 17-7.

In Figure 17-7, the user requests details about a user's wish list. Because the CustomerWishList service doesn't contain all the necessary information, it makes a call to CustomerDemographics to retrieve the missing information, returning the result to the user.

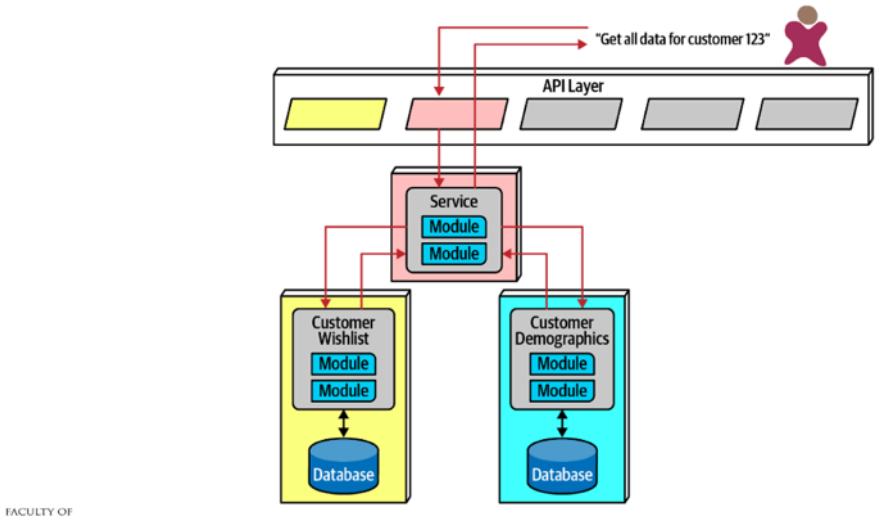


Figure 17-8. Using orchestration in microservices

Because microservices architectures don't include a global mediator like other service-oriented architectures, if an architect needs to coordinate across several services, they can create their own localized mediator, as shown in Figure 17-8.

In Figure 17-8, the developers create a service whose sole responsibility is coordinating the call to get all information for a particular customer. The user calls the ReportCustomerInformation mediator, which calls the necessary other services.

The First Law of Software Architecture suggests that neither of these solutions is perfect—each has trade-offs. In choreography, the architect preserves the highly decoupled philosophy of the architecture style, thus reaping maximum benefits touted by the style. However, common problems like error handling and coordination become more complex in choreographed environments.

Microservices: Transactions and Sagas

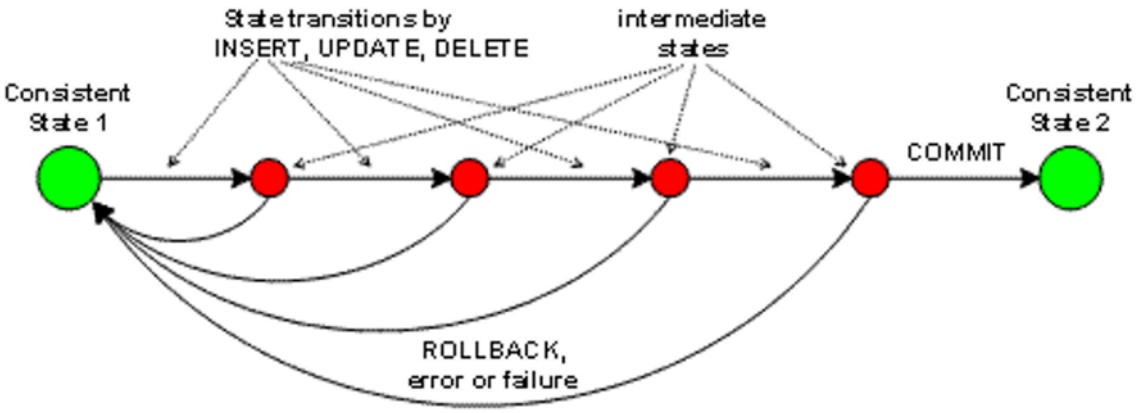


Image source: https://help.sap.com/doc/saphelp_nw70ehp1/7.01.4/ja-JP/42/006bd8c6084230e10000000a114cbd/content.htm?no_cache=true

Transactions can be defined as a block of activity that should be completed all or not at all. Such as in database, we can have a transaction where is some action is failed the transaction that has been committed will be rollbacked.

Saga is a pattern for distributed transaction. We will discuss it later.

Microservices: Transactions and Sagas

- What should we do if there is a transaction of several services?
- **Don't do transactions in microservices—fix granularity instead!**
- But some time, it happen -> Use the ***saga pattern***.

Architects aspire to extreme decoupling in microservices, but then often encounter the problem of how to do transactional coordination across services. Because the decoupling in the architecture encourages the same level for the databases, atomicity that was trivial in monolithic applications becomes a problem in distributed ones.
Building transactions across service boundaries violates the core decoupling principle of the microservices architecture (and also creates the worst kind of dynamic connascence, connascence of value). The best advice for architects who want to do transactions across services is: don't! Fix the granularity components instead. Often, architects who build microservices architectures who then find a need to wire them together with transactions have gone too granular in their design. Transaction boundaries is one of the common indicators of service granularity.
Don't do transactions in microservices—fix granularity instead!
Exceptions always exist. For example, a situation may arise where two different services need vastly different architecture characteristics, requiring distinct service boundaries, yet still need transactional coordination. In those situations, patterns exist to handle transaction orchestration, with serious trade-offs.
A popular distributed transactional pattern in microservices is the saga pattern, illustrated in Figure 17-12.

Notes:
Connascence is a metric, and like all metrics is an imperfect measure. However, connascence takes a more holistic approach, where each instance of connascence in a codebase must be considered on three separate axes:

1. Strength: Stronger connascences are harder to discover, or harder to refactor.
2. Degree: An entity that is connascent with thousands of other entities is likely to be a larger issue than one that is connascent with only a few.
3. Locality: Connascent elements that are close together in a codebase are better than ones that are far apart.

The three properties of Strength, Degree, and Locality give the programmer all the tools they need in order to make informed decisions about when they will permit certain types of coupling, and when the code ought to be refactored.

Microservices: Saga Pattern

- Use another service as mediator/orchestration.
- This mediator monitor the transaction and conduct the rollback.
- For example, if action 4 failed, it will signal (6) to the previous service to rollback its actions.

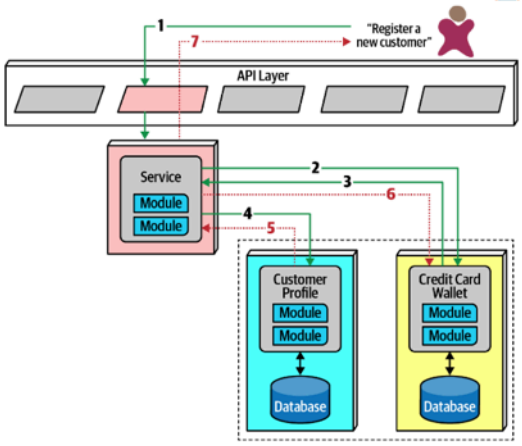


Figure 17-12. Saga pattern compensating transactions for error conditions

FACULTY OF COMPUTER SCIENCE

A service acts a mediator across multiple service calls and coordinates the transaction. The mediator calls each part of the transaction, records success or failure, and coordinates results. If everything goes as planned, all the values in the services and their contained databases update synchronously.

In an error condition, the mediator must ensure that no part of the transaction succeeds if one part fails. Consider the situation shown in Figure 17-12.

In Figure 17-12, if the first part of the transaction succeeds, yet the second part fails, the mediator must send a request to all the parts of the transaction that were successful and tell them to undo the previous request. This style of transactional coordination is called a compensating transaction framework. Developers implement this pattern by usually having each request from the mediator enter a pending state until the mediator indicates overall success. However, this design becomes complex if asynchronous requests must be juggled, especially if new requests appear that are contingent on pending transactional state. This also creates a lot of coordination traffic at the network level.

While it is possible for architects to build transactional behavior across services, it goes against the reason for choosing the microservices pattern. Exceptions always exist, so the best advice for architects is to use the saga pattern sparingly.

Introduction to Event-Driven Architectures Styles

Chapter 14, Mark Richards and Neal Ford, Fundamental of Software Architectures: An engineering approach, O'Reilly, 2021

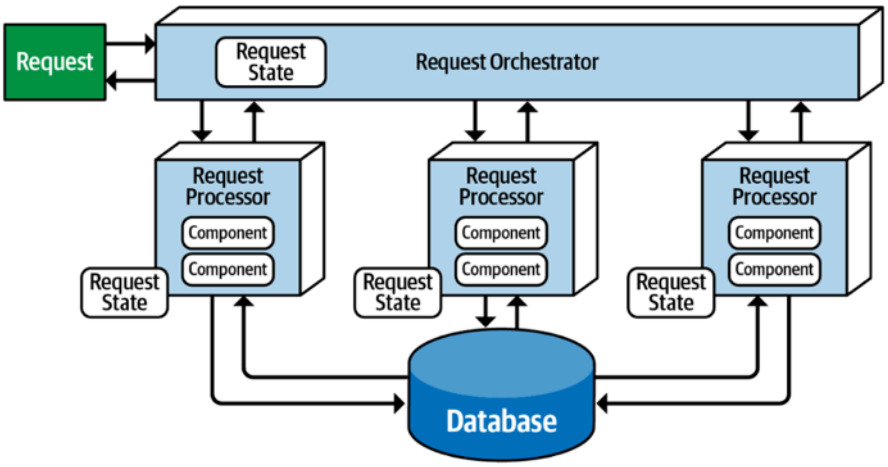
Overview: Event-Driven Architecture Style

- The event-driven architecture style is a popular distributed asynchronous architecture style used to produce highly scalable and high-performance applications.
- It is also highly adaptable and can be used for small applications and as well as large, complex ones.
- Event-driven architecture is made up of decoupled event processing components that asynchronously receive and process events.
- It can be used as a standalone architecture style or embedded within other architecture styles (such as an event-driven microservices architecture, we will discuss it later).

FACULTY OF COMPUTER SCIENCE

The event-driven architecture style is a popular distributed asynchronous architecture style used to produce highly scalable and high-performance applications. It is also highly adaptable and can be used for small applications and as well as large, complex ones. Event-driven architecture is made up of decoupled event processing components that asynchronously receive and process events. It can be used as a standalone architecture style or embedded within other architecture styles (such as an event-driven microservices architecture).

Overview: Request-based model



FACULTY OF COMPUTER SCIENCE

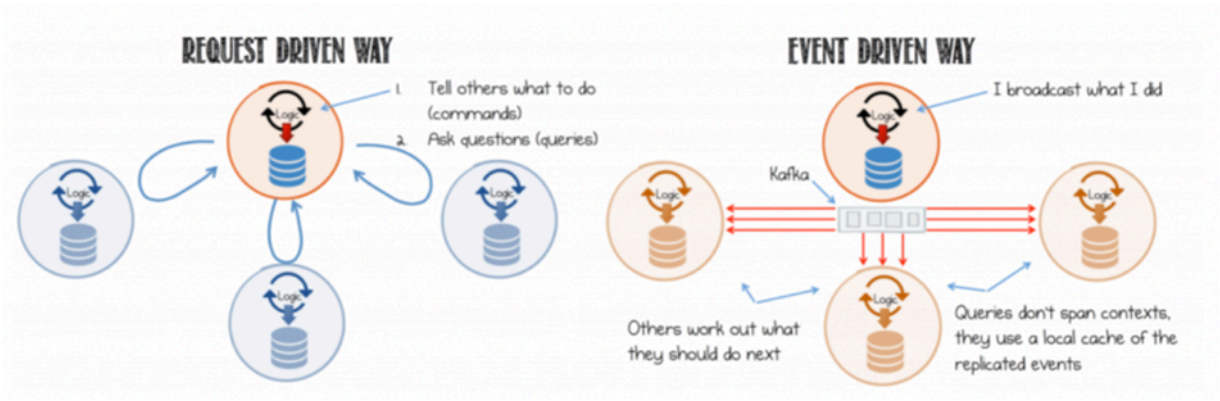
Figure 14-1. Request-based model

Most applications follow what is called a request-based model (illustrated in Figure 14-1). In this model, requests made to the system to perform some sort of action are send to a request orchestrator. The request orchestrator is typically a user interface, but it can also be implemented through an API layer or enterprise service bus. The role of the request orchestrator is to deterministically and synchronously direct the request to various request processors. The request processors handle the request, either retrieving or updating information in a database.

A good example of the request-based model is a request from a customer to retrieve their order history for the past six months. Retrieving order history information is a data-driven, deterministic request made to the system for data within a specific context, not an event happening that the system must react to.

An event-based model, on the other hand, reacts to a particular situation and takes action based on that event. An example of an event-based model is submitting a bid for a particular item within an online auction. Submitting the bid is not a request made to the system, but rather an event that happens after the current asking price is announced. The system must respond to this event by comparing the bid to others received at the same time to determine who is the current highest bidder.

Request-based vs Event-based



FACULTY OF COMPUTER SCIENCE

Image source: https://indatawetrust.blog/2019/04/30/from-monolithic-architecture-to-microservices-and-event-driven-systems/

This is the main differences. In Request-based, every request is synchronous and it will wait for answer from the system. While in event driven, each event is just spawn, and let the one who interested to behave as it should. The client don't need to wait. It is some time an asynchronous call or it possibly don't need any reply (fire and forget). It will speed up the performance and reduce the network bandwidth significantly.

Kafka is distributed event streaming platform. The complete name is Apache Kafka. It is in the same variant with RabbitMQ. But those are having different motivation in building it. They are designed for different use cases. So they have differences in handling event.

Event-Driven: Topology

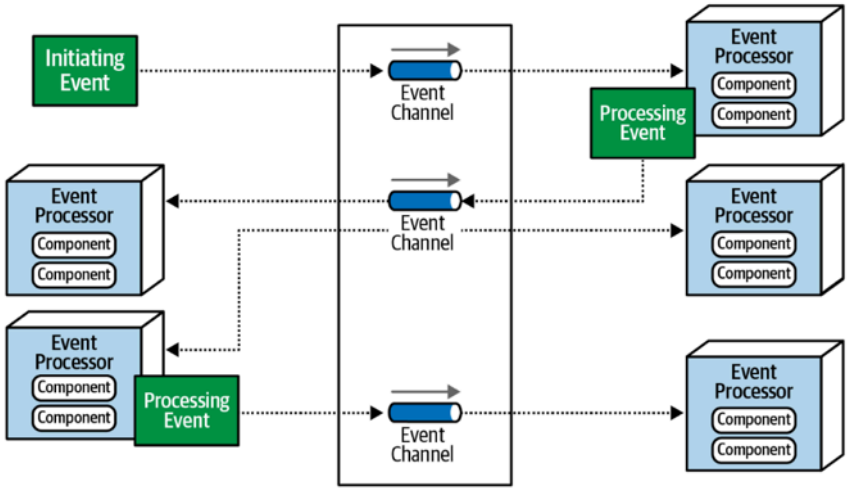
- Two primary topologies :
  - ***Mediator topology*** is commonly used when you **require control over the workflow** of an event process.
  - ***Broker topology*** is used when you **require a high degree of responsiveness and dynamic control** over the processing of an event

FACULTY OF COMPUTER SCIENCE

There are two primary topologies within event-driven architecture: the mediator topology and the broker topology. The mediator topology is commonly used when you require control over the workflow of an event process, whereas the broker topology is used when you require a high degree of responsiveness and dynamic control over the processing of an event. Because the architecture characteristics and implementation strategies differ between these two topologies, it is important to understand each one to know which is best suited for a particular situation.

Broker Topology

Event-Driven: Broker Topology



FACULTY OF COMPUTER SCIENCE

Figure 14-2. Broker topology

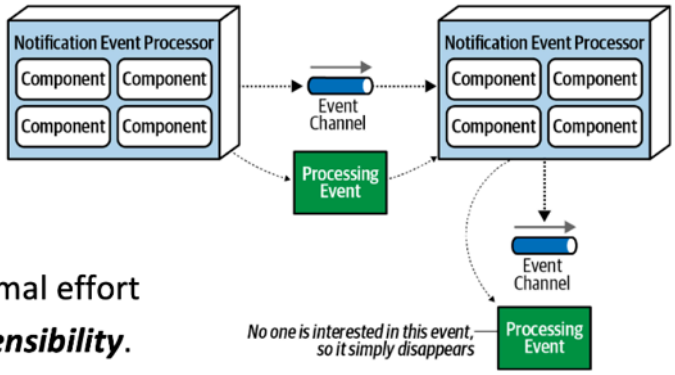
The broker topology differs from the mediator topology in that there is no central event mediator. Rather, the message flow is distributed across the event processor components in a chain-like broadcasting fashion through a lightweight message broker (such as RabbitMQ, ActiveMQ, HornetQ, and so on). This topology is useful when you have a relatively simple event processing flow and you do not need central event orchestration and coordination.

There are four primary architecture components within the broker topology: an initiating event, the event broker, an event processor, and a processing event. The initiating event is the initial event that starts the entire event flow, whether it be a simple event like placing a bid in an online auction or more complex events in a health benefits system like changing a job or getting married. The initiating event is sent to an event channel in the event broker for processing. Since there is no mediator component in the broker topology managing and controlling the event, a single event processor accepts the initiating event from the event broker and begins the processing of that event. The event processor that accepted the initiating event performs a specific task associated with the processing of that event, then asynchronously advertises what it did to the rest of the system by creating what is called a processing event. This processing event is then asynchronously sent to the event broker for further processing, if needed. Other event processors listen to the processing event, react to that event by doing something, then advertise through a new processing event what they did. This process continues until no one is interested in what a final event processor did. Figure 14-2 illustrates this event processing flow.

The event broker component is usually federated (meaning multiple domain-based clustered instances), where each federated broker contains all of the event channels used within the event flow for that particular domain. Because of the decoupled asynchronous fire-and-forget broadcasting nature of the broker topology, topics (or topic exchanges in the case of AMQP) are usually used in the broker topology using a publish-and-subscribe messaging model.

Event-Driven: Architectural Extensibility

- Email notification is generated. It is sent to a channel
- But it is ignored.
  Fire and forget.
- Suppose that the system grows
- Additional requirement to analyze the email.
- We can just add it with minimal effort
- It is called ***Architectural Extensibility***.



FACULTY OF COMPUTER SCIENCE

Figure 14-3. Notification event is sent but ignored

It is always a good practice within the broker topology for each event processor to advertise what it did to the rest of the system, regardless of whether or not any other event processor cares about what that action was. This practice provides architectural extensibility if additional functionality

is required for the processing of that event. For example, suppose as part of a complex event process, as illustrated in Figure 14-3, an email is generated and sent to a customer notifying them of a particular action taken. The Notification event processor would generate and send the email, then advertise that action to the rest of the system through a new processing event sent to a topic. However, in this case, no other event processors are listening for events on that topic, and as such the message simply goes away.

This is a good example of architectural extensibility. While it may seem like a waste of resources sending messages that are ignored, it is not. Suppose a new requirement comes along to analyze emails that have been sent to customers. This new event processor can be added to the overall system with minimal effort because the email information is available via the email topic to the new analyzer without having to add any additional infrastructure or apply any changes to other event processors.

Broker topology

- An event of order is placed
- Order Placement takes the event
- It processed and sent a order-created
- This order-created is taken by Notification, Payment and Inventory.
- Inventory creates an inventory-update event
- Payment creates payment-denied or payment applied event.
- And so on....

FACULTY OF COMPUTER SCIENCE

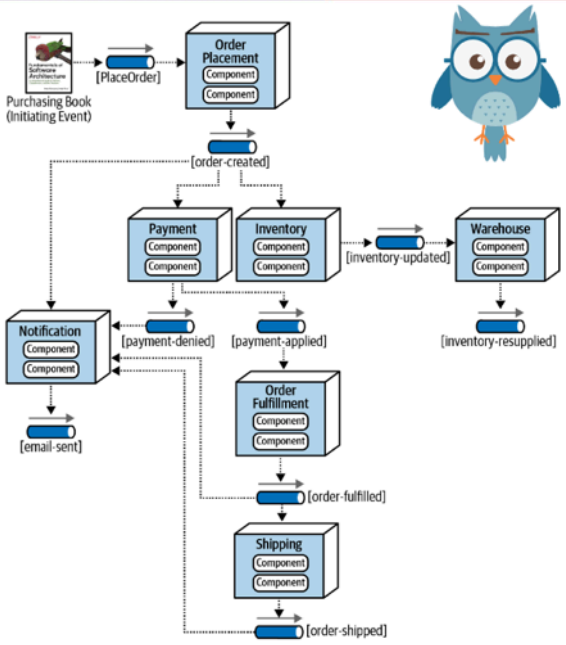


Figure 14-4. Example of the broker topology

To illustrate how the broker topology works, consider the processing flow in a typical retail order entry system, as illustrated in Figure 14-4, where an order is placed for an item (say, a book like this one). In this example, the OrderPlacement event processor receives the initiating event (PlaceOrder), inserts the order in a database table, and returns an order ID to the customer. It then advertises to the rest of the system that it created an order through an order-created processing event. Notice that three event processors are interested in that event: the Notification event processor, the Payment event processor, and the Inventory event processor. All three of these event processors perform their tasks in parallel.

The Notification event processor receives the order-created processing event and emails the customer. It then generates another processing event (email-sent). Notice that no other event processors are listening to that event. This is normal and illustrates the previous example describing architectural extensibility—an in-place hook so that other event processors can eventually tap into that event feed, if needed. The Inventory event processor also listens for the order-created processing event and decrements the corresponding inventory for that book. It then advertises this action through an inventory-updated processing event, which is in turn picked up

by the Warehouse event processor to manage the corresponding inventory between warehouses, reordering items if supplies get too low.
The Payment event processor also receives the order-created processing event and charges the customer's credit card for the order that was just created. Notice in Figure 14-4 that two events are generated as a result of the actions taken by the Payment event processor: one to notify the rest of the system that the payment was applied (payment-applied) and one processing event to notify the rest of the system that the payment was denied (payment-denied). Notice that the Notification event processor is interested in the payment-denied processing event, because it must, in turn, send an email to the customer informing them that they must update their credit card information or choose a different payment method.
The OrderFulfillment event processor listens to the payment-applied processing event and does order picking and packing. Once completed, it then advertises to the rest of the system that it fulfilled the order via an order-fulfilled processing event. Notice that both the Notification processing unit and the Shipping processing unit listen to this processing event. Concurrently, the Notification event processor notifies the customer that the order has been fulfilled and is ready for shipment, and at the same time the Shipping event processor selects a shipping method. The Shipping event processor ships the order and sends out an order-shipped processing event, which the Notification event processor also listens for to notify the customer of the order status change.

Analyzing previous example

- All of the event processors are highly decoupled and independent of each other.
- Once an event processor hands off the event, it is no longer involved with the processing of that specific event and is available to react to other initiating or processing events.
- Each event processor can scale independently from one other to handle varying load conditions or backups in the processing within that event.
- No control over the overall workflow.
- Error handling is also a big challenge with the broker topology.
- The ability to restart a business transaction (recoverability) is also something not supported with the broker topology.

FACULTY OF COMPUTER SCIENCE

In analyzing the prior example, notice that all of the event processors are highly decoupled and independent of each other. The best way to understand the broker topology is to think about it as a relay race. In a relay race, runners hold a baton (a wooden stick) and run for a certain distance (say 1.5 kilometers), then hand off the baton to the next runner, and so on down the chain until the last runner crosses the finish line. In relay races, once a runner hands off the baton, that runner is done with the race and moves on to other things. This is also true with the broker topology. Once an event processor hands off the event, it is no longer involved with the processing of that specific event and is available to react to other initiating or processing events. In addition, each event processor can scale independently from one other to handle varying load

conditions or backups in the processing within that event. The topics provide the back pressure point if an event processor comes down or slows down due to some environment issue.

Trade-offs of the broker topology

Advantages	Disadvantages
Highly decoupled event processors	Workflow control
High scalability	Error handling
High responsiveness	Recoverability
High performance	Restart capabilities
High fault tolerance	Data inconsistency

FACULTY OF COMPUTER SCIENCE

While performance, responsiveness, and scalability are all great benefits of the broker topology, there are also some negatives about it. First of all, there is no control over the overall workflow associated with the initiating event (in this case, the PlaceOrder event). It is very dynamic based on various conditions, and no one in the system really knows when the business transaction of placing an order is actually complete. Error handling is also a big challenge with the broker topology. Because there is no mediator monitoring or controlling the business transaction, if a failure occurs (such as the Payment event processor crashing and not completing its assigned task), no one in the system is aware of that crash. The business process gets stuck and is unable to move without some sort of automated or manual intervention. Furthermore, all other processes are moving along without regard for the error. For example, the Inventory event processor still decrements the inventory, and all other event processors react as though everything is fine.
The ability to restart a business transaction (recoverability) is also something not supported with the broker topology. Because other actions have asynchronously been taken through the initial processing of the initiating event, it is not possible to resubmit the initiating event. No component in the broker topology is aware of the state or even owns the state of the original business request, and therefore no one is responsible in this topology for restarting the business transaction (the initiating event) and knowing where it left off.

Mediator Topology

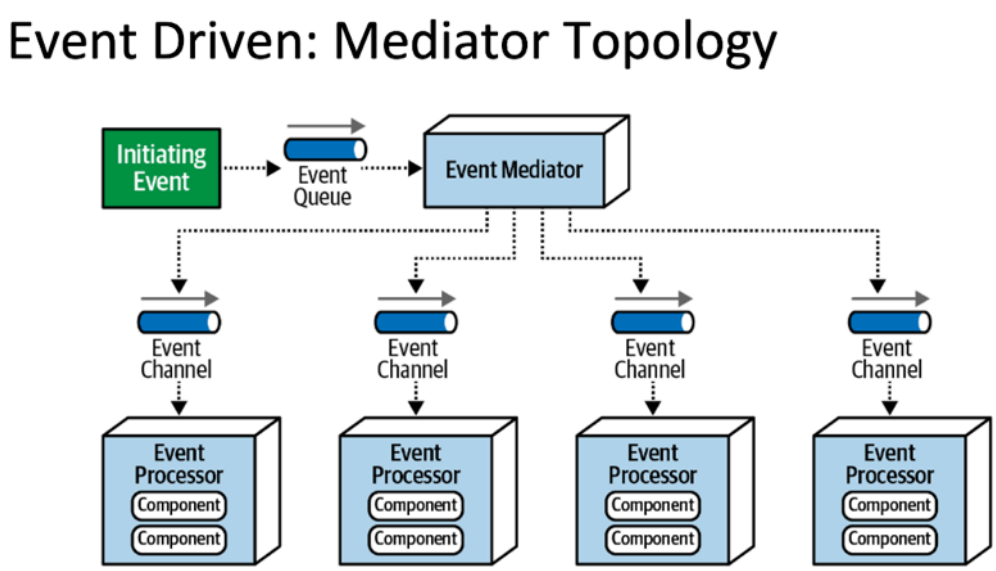


Figure 14-5. Mediator topology

The mediator topology of event-driven architecture addresses some of the shortcomings of the broker topology described in the previous section. Central to this topology is an event mediator, which manages and controls the workflow for initiating events that require the coordination of multiple event processors. The architecture components that make up the mediator topology are an initiating event, an event queue, an event mediator, event channels, and event processors. Like in the broker topology, the initiating event is the event that starts the whole eventing process. Unlike the broker topology, the initiating event is sent to an initiating event queue, which is accepted by the event mediator. The event mediator only knows the steps involved in processing the event and therefore generates corresponding processing events that are sent to dedicated event channels (usually queues) in a point-to-point messaging fashion. Event processors then listen to dedicated event channels, process the event, and usually respond back to the mediator that they have completed their work. Unlike the broker topology, event processors within the mediator topology do not advertise what they did to the rest of the system. The mediator topology is illustrated in Figure 14-5.

Event-Driven: Event Mediator

- It control how initiate event will be handled.
- It control the overall workflow
- Example of mediator: Apache Camel, Mule ESB, Spring Integration
- How the messages flow and routes typically custom written in programming code.
- If the workflow is more complicated, the use of Business Process Execution Language (BPEL) could help to simplify. Example of tools: Apache ODE, Oracle BPEL Process Manager

FACULTY OF COMPUTER SCIENCE

The event mediator can be implemented in a variety of ways, depending on the nature and complexity of the events it is processing. For example, for events requiring simple error handling and orchestration, a mediator such as Apache Camel, Mule ESB, or Spring Integration will usually suffice. Message flows and message routes within these types of mediators are typically custom written in programming code (such as Java or C#) to control the workflow of the event processing.

However, if the event workflow requires lots of conditional processing and multiple dynamic paths with complex error handling directives, then a mediator such as Apache ODE or the Oracle BPEL Process Manager would be a good choice. These mediators are based on Business Process Execution Language (BPEL), an XML-like structure that describes the steps involved in processing an event. BPEL artifacts also contain structured elements used for error handling, redirection, multicasting, and so on. BPEL is a powerful but relatively complex language to learn, and as such is usually created using graphical interface tools provided in the product's BPEL engine suite.

BPEL is good for complex and dynamic workflows, but it does not work well for those event workflows requiring long-running transactions involving human intervention throughout the event process. For example, suppose a trade is being placed through a place-trade initiating event. The event mediator accepts this event, but during the processing finds that a manual approval is required because the trade is over a certain amount of shares. In this case the event mediator would have to stop the event processing, send a notification to a senior trader for the manual approval, and wait for that approval to occur. In these cases a Business Process Management (BPM) engine such as jBPM would be required

Event-Driven: Delegating Event Mediator

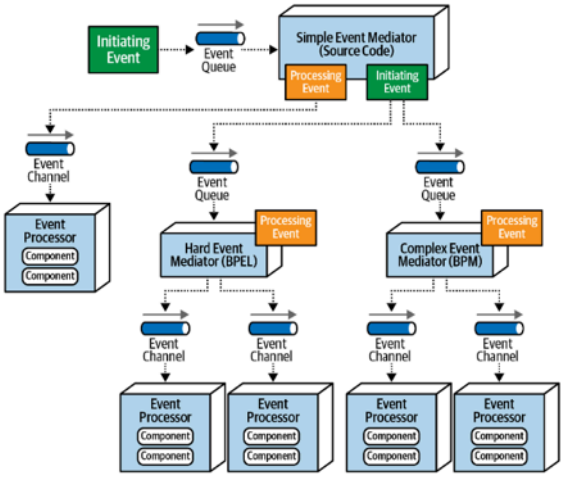


Figure 14-6. Delegating the event to the appropriate type of event mediator

Given that it's rare to have all events of one class of complexity, we recommend classifying events as simple, hard, or complex and having every event always go through a simple mediator (such as Apache Camel or Mule). The simple mediator can then interrogate the classification of the event, and based on that classification, handle the event itself or forward it to another, more complex, event mediator. In this manner, all types of events can be effectively processed by the type of mediator needed for that event. This mediator delegation model is illustrated in Figure 14-6.

Notice in Figure 14-6 that the Simple Event Mediator generates and sends a processing event when the event workflow is simple and can be handled by the simple mediator. However, notice that when the initiating event coming into the Simple Event Mediator is classified as either hard or complex, it forwards the original initiating event to the corresponding mediators (BPEL or BMP). The Simple Event Mediator, having intercepted the original event, may still be responsible for knowing when that event is complete, or it simply delegates the entire workflow (including client notification) to the other mediators.

Mediator Topology

FACULTY OF COMPUTER SCIENCE

Figure 14-7. Mediator steps for placing an order

To illustrate how the mediator topology works, consider the same retail order entry system example described in the prior broker topology section, but this time using the mediator topology. In this example, the mediator knows the steps required to process this particular event. This event flow (internal to the mediator component) is illustrated in Figure 14-7.

In keeping with the prior example, the same initiating event (PlaceOrder) is sent to the customer-event-queue for processing. The Customer mediator picks up this initiating event and begins generating processing events based on the flow in Figure 14-7. Notice that the multiple events shown in steps 2, 3, and 4 are all done concurrently and serially between steps. In other words, step 3 (fulfill order) must be completed and acknowledged before the customer can be notified that the order is ready to be shipped in step 4 (ship order).

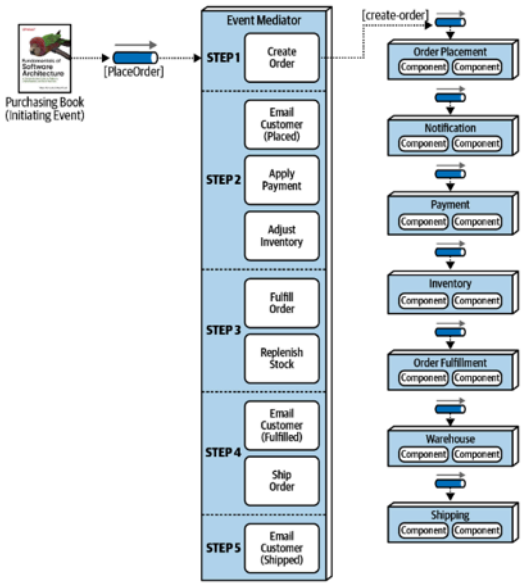


Figure 14-8. Step 1 of the mediator example

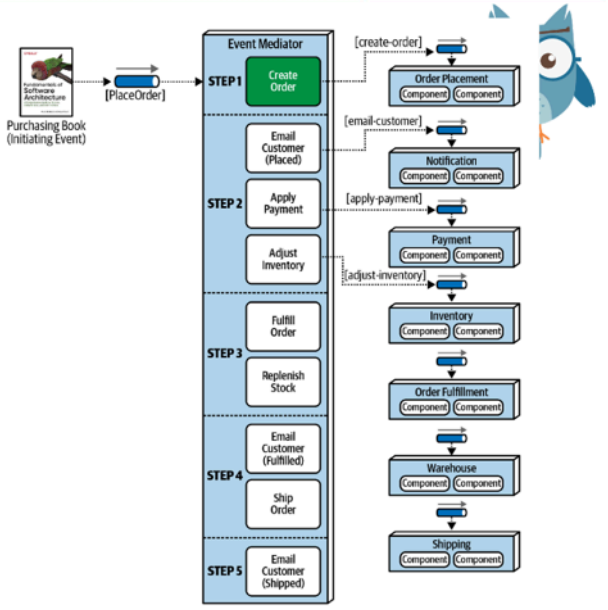


Figure 14-9. Step 2 of the mediator example

Once the initiating event has been received, the Customer mediator generates a create-order processing event and sends this message to the order-placement-queue (see Figure 14-8). The OrderPlacement event processor accepts this event and validates and creates the order, returning to the mediator an acknowledgement along with the order ID. At this point the mediator might send that order ID back to the customer, indicating that the order was placed, or it might have to continue until all the steps are complete (this would be based on specific business rules about order placement).

Now that step 1 is complete, the mediator now moves to step 2 (see Figure 14-9) and generates three messages at the same time: email-customer, apply-payment, and adjust-inventory. These processing events are all sent to their respective queues. All three event processors receive these messages, perform their respective tasks, and notify the mediator that the processing has been completed. Notice that the mediator must wait until it receives acknowledgement from all three parallel processes before moving on to step 3. At this point, if an error occurs in one of the parallel event processors, the mediator can take corrective action to fix the problem (this is discussed later in this section in more detail).

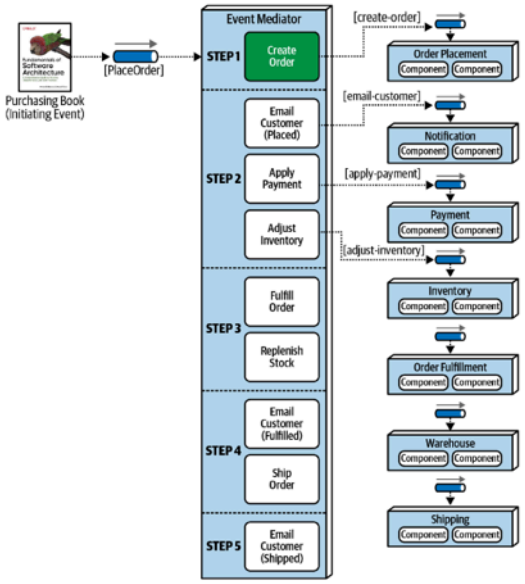


Figure 14-9. Step 2 of the mediator example

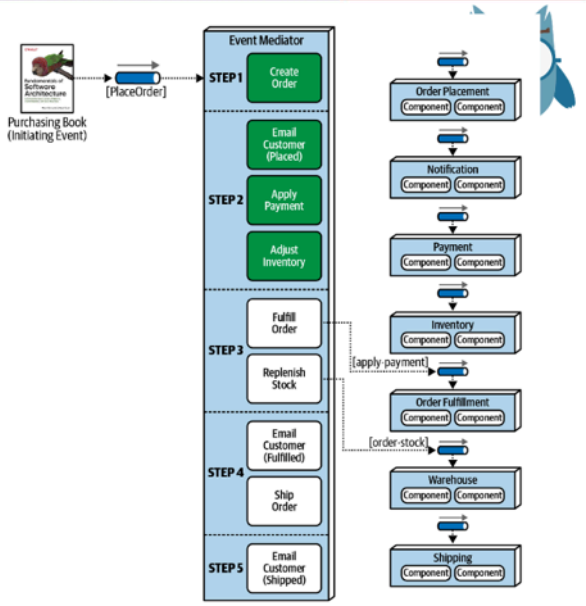


Figure 14-10. Step 3 of the mediator example

SCIENCE

Once the mediator gets a successful acknowledgment from all of the event processors in step 2, it can move on to step 3 to fulfill the order (see Figure 14-10). Notice once again that both of these events (fulfill-order and order-stock) can occur simultaneously. The OrderFulfillment and Warehouse event processors accept these events, perform their work, and return an acknowledgement to the mediator.

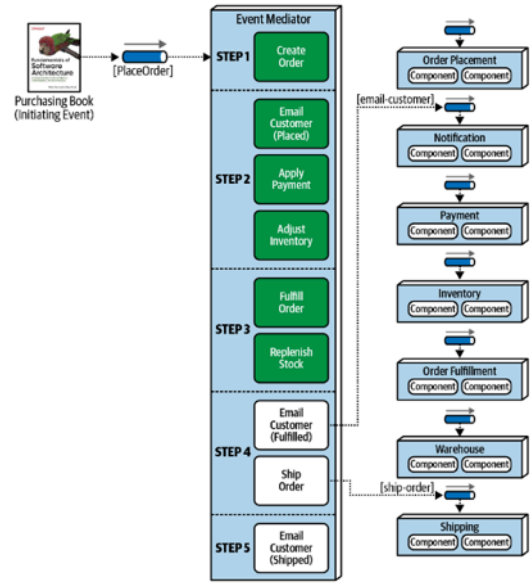


Figure 14-11. Step 4 of the mediator example

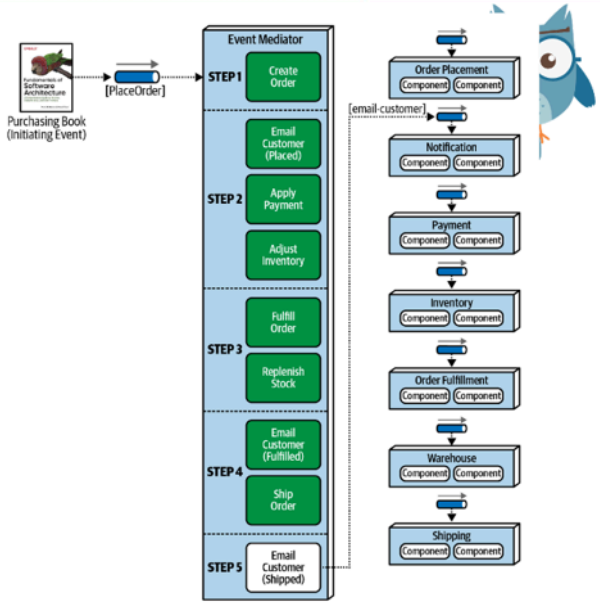


Figure 14-12. Step 5 of the mediator example

SCIENCE

Once these events are complete, the mediator then moves on to step 4 (see Figure 14-11) to ship the order. This step generates another email-customer processing event with specific information

about what to do (in this case, notify the customer that the order is ready to be shipped), as well as a ship-order event.
Finally, the mediator moves to step 5 (see Figure 14-12) and generates another contextual email-customer event to notify the customer that the order has been shipped. At this point the workflow is done, and the mediator marks the initiating event flow complete and removes all state associated with the initiating event.

Trade-offs of the mediator topology

Advantages	Disadvantages
Workflow control	More coupling of event processors
Error handling	Lower scalability
Recoverability	Lower performance
Restart capabilities	Lower fault tolerance
Better data consistency	Modeling complex workflows

FACULTY OF COMPUTER SCIENCE

The mediator component has knowledge and control over the workflow, something the broker topology does not have. Because the mediator controls the workflow, it can maintain event state and manage error handling, recoverability, and restart capabilities. For example, suppose in the prior example the payment was not applied due to the credit card being expired. In this case the mediator receives this error condition, and knowing the order cannot be fulfilled (step 3) until payment is applied, stops the workflow and records the state of the request in its own persistent datastore. Once payment is eventually applied, the workflow can be restarted from where it left off (in this case, the beginning of step 3).
Another inherent difference between the broker and mediator topology is how the processing events differ in terms of their meaning and how they are used. In the broker topology example in the previous section, the processing events were published as events that had occurred in the system (such as order-created, payment-applied, and email-sent). The event processors took some action, and other event processors react to that action. However, in the mediator topology, processing occurrences such as place-order, send-email, and fulfill-order are commands (things that need to happen) as opposed to events (things that have already happened). Also, in the mediator topology, a command must be processed, whereas an event can be ignored in the broker topology.
While the mediator topology addresses the issues associated with the broker topology, there are some negatives associated with the mediator topology. First of all, it is very difficult to declaratively model the dynamic processing that occurs within a complex event flow. As a result, many workflows within the mediator only handle the general processing, and a hybrid model

combining both the mediator and broker topologies is used to address the dynamic nature of complex event processing (such as out-of-stock conditions or other nontypical errors). Furthermore, although the event processors can easily scale in the same manner as the broker topology, the mediator must scale as well, something that occasionally produces a bottleneck in the overall event processing flow. Finally, event processors are not as highly decoupled in the mediator topology as with the broker topology, and performance is not as good due to the mediator controlling the processing of the event.

Asynchronous and Broadcast

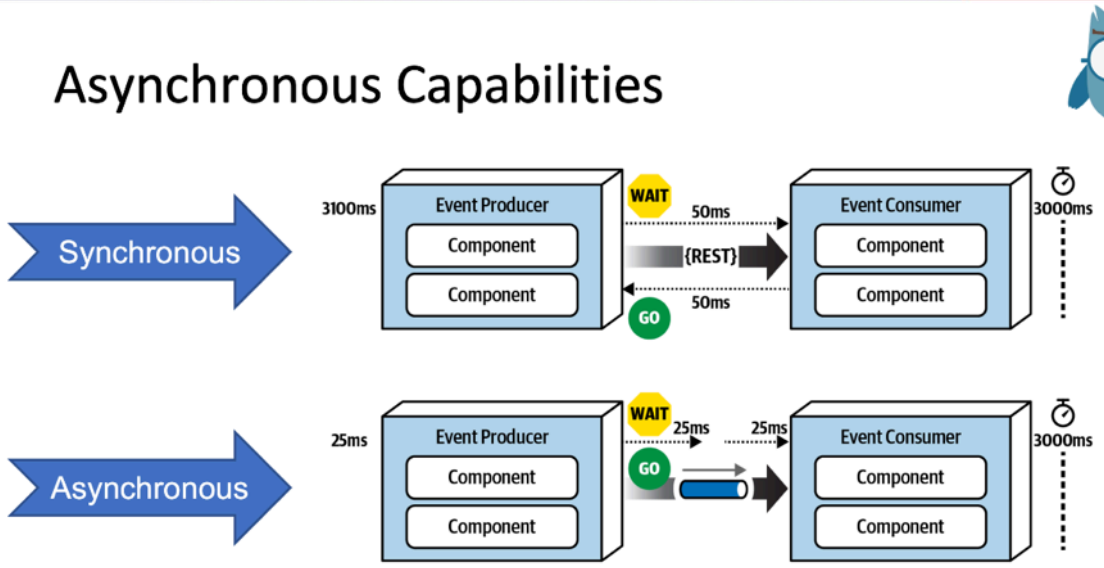


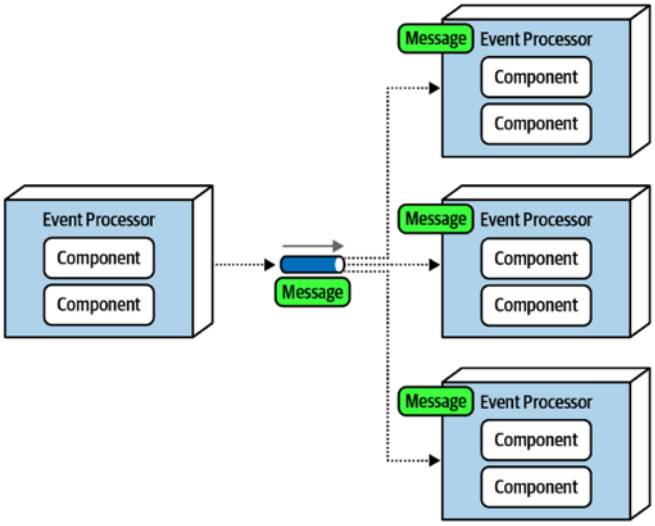
Figure 14-13. Synchronous versus asynchronous communication

FACULTY OF COMPUTER SCIENCE

The event-driven architecture style offers a unique characteristic over other architecture styles in that it relies solely on asynchronous communication for both fire-and-forget processing (no response required) as well as request/reply processing (response required from the event consumer). Asynchronous communication can be a powerful technique for increasing the overall responsiveness of a system.

We will discuss more detail about Asynchronous in the next module.

Broadcast Capabilities



FACULTY OF COMPUTER SCIENCE

One of the other unique characteristics of event-driven architecture is the capability to broadcast events without knowledge of who (if anyone) is receiving the message and what they do with it. This technique, which is illustrated in Figure 14-18, shows that when a producer publishes a message, that same message is received by multiple subscribers.

Broadcasting is perhaps the highest level of decoupling between event processors because the producer of the broadcast message usually does not know which event processors will be receiving the broadcast message and more importantly, what they will do with the message. Broadcast capabilities are an essential part of patterns for eventual consistency, complex event processing (CEP), and a host of other situations. Consider frequent changes in stock prices for instruments traded on the stock market. Every ticker (the current price of a particular stock) might influence a number of things. However, the service publishing the latest price simply broadcasts it with no knowledge of how that information will be used.

Request and Reply

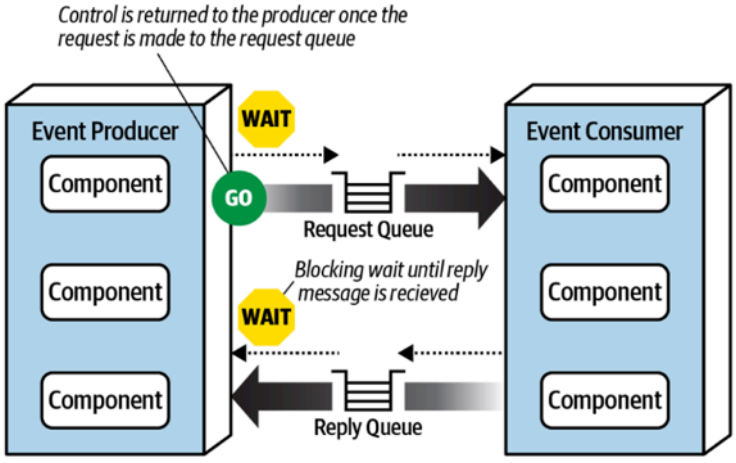


Figure 14-19. Request-reply message processing

So far in this chapter we've dealt with asynchronous requests that don't need an immediate response from the event consumer. But what if an order ID is needed when ordering a book? What if a confirmation number is needed when booking a flight? These are examples of communication between services or event processors that require some sort of synchronous communication.

In event-driven architecture, synchronous communication is accomplished through request-reply messaging (sometimes referred to as pseudosynchronous communications). Each event channel within request-reply messaging consists of two queues: a request queue and a reply queue. The initial request for information is asynchronously sent to the request queue, and then control is returned to the message producer. The message producer then does a blocking wait on the reply queue, waiting for the response. The message consumer receives and processes the message and then sends the response to the reply queue. The event producer then receives the message with the response data.

Request and Reply (using message ID)

1. The event producer sends a message to the request queue and records the unique message ID (in this case ID 124).
2. The event producer now does a blocking wait on the reply queue with a message filter (also called a message selector).
3. The event consumer receives the message (ID 124) and processes the request.

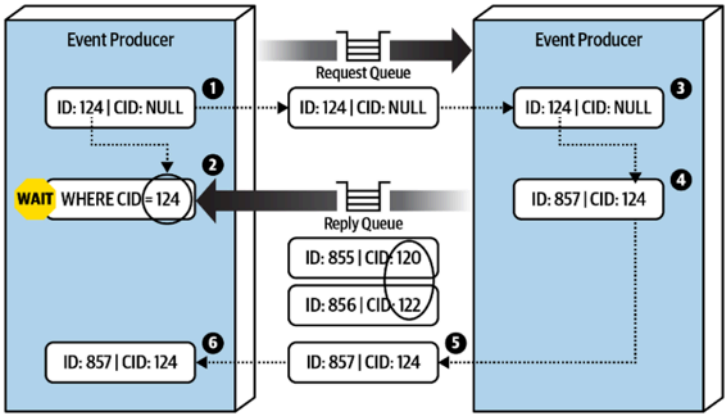


Figure 14-20. Request-reply message processing using a correlation ID

FACULTY OF COMPUTER SCIENCE

This technique, as illustrated in Figure 14-20, works as follows, with the message ID indicated with ID, and the correlation ID indicated with CID:

1. The event producer sends a message to the request queue and records the unique message ID (in this case ID 124). Notice that the correlation ID (CID) in this case is null.
2. The event producer now does a blocking wait on the reply queue with a message filter (also called a message selector), where the correlation ID in the message header equals the original message ID (in this case 124). Notice there are two messages in the reply queue: message ID 855 with correlation ID 120, and message ID 856 with correlation ID 122. Neither of these messages will be picked up because the correlation ID does not match what the event consumer is looking for (CID 124).
3. The event consumer receives the message (ID 124) and processes the request.

Request and Reply (using message ID)

4. The event consumer creates the reply message containing the response and sets the correlation ID (CID).
5. The event consumer sends the new message (ID 857) to the reply queue.
6. The event producer receives the message because the correlation ID (124) matches the message selector from step 2.

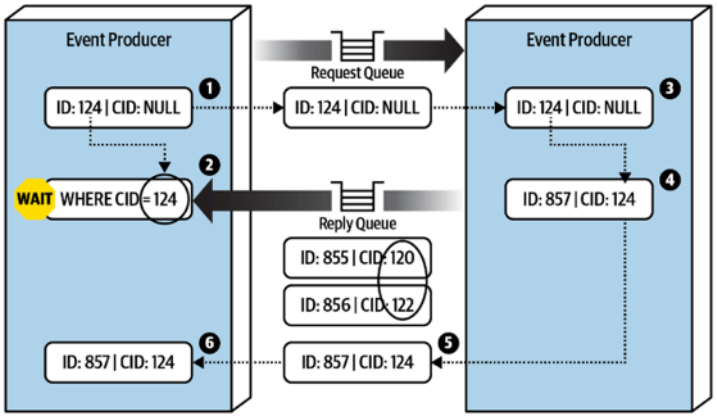


Figure 14-20. Request-reply message processing using a correlation ID

FACULTY OF COMPUTER SCIENCE

4. The event consumer creates the reply message containing the response and sets the correlation ID (CID) in the message header to the original message ID (124).
5. The event consumer sends the new message (ID 857) to the reply queue.
6. The event producer receives the message because the correlation ID (124) matches the message selector from step 2.

There is other technique called temporary queue, we will not discuss it here. Interested reader can refer to the book.

Event-Driven Microservice Architecture

Chapter 1, Building Event-Driven Microservices by Adam Bellemare, O'Reilly, 2020

https://www.infoworld.com/article/3694133/how-to-get-started-with-event-driven-microservices.html

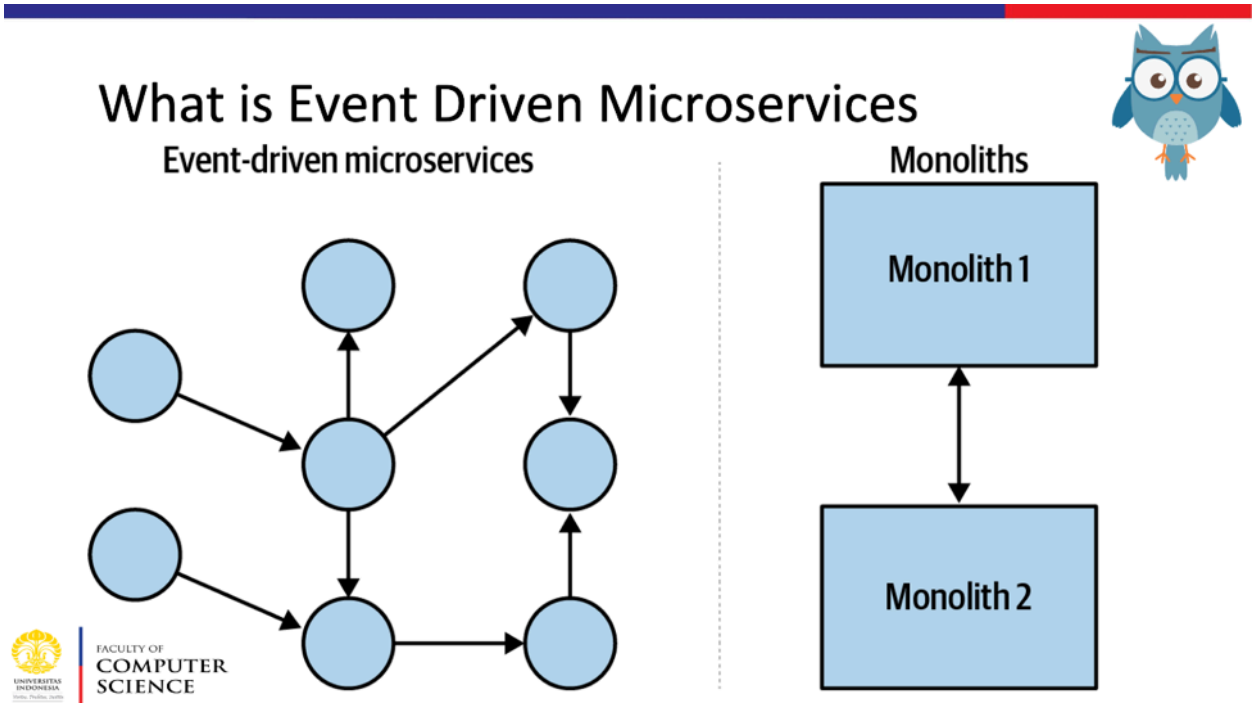
Why Event-Driven Microservices

- Event-driven provide a powerful, flexible way to meet today's requirements.
- Meanwhile, microservices provide the flexibility and choice to use the right tools for solving business problems.
- Both, Event-driven microservices are an excellent way to deliver both historical and new data to all of the systems and teams that need it, but they come with additional overhead and management requirements.

FACULTY OF COMPUTER SCIENCE

Why Event-Driven Microservices?
The medium is the message.
Marshall McLuhan
McLuhan argues that it is not the *content* of media, but rather engagement with its medium, that impacts humankind and introduces fundamental changes to society. Newspapers, radio,

television, the internet, instant messaging, and social media have all changed human interaction and social structures thanks to our collective engagement.
The same is true with computer system architectures. You need only look at the history of computing inventions to see how network communications, relational databases, big-data developments, and cloud computing have significantly altered how architectures are built and how work is performed. Each of these inventions changed not only the way that technology was used within various software projects, but also the way that organizations, teams, and people communicated with one another. From centralized mainframes to distributed mobile applications, each new medium has fundamentally changed people's relationship with computing.
The medium of the asynchronously produced and consumed event has been fundamentally shifted by modern technology. These events can now be persisted indefinitely, at extremely large scale, and be consumed by any service as many times as necessary. Compute resources can be easily acquired and released on-demand, enabling the easy creation and management of microservices. Microservices can store and manage their data according to their own needs, and do so at a scale that was previously limited to batch-based big-data solutions. These improvements to the humble and simple event-driven medium have far-reaching impacts that not only change computer architectures, but also completely reshape how teams, people, and organizations create systems and businesses.



Microservices and microservice-style architectures have existed for many years, in many different forms, under many different names. Service-oriented architectures (SOAs) are often composed of multiple microservices synchronously communicating directly with one another. Message-passing architectures use consumable events to asynchronously communicate with one another. Event-based communication is certainly not new, but the need for handling big data sets, at scale and in real time, *is* new and necessitates a change from the old architectural styles.
In a modern event-driven microservices architecture, systems communicate by issuing and consuming events. These events are not destroyed upon consumption as in message-passing

systems, but instead remain readily available for other consumers to read as they require. This is an important distinction, as it allows for the truly powerful patterns covered in this book.
The services themselves are small and purpose-built, created to help fulfill the necessary business goals of the organization. A typical definition of "small" is something that takes no more than two weeks to write. By another definition, the service should be able to (conceptually) fit within one's own head. These services consume events from input event streams; apply their specific business logic; and may emit their own output events, provide data for request-response access, communicate with a third-party API, or perform other required actions. As this book will detail, these services can be stateful or stateless, complex or simple; and they might be implemented as long-running, standalone applications or executed as a function using Functions-as-a-Service.
This combination of event streams and microservices forms an interconnected graph of activity across a business organization. Traditional computer architectures, composed of monoliths and intermonolith communications, have a similar graph structure.

More on Domain Driven and Bounded Context

- To really understand how to apply Event-Driven Microservices, we have to get deeper into Domain Driven Design and Bounded Context.
- These concepts are the underlying concepts to understand how to build event-driven microservices.

FACULTY OF COMPUTER SCIENCE

Domain-driven design, as coined by Eric Evans in his book of the same title, introduces some of the necessary concepts for building event-driven microservices.

Domain-Driven Design

- ***Domain***: The problem space that a business occupies and provides solutions to
- ***Subdomain***: A component of the main domain. Each subdomain focuses on a specific subset of responsibilities and typically reflects some of the business's organizational structure (such as Warehouse, Sales, and Engineering).
- ***Domain (and subdomain) model***: An abstraction of the actual domain useful for business purposes.
- ***Bounded context***: The logical boundaries, including the inputs, outputs, events, requirements, processes, and data models, relevant to the subdomain.

FACULTY OF COMPUTER SCIENCE

Domain: The problem space that a business occupies and provides solutions to. This encompasses everything that the business must contend with, including rules, processes, ideas, business-specific terminology, and anything related to its problem space, *regardless of whether or not the business concerns itself with it*. The domain exists regardless of the existence of the business.
Subdomain: A component of the main domain. Each subdomain focuses on a specific subset of responsibilities and typically reflects some of the business's organizational structure (such as Warehouse, Sales, and Engineering). A subdomain can be seen as a domain in its own right. Subdomains, like the domain itself, belong to the problem space.
Domain (and subdomain) model: An abstraction of the actual domain useful for business purposes. The pieces and properties of the domain that are most important to the business are used to generate the model. The main domain model of an business is discernible through the products the business provides its customers, the interfaces by which customers interact with the products, and the various other processes and functions by which the business fulfils its stated goals. Models often need to be refined as the domain changes and as business priorities shift. A domain model is part of the solution space, as it is a construct the business uses to solve problems.
Bounded context: The logical boundaries, including the inputs, outputs, events, requirements, processes, and data models, relevant to the subdomain. While ideally a bounded context and a subdomain will be in complete alignment, legacy systems, technical debt, and third-party integrations often create exceptions. Bounded contexts are also a property of the solution space and have a significant impact on how microservices interact with one another.
Bounded contexts should be highly cohesive. The internal operations of the context should be intensive and closely related, with the vast majority of communication occurring internally rather than cross-boundary. Having highly cohesive responsibilities allows for reduced design scope and simpler implementations.

Connections between bounded contexts should be loosely coupled, as changes made within one bounded context should minimize or eliminate the impact on neighbouring contexts. A loose coupling can ensure that requirement changes in one context do not propagate a surge of dependent changes to neighbouring contexts.
Every organization forms a single domain between itself and the outside world. Everyone working within the organization is operating to support the needs of its domain.
This domain is broken down into subdomains—perhaps, for a technology-centric company, an Engineering department, a Sales department, and a Customer Support department. Each subdomain has its own requirements and duties and may itself be subdivided. This division process repeats until the subdomain models are granular and actionable and can be translated into small and independent services by the implementing teams. Bounded contexts are established around these subdomains and form the basis for the creation of microservices.

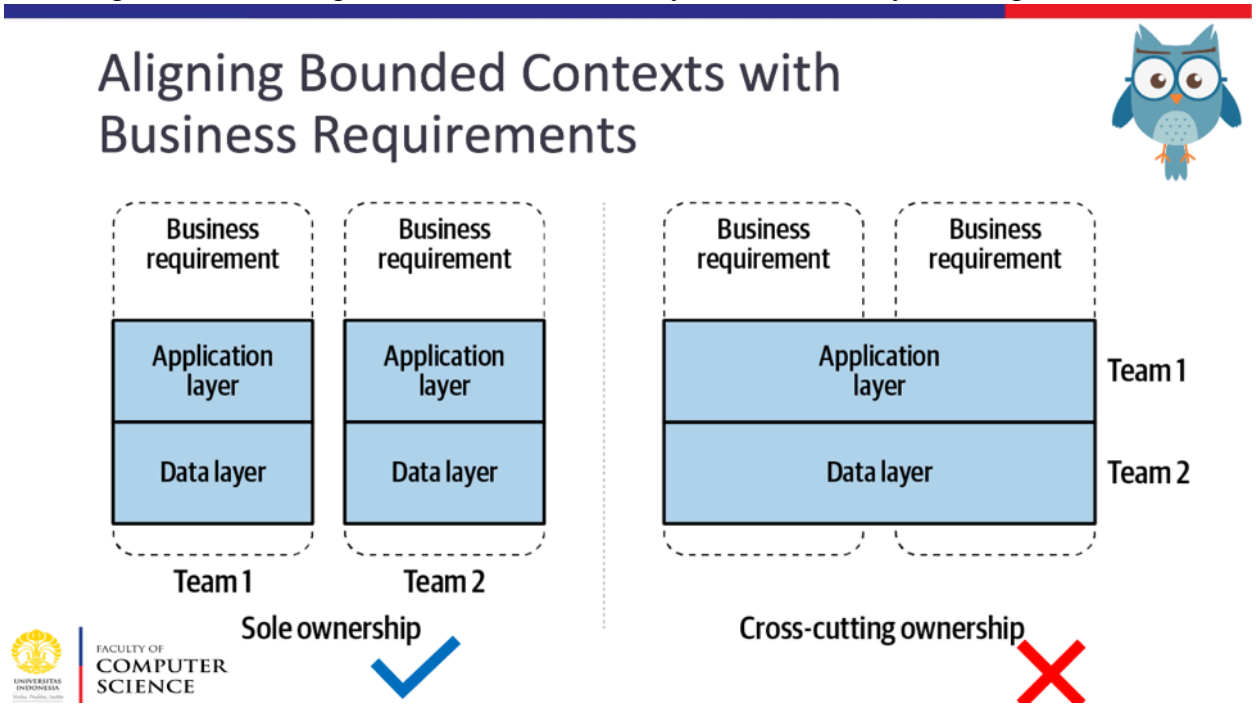
Aligning Bounded Contexts with Business Requirements

- **Bounded Context should be built around business requirement and not technological requirements.**
- Aligning bounded contexts on business requirements allows teams to make changes to microservice implementations in a loosely coupled and highly cohesive way.

FACULTY OF COMPUTER SCIENCE

It is common for the business requirements of a product to change during its lifetime, perhaps due to organizational changes or new feature requests. In contrast, it's rare for a company to need to change the underlying implementation of any given product without accompanying business requirement changes. This is why bounded contexts should be built around business requirements and not technological requirements.
Aligning bounded contexts on business requirements allows teams to make changes to microservice implementations in a loosely coupled and highly cohesive way. It provides a team with the autonomy to design and implement a solution for the specific business needs, which greatly reduces inter team dependencies and enables each team to focus strictly on its own requirements.
Conversely, aligning microservices on technical requirements is problematic. This pattern is often seen in improperly designed synchronous point-to-point microservices and in traditional monolith-style computing systems where teams own specific technical layers of the application. The main issue with technological alignment is that it distributes the responsibility of fulfilling the business function across multiple bounded contexts, which may involve multiple teams with

differing schedules and duties. Because no team is solely responsible for implementing a solution, each service becomes coupled to another across both team and API boundaries, making changes difficult and expensive. A seemingly innocent change, a bug, or a failed service can have serious ripple effects to the business-serving capabilities of all services that use the technical system. Technical alignment is seldomly used in event-driven microservice (EDM) architectures and should be avoided completely whenever possible. Eliminating cross-cutting technological and team dependencies will reduce a system's sensitivity to change.



Sole ownership on the left and cross-cutting ownership on the right. With sole ownership, the team is fully organized around the two independent business requirements (bounded contexts) and has complete control over its application code and the database layer. On the right, the teams have been organized via technical requirements, where the application layer is managed separate from the data layer. This creates explicit dependencies between the teams, as well as implicit dependencies between the business requirements.

Modeling event-driven microservices architectures around business requirements is preferred, though there are trade-offs with this approach. Code may be replicated a number of times, and many services may use similar data access patterns. Product developers may try to reduce repetition by sharing data sources with other products or by coupling on boundaries. In these cases, the subsequent tight coupling may be far more costly in the long run than repeating logic and storing similar data. These trade-offs will be examined in greater detail throughout this book.

Tip

Keep loose coupling between bounded contexts, and focus on minimizing intercontext dependencies. This will allow bounded context implementations to change as necessary, without subsequently breaking many (or any) other systems.

Additionally, each team may be required to have full stack expertise, which can be complicated by the need for specialized skill sets and access permissions. The organization should operationalize the most common requirements such that these vertical teams can support themselves, while more specialized skill sets can be provided on a cross-team, as-needed basis.

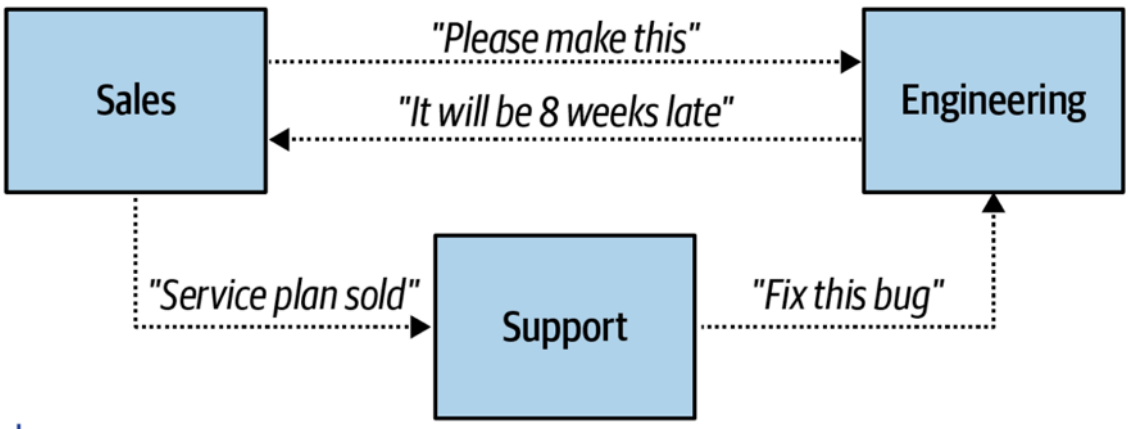
Communication Structures

- Business Communication Structures
- Implementation Communication Structures
- Data Communication Structures

FACULTY OF COMPUTER SCIENCE

An organization's teams, systems, and people all must communicate with one another to fulfil their goals. These communications form an interconnected topology of dependencies called a *communication structure*. There are three main communication structures, and each affects the way businesses operate.

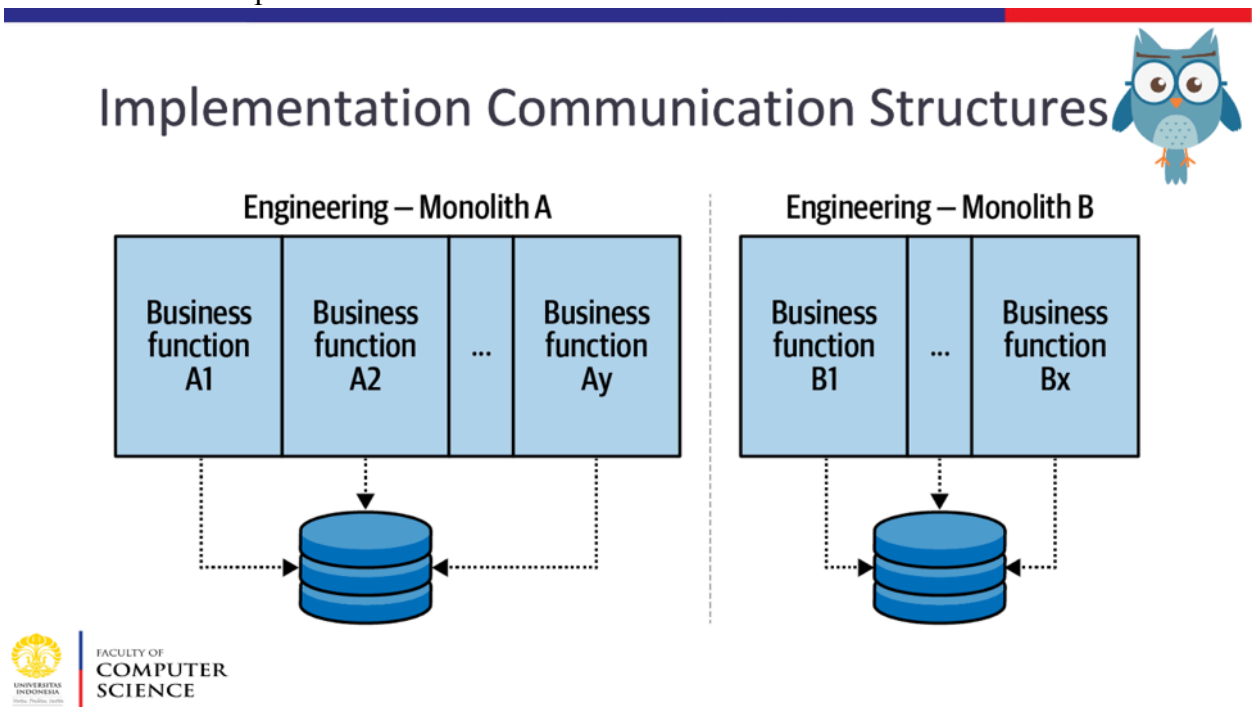
Business Communication Structure



FACULTY OF COMPUTER SCIENCE

The business communication structure dictates communication between teams and departments, each driven by the major requirements and responsibilities assigned to it. For example, Engineering produces the software products, Sales sells to customers, and Support ensures that

customers and clients are satisfied. The organization of teams and the provisioning of their goals, from the major business units down to the work of the individual contributor, fall under this structure. Business requirements, their assignment to teams, and team compositions all change over time, which can greatly impact the relationship between the business communication structure and the implementation communication structure.

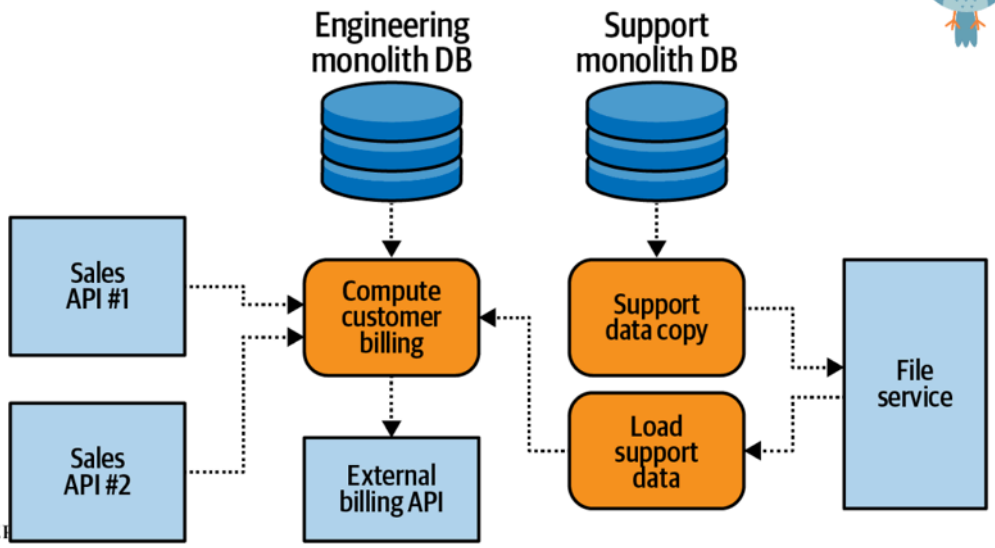


FACULTY OF COMPUTER SCIENCE

The *implementation communication structure* is the data and logic pertaining to the subdomain model as dictated by the organization. It formalizes business processes, data structures, and system design so that business operations can be performed quickly and efficiently. This results in a tradeoff in flexibility for the business communication structure, as redefining the business requirements that must be satisfied by the implementation requires a rewrite of the logic. These rewrites are most often iterative modifications to the subdomain model and associated code, which over time reflect the evolution of the implementation to fulfill the new business requirements.

The quintessential example of an implementation communication structure for software engineering is the monolithic database application. The business logic of the application communicates internally via either function calls or shared state. This monolithic application, in turn, is used to satisfy the business requirements dictated by the business communication structure.

Data Communication Structures



The data communication structure is the process through which data is communicated across the business and particularly between implementations. Although a data communication structure comprising email, instant messaging, and meetings is often used for communicating business changes, it has largely been neglected for software implementations. Its role has usually been fulfilled ad hoc, from system to system, with the implementation communication structure often playing double duty by including data communication functions in addition to its own requirements. This has caused many problems in how companies grow and change over time, the impact of which is evaluated in the next section.

Conway's Law and Communication Structures

> Organizations which design systems...are constrained to produce designs which are copies of the communication structures of these organizations.
>
> -- Melvin Conway—*How Do Committees Invent?* (April 1968)

This quote, known as *Conway's law*, implies that a team will build products according to the communication structures of its organization.

Business communication structures organize people into teams, and these teams typically produce products that are delimited by their team boundaries. Implementation communication structures provide access to the subdomain data models for a given product, but also restrict access to other products due to the weak data communication capabilities.

Because domain concepts span the business, domain data is often needed by other bounded contexts within an organization. Implementation communication structures are generally poor at providing this communication mechanism, though they excel at supplying the needs of their own bounded context. They influence the design of products in two ways. First, due to the inefficiencies of communicating the necessary domain data across the organization, they discourage the creation of new, logically separate products. Second, they provide easy access to existing domain data, at the risk of continually expanding the domain to encompass the new business requirements. This particular pattern is embodied by monolithic designs.

Data communication structures play a pivotal role in how an organization designs and builds products, but for many organizations this structure has long been missing. As noted, implementation communication structures frequently play this role in addition to their own. Some organizations attempt to mitigate the inability to access domain data from other implementations, but these efforts have their own drawbacks. For example, shared databases are often used, though these frequently promote anti-patterns and often cannot scale sufficiently to accommodate all performance requirements. Databases may provide read-only replicas; however, this can expose their inner data models unnecessarily. Batch processes can dump data to a file store to be read by other processes, but this approach can create issues around data consistency and multiple sources of truth. Lastly, all of these solutions result in a strong coupling between implementations and further harden an architecture into direct point-to-point relationships.

Caution: If you find that it is too hard to access data in your organization or that your products are scope-creeping because all the data is located in a single implementation, you're likely experiencing the effects of poor data communication structures. This problem will be magnified as the organization grows, develops new products, and increasingly needs to access commonly used domain data.

Event-Driven Communication Structures

- Events Are the Basis of Communication
- Event Streams Provide the Single Source of Truth
- Consumers Perform Their Own Modeling and Querying
- Data Communication Is Improved Across the Organization
- Accessible Data Supports Business Communication Changes

FACULTY OF COMPUTER SCIENCE

The event-driven approach offers an alternative to the traditional behavior of implementation and data communication structures. Event-based communications are not a drop-in replacement for request-response communications, but rather a completely different way of communicating between services. An event-streaming data communication structure decouples the production and ownership of data from the access to it. Services no longer couple directly through a request-response API, but instead through event data defined within event streams. Producers' responsibilities are limited to producing well-defined data into their respective event streams.

Events Are the Basis of Communication

All shareable data is published to a set of event streams, forming a continuous, canonical narrative detailing everything that has happened in the organization. This becomes the channel by which systems communicate with one another. Nearly anything can be communicated as an event, from simple occurrences to complex, stateful records. Events *are* the data; they are not merely signals indicating data is ready elsewhere or just a means of direct data transfer from one implementation to another. Rather, they act both as data storage and as a means of asynchronous communication between services.

Event Streams Provide the Single Source of Truth

Each event in a stream is a statement of fact, and together these statements form the single source of truth—the basis of communication for all systems within the organization. A communication structure is only as good as the veracity of its information, so it's critical that the organization adopts the event stream narrative as a single source of truth. If some teams choose instead to put conflicting data in other locations, the event stream's function as the organization's data communications backbone is significantly diminished.

Consumers Perform Their Own Modeling and Querying

The event-based data communication structure differs from an overextended implementation communication structure in that it is incapable of providing any querying or data lookup functionality. All business and application logic must be encapsulated within the producer and consumer of the events.

Data access and modeling requirements are completely shifted to the consumer, with consumers each obtaining their own copy of events from the source event streams. Any querying complexity is also shifted from the implementation communication structure of the data owner to that of the consumer. The consumer remains fully responsible for any mixing of data from multiple event streams, special query functionality, or other business-specific implementation logic. Both producers and consumers are otherwise relieved of their duty to provide querying mechanisms, data transfer mechanisms, APIs (application programming interfaces), and cross-team services for the means of communicating data. They are now limited in their responsibility to only solving the needs of their immediate bounded context.

Data Communication Is Improved Across the Organization

The usage of a data communications structure is an inversion, with all shareable data being exposed outside of the implementation communication structure. Not all data must be shared, and thus not all of it needs to be published to the set of event streams. However, any data that is of interest to any other team or service must be published to the common set of event streams, such that the production and ownership of data becomes fully decoupled. This provides the formalized data communication structure that has long been missing from system architectures and better adheres to the bounded context principles of loose coupling and high cohesiveness. Applications can now access data that would otherwise have been laborious to obtain via point-to-point connections. New services can simply acquire any needed data from the canonical event streams, create their own models and state, and perform any necessary business functions without depending on direct point-to-point connections or APIs with any other service. This unlocks the potential for an organization to more effectively use its vast amounts of data in any product, and even mix data from multiple products in unique and powerful ways.

Accessible Data Supports Business Communication Changes

Event streams contain core domain events that are central to the operation of the business. Though teams may restructure and projects may come and go, the important core domain data remains readily available to any new product that requires it, *independent of any specific implementation communication structure*. This gives the business unparalleled flexibility, as access to core domain events no longer relies upon any particular implementation.

Event-Driven Microservices

- Granularity
- Scalability
- Technological flexibility
- Business requirement flexibility
- Loosely coupling
- Continuous delivery support
- High testability

FACULTY OF COMPUTER SCIENCE

Event-driven microservices enable the business logic transformations and operations necessary to meet the requirements of the bounded context. These applications are tasked with fulfilling these requirements and emitting any of their own necessary events to other downstream consumers. Here are a few of the primary benefits of using event-driven microservices:
Granularity: Services map neatly to bounded contexts and can be easily rewritten when business requirements change.
Scalability: Individual services can be scaled up and down as needed.
Technological flexibility: Services use the most appropriate languages and technologies. This also allows for easy prototyping using pioneering technology.
Business requirement flexibility: Ownership of granular microservices is easy to reorganize. There are fewer cross-team dependencies compared to large services, and the organization can react more quickly to changes in business requirements that would otherwise be impeded by barriers to data access.
Loosely coupling: Event-driven microservices are coupled on domain data and not on a specific implementation API. Data schemas can be used to greatly improve how data changes are managed, as will be discussed in Chapter 3.
Continuous delivery support: It's easy to ship a small, modular microservice, and roll it back if needed.
High testability: Microservices tend to have fewer dependencies than large monoliths, making it easier to mock out the required testing endpoints and ensure proper code coverage.

Summary: Software Architecture

- A blueprint and a road map of software development.
- There are several styles of software architectures that has been defined. But there is no panacea. Everything is a trade-off.
- It cover many aspect, not only the evolvement of technology but also business.
- It is not a basic knowledge but rather an advanced topic of evolving and changing concept
- A software architect requires basic and depth technical skill. A software architect should code!
- A software architect should broaden its technical breadth.
- Use tools wherever possible to ease to process.

FACULTY OF COMPUTER SCIENCE

Tutorial A: Event-Driven Architecture

This tutorial is modified and based on https://medium.com/geekculture/event-driven-programming-with-rust-and-rabbitmq-using-crosstown-bus-c39c50ce6c98

However, some of the notes in the blog is not properly synchronized so please just follow this tutorial and contact the helpdesk channel of Advanced Programming in discord if you have difficulties.

The final working source code for references and checking can be found in this URL: https://github.com/paolopolosso/crosstown_bus_client_poc

Objectives

By doing this tutorial you should learn:

- How event driven architecture works
- How the message broker such as RabbitMQ or Kafka works
- How the event driven architecture could help a slow application become more responsive by distributing it and avoiding possible crash.

Preparation

1. Your favourite code editor.
2. Create two empty repositories; one for subscriber and another one for publisher

3. Install rust and cargo, follow the previous instruction if you haven't done that (or you can jump to: https://doc.rust-lang.org/book/ch01-00-getting-started.html)
4. Docker to run RabbitMQ. We will briefly show you how, after the tutorial about the code. If you already know how to install or use RabbitMQ separately, you may not need Docker. We suggest using docker just to simplify this tutorial.

Starting tutorial step by step:

In event-driven architecture, we create a system that consists of several independent program. Those programs communicate by sending data. We call the data as ***event***. The data or event is not directly sent to another, but it is sent to a ***message broker***. We use **RabbitMQ** as the message broker. There is a program that behave as the one that create event and publish it to the message broker. We call it ***publisher***. There is also a program (or more) that behave as the processor of the event. It consumes the event and processes it. We call it ***subscriber***.

Preparing the subscriber

0. Prepare an empty repository in gitlab or github or other repository you prefer.
1. First create a directory tutorial8, and under that directory you can type:

```
cargo new --bin subscriber
```

2. It will create another sub-directory of the name subscriber, you need to edit the main.rs file inside the src directory as below:

```
use borsh::{BorshDeserialize, BorshSerialize};
use crosstown_bus::{CrosstownBus,
MessageHandler, HandleError};
use std::{thread, time};


#[derive(Debug, Clone, BorshDeserialize,
BorshSerialize)]
pub struct UserCreatedEventMessage {
  pub user_id: String,
  pub user_name: String
}


pub struct UserCreatedHandler;
```

```
impl MessageHandler<UserCreatedEventMessage> for UserCreatedHandler {

  fn handle(&self, message: Box<UserCreatedEventMessage>
  ) -> Result<(), HandleError> {
    let ten_millis = time::Duration::from_millis(1000);
    let now = time::Instant::now();

    // thread::sleep(ten_millis);

    println!("In Ade's Computer [129500004y]. Message received: {:?}", message);
    ok(())
  }
}

fn main() {
  let listener = CrosstownBus::new_queue_listener("amqp://guest:guest@localhost:5672".to_owned()).unwrap();
  _ = listener.listen("user_created".to_owned(), UserCreatedHandler{}, crosstown_bus::QueueProperties { auto_delete: false, durable: false, use_dead_letter: true });

  loop {
  }
```

```
}
```

Notes: You need to change the sub-string “In Ade’s Computer.”, which other string that identify yourself, at least put your NPM in that string so that the assistant can check it.

3. in the Cargo.toml file, you need to edit it as follows:

```
[package]
name = "subscriber"
version = "0.1.0"
edition = "2021"

# See more keys and their definitions at https://doc.rust-lang.org/cargo/reference/manifest.html

[dependencies]
crosstown_bus = "0.5.3"
futures = "0.3"
borsh = "0.9.3"
borsh-derive = "0.9.1"
tokio = {version="1.21.2", features=["full"]}
```

Pay attention that in this tutorial, we are still using the old version of the library crosstown_bus. You may want to update it to latest version, but consequently you need to make some adjustment to the code. If you wish, you can do it later.
4. Try to build it (compile it), you should not run-it yet. You can try build it by:

```
cargo build
```

5. If it is compiled properly no error, than you can continue. If you found some error messages, check again for mistyped, and if it is failed to compile, don’t hesitate to ask the assistants by asking via the held-desk channel in discord.
6. Then try to commit and push it to your prepared repository. Remember that whenever you do ‘cargo new’, they create a git directory but locally. So you need to update the remote variable which previously still empty. Check previous course tutorial for this, or you could ask assistant if you forgot.

Commit and push with message “Initial subscriber code.”

7. Try to answer the following questions, and write the answer in the and new file readme.md in you repository.

a. what is ***amqp***?
b. what it means? guest:guest@localhost:5672 , what is the first ***guest***, and what is the second ***guest***, and what is ***localhost:5672*** is for?

Commit and push your answer with message "Understanding subscriber and message broker."

We are done for the moment with subscriber, next let's prepare the publisher.

Preparing the publisher

0. Prepare an empty repository in gitlab or github or other repository you prefer.
1. First create a directory tutorial8, and under that directory you can type:

```
cargo new --bin publisher
```

2. It will create another sub-directory of the name publisher, you need to edit the main.rs file inside the src directory as below:

```
use borsh::{BorshDeserialize, BorshSerialize};
use crosstown_bus::{CrosstownBus,
MessageHandler, HandleError};


#[derive(Debug, Clone, BorshDeserialize,
BorshSerialize)]
pub struct UserCreatedEventMessage {
  pub user_id: String,
  pub user_name: String
}


pub struct UserCreatedHandler;


impl
MessageHandler<UserCreatedEventMessage> for
UserCreatedHandler {
```

```
  fn handle(&self, message:
Box<UserCreatedEventMessage>) -> Result<(),
HandleError> {
    println!("Message received on handler 1: {:?}",
message);
    ok(())
  }
}

fn main() {
  let mut p =
CrosstownBus::new_queue_publisher("amqp://gue
st:guest@localhost:5672".to_owned()).unwrap();
  _ = p.publish_event("user_created".to_owned(),
UserCreatedEventMessage { user_id:
"1".to_owned(), user_name:
"1295oooo4y-Amir".to_owned() });
  _ = p.publish_event("user_created".to_owned(),
UserCreatedEventMessage { user_id:
"2".to_owned(), user_name:
"1295oooo4y-Budi".to_owned() });
  _ = p.publish_event("user_created".to_owned(),
UserCreatedEventMessage { user_id:
"3".to_owned(), user_name:
"1295oooo4y-Cica".to_owned() });
  _ = p.publish_event("user_created".to_owned(),
UserCreatedEventMessage { user_id:
"4".to_owned(), user_name:
"1295oooo4y-Dira".to_owned() });
  _ = p.publish_event("user_created".to_owned(),
UserCreatedEventMessage { user_id:
"5".to_owned(), user_name:
"1295oooo4y-Emir".to_owned() });
```

```
}
```

Notes: You need to change the sub-string “129500004y”, with your NPM so that the assistant can check it.

3. In directory publisher, you may find Cargo.toml, you need to edit it. Here is the expected contains:

```
[package]
name = "publisher"
version = "0.1.0"
edition = "2021"

# See more keys and their definitions at https://doc.rust-lang.org/cargo/reference/manifest.html

[dependencies]
crosstown_bus = "0.5.0"
futures = "0.3"
borsh = "0.9.3"
borsh-derive = "0.9.1"
tokio = {version="1.21.2", features=["full"]}
```

4. Try to build it (compile it), you should not run-it yet. You can try build it (in publisher directory) by:

```
cargo build
```

5. If it is compiled properly no error, than you can continue. If you found some error messages, check again for mistyped, and if it is failed to compile, don’t hesitate to ask the assistants by asking via the held-desk channel in discord.
6. Then try to commit and push it to your prepared repository (different repository than subscriber). Remember that whenever you do ‘cargo new’, they create a git directory but locally. So you need to update the remote variable which previously still empty. Check previous course tutorial for this, or you could ask assistant if you forgot.

Commit and push with message “Initial publisher code.”

7. Try to answer the following questions, and write the answer in the and new file readme.md in you repository.

a. How many data your publlsher program will send to the message broker in one run?
b. The url of: "amqp://guest:guest@localhost:5672" is the same as in the subscriber program, what does it mean?

Commit and push your answer with message "Understanding publisher and message broker."

We are done for the moment with publisher, next let's prepare the message broker (RabbitMQ).

Preparing Message Broker (RabbitMQ)

There are several tools that can serve as Message Broker, such as Kafka and RabbitMQ. To simplify we use docker. Students who already familiar on how to install it, can just install it without following this guide. Just make sure that the username, password, port and others configuration has been setup and adjust properly with the code as well.

Let's install message broker using RabbitMQ in docker.

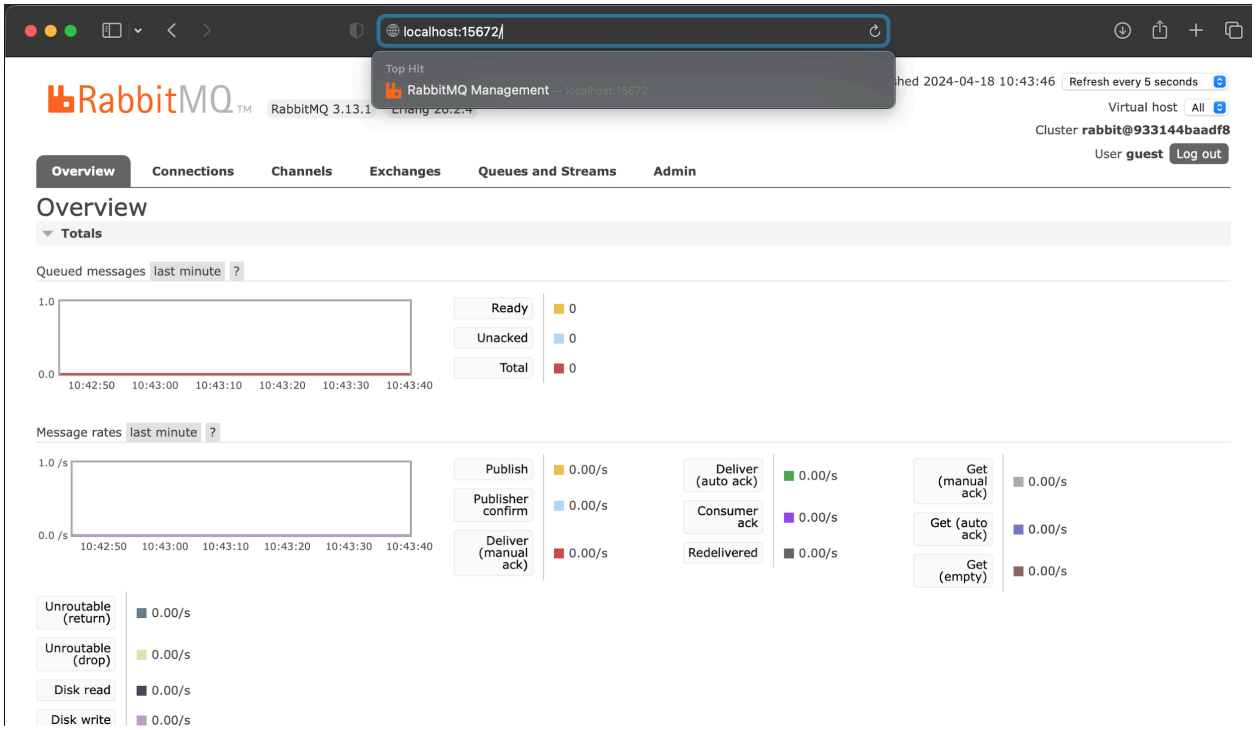
0. Install docker and run you docker.
1. if you have installed and make sure you docker has been installed properly, try to execute this in the console: (notes: it is a one line command)

```
docker run -it --rm --name rabbitmq -p 5672:5672 -p 15672:15672 rabbitmq:3.13-management
```

This command will find rabbitmq image, if it is not found in your computer, it will download it. So, make sure you have a fine network connection and spare harddisk.

The message broker RabbitMQ will listen on port 5672, and the interface can be reach on your localhost port 15672. The username and the password for this default RabbitMQ is ***guest*** and ***guest***. (Username: `guest`, password: `guest`).

If you open your browser, and enter the mentioned username and password, you will have the screen:



On your publisher Readme.md, edit it, and put your screen of your running RabbitMQ.

Commit and push your changes with message “Running RabbitMQ as message broker.”

Make it works

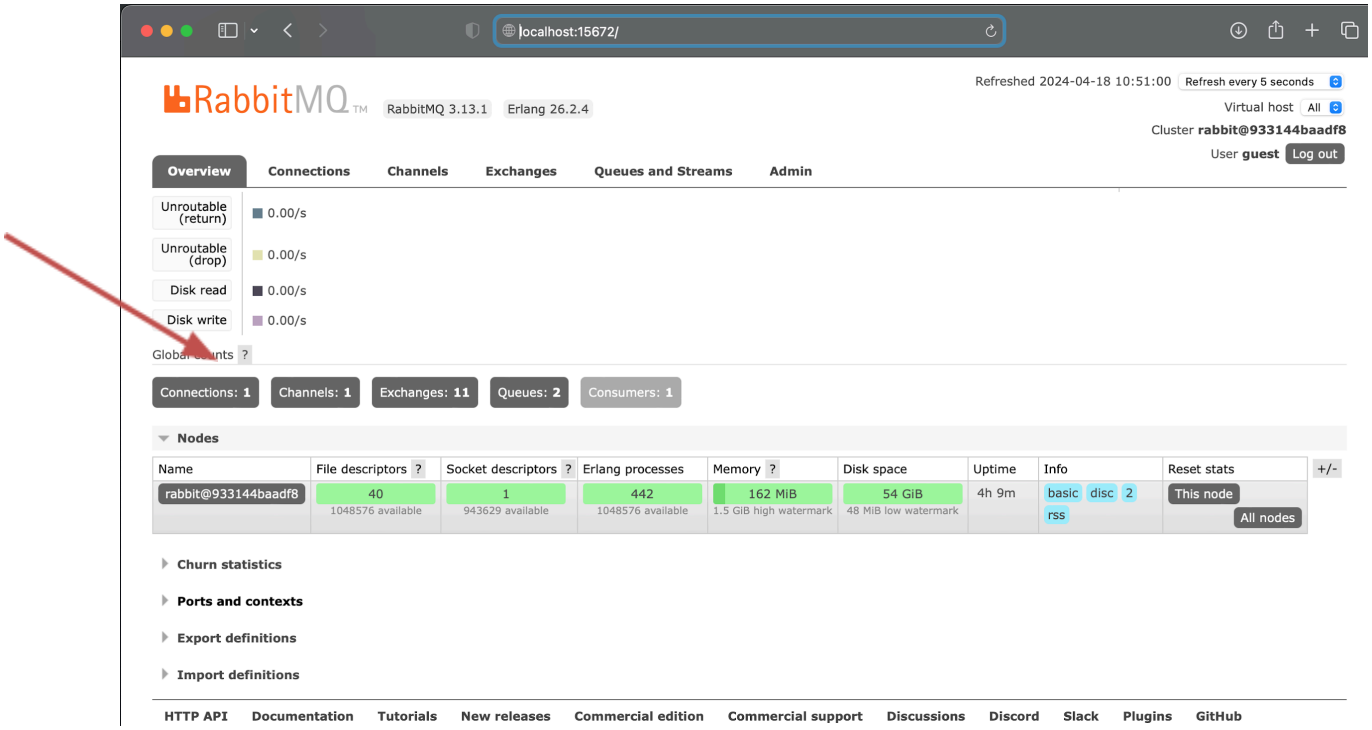
1. Open a console, go to your subscriber directory, and type

```
cargo run
```

2. Open another console, go to your publisher directory, and type

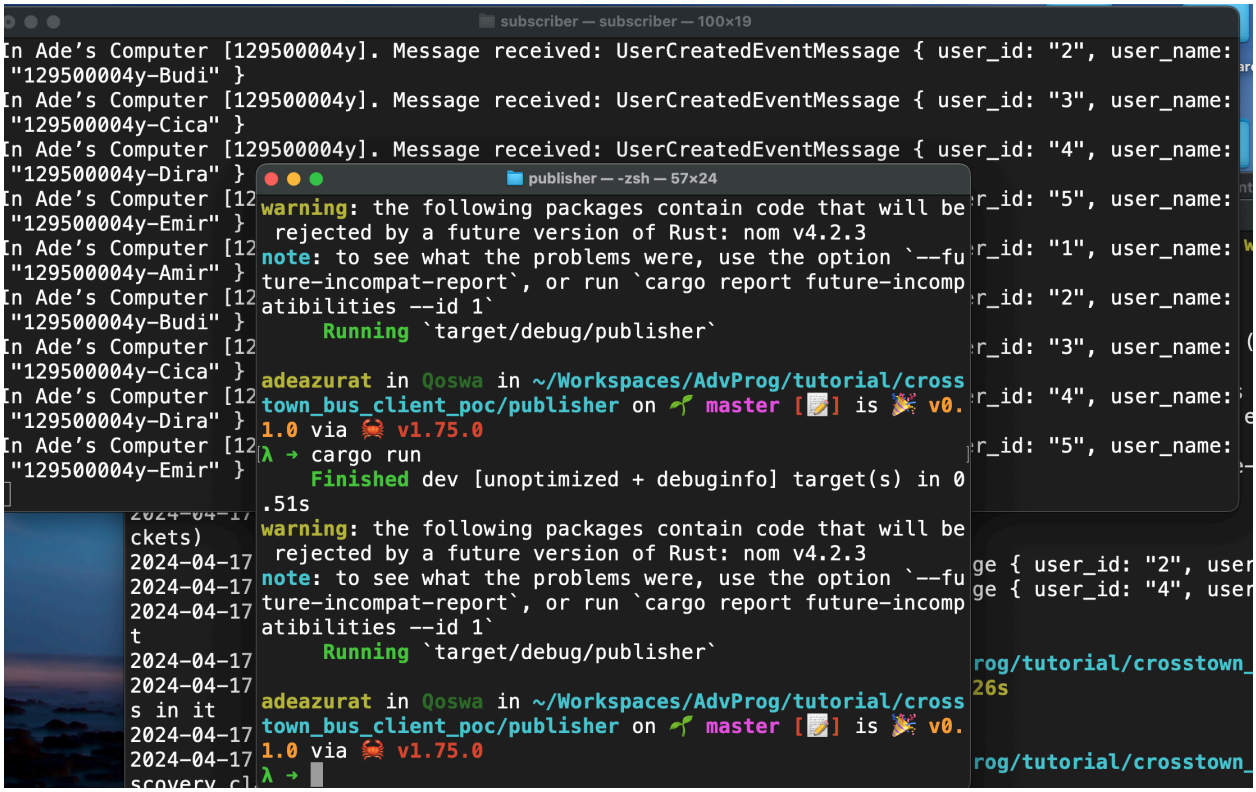
```
cargo run
```

Go to your RabbitMQ browser, scroll down a little bit, now you should at least have one connection as shown:



It means you have one subscriber making connection to the message broker.

Now try to run the publisher again, ‘cargo run’ in the directory publisher. Pay attention on your subscriber console. You may find some thing like this:

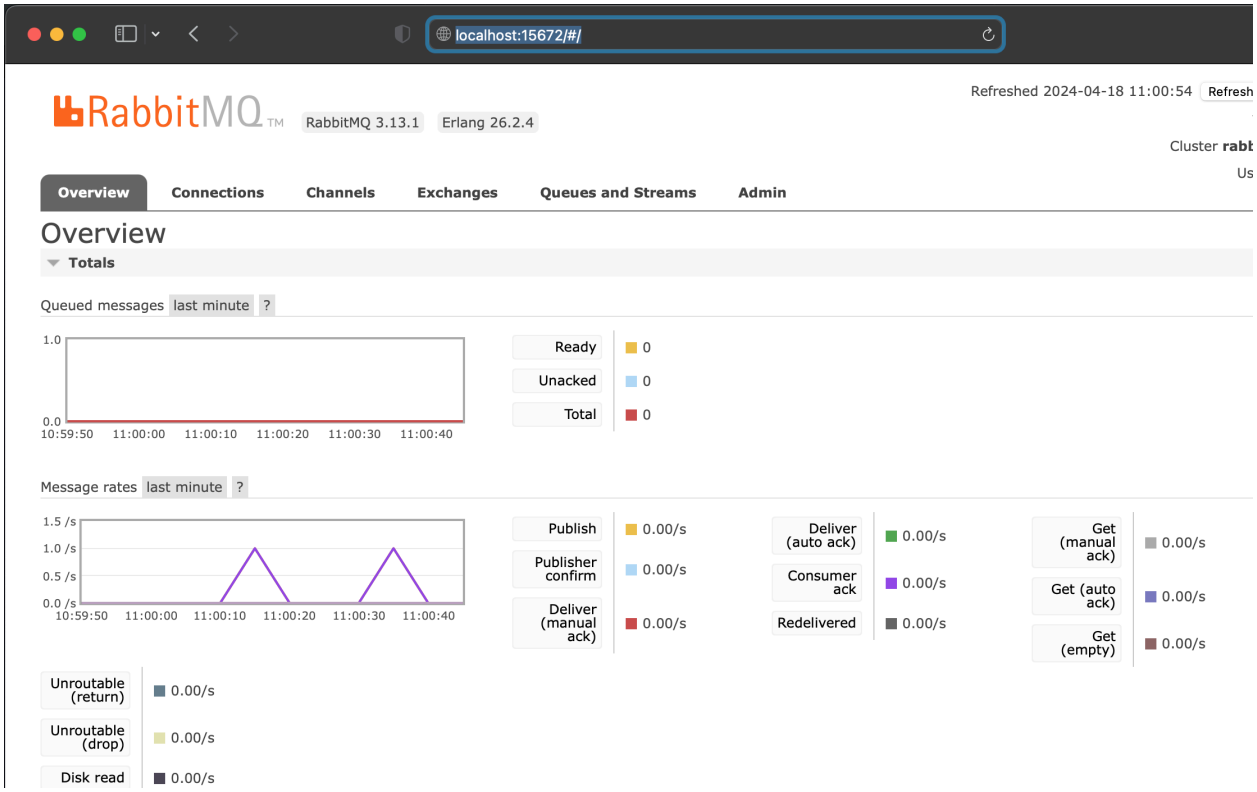


What happen is: when I run 'cargo run' on publisher (I run the publisher), the publisher sent 5 event to the message broker. Those event later consumed and processed by the subscriber.

Try to capture your screen showing these console, put it on your publisher repository Readme.md, put some sentences describing what was happening.

Commit and push your changes with message "Sending and processing event."

Try to run the publisher again, but know, pay attention on your RabbitMQ browser. (you may need to scroll up to see the chart).



You may see some spikes on the second charts. You can try repeatedly run the publisher.

Edit your publisher readme.md again, capture your browser, and explain how the spike got to do with running the publisher. Put it on readme.md.

Commit and push your changes with message “Monitoring chart based on publisher.”

Simulating slow subscriber

In reality, sometime program perform slow, while the demand from user is high. Just remember when you fill in IRS every semester. Processing an IRS requires some computing time, but during the SIAK War, so many students try to fill in as soon as possible. The demand is so high, but the computing power is slow. If this happen, it may crash the system just like what happen with SIAK sometimes.

Let’s simulate. We will make our subscriber slow, it enforce additional delay 1 second for every process.

Now, open your subscriber directory, open the main.rs in the subscriber/src, uncomment this statement: (probably line 22)

```
thread::sleep(ten_millis);
```

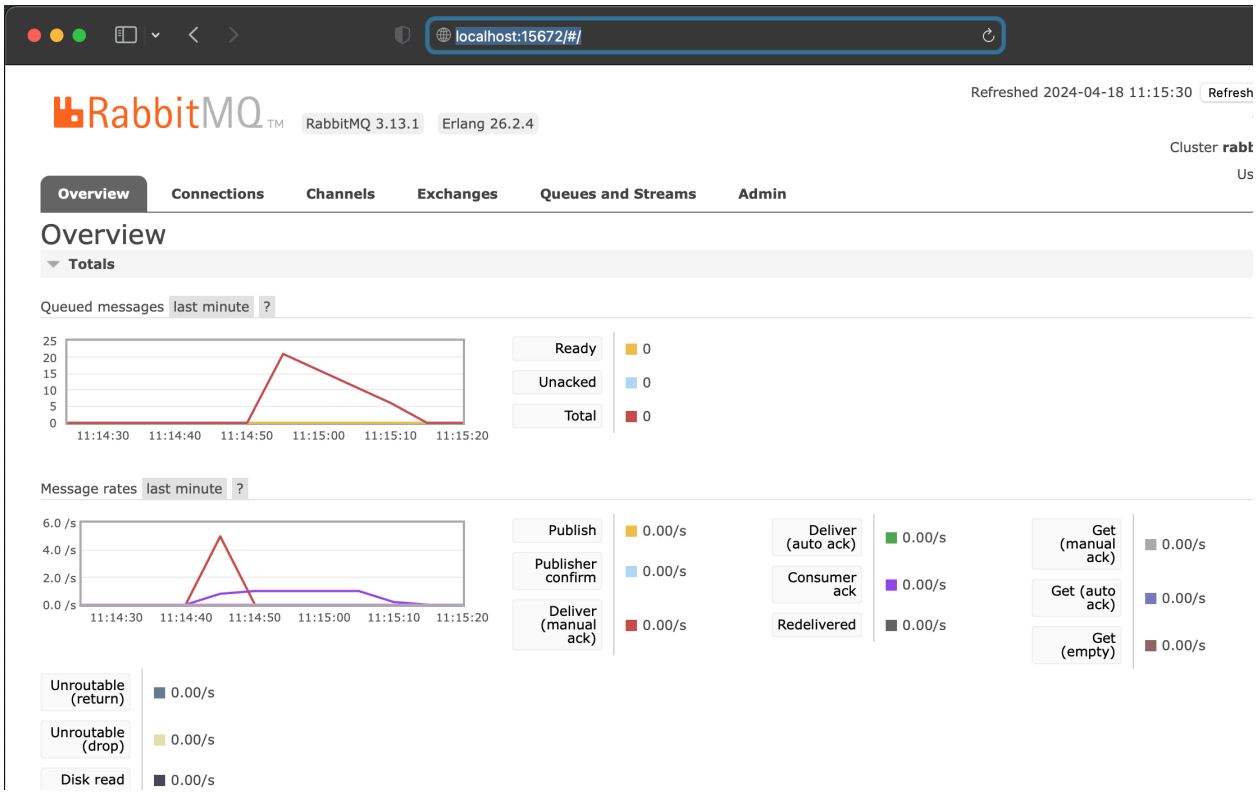
This additional code is enough to simulate a slow subscriber. You may want to close / kill your previous subscriber by closing the console or typing ctrl-C on that console.
Don't forget to save in the editor and then compile your subscriber by running the following command in subscriber directory:

```
cargo build
```

And then run it again

```
cargo run
```

Now move to directory publisher, and run it several times quickly, by typing 'cargo run' in publisher directory several time quickly, and move to your browser RabbitMQ pay attention to the first chart:



You will see some thing like this. It means the producer can just keep sending request, and those request (as event) is put on a queue message. Slowly the consumer will process it one by one.

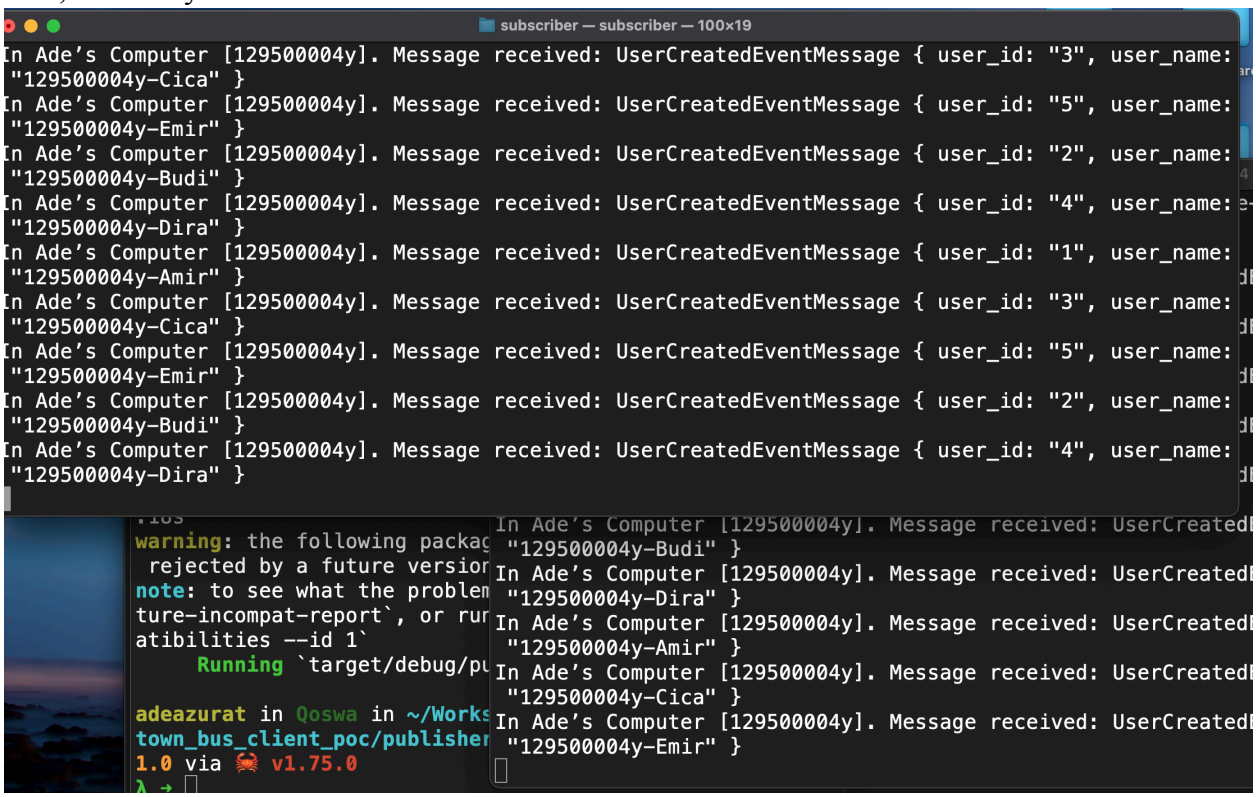
In your subscriber directory, edit your Readme.md, add the screen capture of yours, and answer why the total number of queue is as such (in my machine is 20, what about yours?)

Commit and push your changes with message "Simulation slow subscriber"

Ok, let see if we really have slow subscriber, what should we do. In event-driven architecture, it is possible that we spawn many subscribers connected to the same queue. So you can try to open some other consoles, in each console go to subscriber directory, and the run it by typing ‘cargo run’.

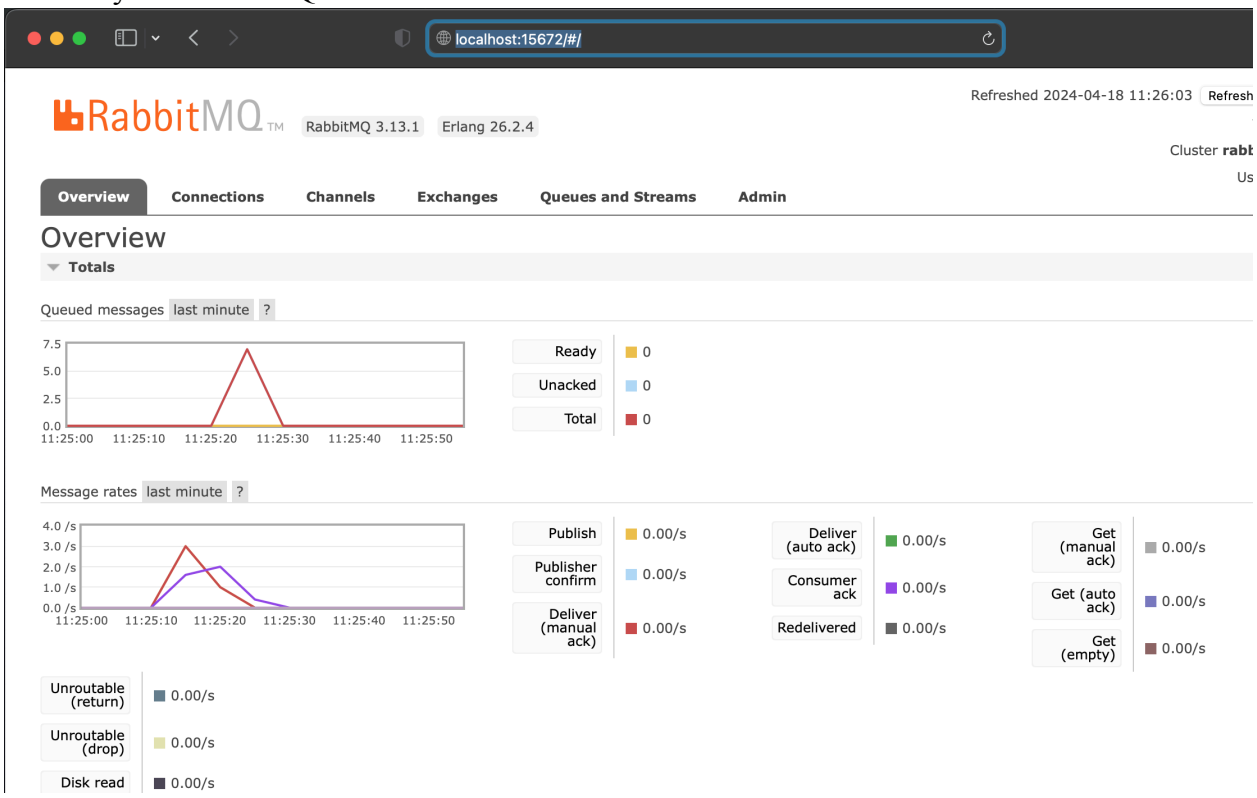
In the publisher directory you can repeat simulating so many request by typing ‘cargo run’ several times.

First, monitor your console:



You see in the console, the event processing is splited, one is processing "Budi" and "Dira", and the other is processing "Amir", "Cica" and "Emir" (see the last few messages of each console). So both subscribers are processing the message together. In my machine it only run two subscriber consoles, you need to try to open three consoles.

Monitor your RabbitMQ browser.



You see the spike of the message queue is reduced quicker than before.

Edit your subscriber readme.md, put your capture in the readme.md, and also add some explanation/reflection of why it is like that. Take a look at the code of publisher and subscriber, do you see something to improve?

Commit and push your changes with message “Reflection and Running at least three subscribers”

Hope you learned about event driven.

Bonus:
You run your experiment on cloud, not in your local machine. Then re-do the commit message, from section “Make it works” and “Simulating slow subscriber”. On the cloud, you may need to open the port for external connection in the firewall configuration.

Grading Scheme

- Students should follow the instructions in the given module.
- Do the tutorial in the module.
- Make sure to do all the commits as requested.
- Do not squash your commit during the merge process. It may destroy your commits history and the teaching assistants cannot evaluate it.
- Write down your personal reflection/answering notes as markdown in the readme.md on your master branch.
- Push your work to the repository.
- Give access to the teaching assistants.
- Submit the both repositories in the scele as requested.

Scale

All components will be scored on discrete scale 0,1,2,3,4. Grade 4 is the highest grade, equivalent to A. Score 0 is for not submitting. No partial score.

Generic Components

- 40% - Commits
- 40% - Reflection
- 20% - Correctness
- Bonus 10%:
    - Reflection (5%) and
    - Correctness (5%).

Generic Rubrics

	Score 4	Score 3	Score 2	Score 1
Correctness	Correct perfectly as requested.	Something is missing. Or only do the tutorial parts. Exercise is missing.	A lot of requirements are missing, but the program still runs.	Incorrect program
Reflection (for each reflection)	More than 5 sentences. The description is sound.	Less than or equal 5 sentences. The description is sound.	The description is not sound although still related.	The description is not sound. It is not related to the topics.
Commit (evaluated on all commits)	Commit correctly	Commit is not correct.	Commit correctly but late	Incorrect commit

8. Analysing Architectural Risk

Introduction

- Every architecture has risk associated with it, whether it be risk involving availability, scalability, or data integrity.
- Analyzing architecture risk is one of the key activities of architecture.
- We will discuss some of the key ***techniques*** and ***practices*** for:
  - qualifying risk,
  - creating risk assessments, and
  - identifying risk
- through an activity called ***risk storming***.

FACULTY OF COMPUTER SCIENCE

Every architecture has risk associated with it, whether it be risk involving availability, scalability, or data integrity. Analyzing architecture risk is one of the key activities of architecture. By continually analyzing risk, the architect can address deficiencies within the architecture and take corrective action to mitigate the risk. In this chapter we introduce some of the key techniques and practices for qualifying risk, creating risk assessments, and identifying risk through an activity called risk storming.

Risk Matrix

The first issue that arises when assessing architecture risk is determining whether the risk should be classified as low, medium, or high. Too much subjectiveness usually enters into this classification, creating confusion about which parts of the architecture are really high risk versus medium risk. Fortunately, there is a risk matrix architects can leverage to help reduce the level of subjectiveness and qualify the risk associated with a particular area of the architecture.

The architecture risk matrix (illustrated in Figure 20-1) uses two dimensions to qualify risk: the overall impact of the risk and the likelihood of that risk occurring. Each dimensions has a low (1), medium (2), and high (3) rating. These numbers are multiplied together within each grid of the matrix, providing an objective numerical number representing that risk. Numbers 1 and 2 are considered low risk (green), numbers 3 and 4 are considered medium risk (yellow), and numbers 6 through 9 are considered high risk (red).

Risk Matrix

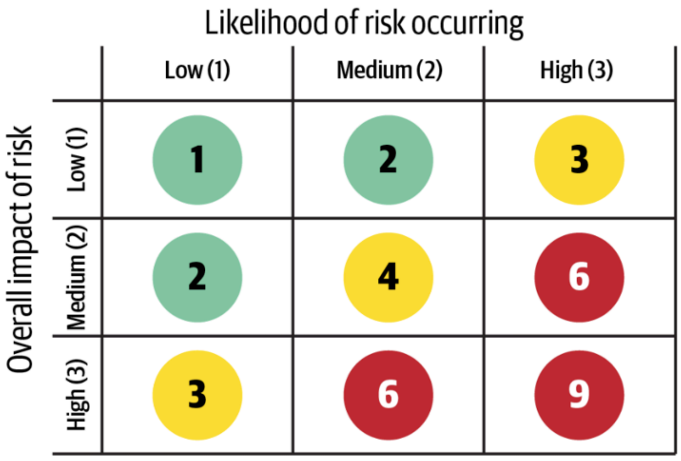


Figure 20-1. Matrix for determining architecture risk

To see how the risk matrix can be used, suppose there is a concern about availability with regard to a primary central database used in the application. First, consider the impact dimension—what is the overall impact if the database goes down or becomes unavailable? Here, an architect might deem that high risk, making that risk either a 3 (medium), 6 (high), or 9 (high). However, after applying the second dimension (likelihood of risk occurring), the architect realizes that the database is on highly available servers in a clustered configuration, so the likelihood is low that the database would become unavailable. Therefore, the intersection between the high impact and low likelihood gives an overall risk rating of 3 (medium risk).

```
Tip:

When leveraging the risk matrix to qualify the risk, consider
the impact dimension first and the likelihood dimension
second.
```

Risk Assessment

RISK CRITERIA	Customer registration	Catalog checkout	Order fulfillment	Order shipment	TOTAL RISK
Scalability	2	6	1	2	11
Availability	3	4	2	1	10
Performance	4	2	3	6	15
Security	6	3	1	1	11
Data integrity	9	6	1	1	17
TOTAL RISK	24	21	8	11	

Figure 20-2. Example of a standard risk assessment

FACULTY OF COMPUTER SCIENCE

The risk matrix described in the previous section can be used to build what is called a risk assessment. A risk assessment is a summarized report of the overall risk of an architecture with respect to some sort of contextual and meaningful assessment criteria.

Risk assessments can vary greatly, but in general they contain the risk (qualified from the risk matrix) of some assessment criteria based on services or domain areas of an application. This basic risk assessment report format is illustrated in Figure 20-2, where light gray (1-2) is low risk, medium gray (3-4) is medium risk, and dark gray (6-9) is high risk. Usually these are color-coded as green (low), yellow (medium), and red (high), but shading can be useful for black-and-white rendering and for color blindness.

The quantified risk from the risk matrix can be accumulated by the risk criteria and also by the service or domain area. For example, notice in Figure 20-2 that the accumulated risk for data integrity is the highest risk area at a total of 17, whereas the accumulated risk for Availability is only 10 (the least amount of risk). The relative risk of each domain area can also be determined by the example risk assessment. Here, customer registration carries the highest area of risk, whereas order fulfillment carries the lowest risk. These relative numbers can then be tracked to demonstrate either improvements or degradation of risk within a particular risk category or domain area.

Although the risk assessment example in Figure 20-2 contains all the risk analysis results, rarely is it presented as such. Filtering is essential for visually indicating a particular message within a given context. For example, suppose an architect is in a meeting for the purpose of presenting areas of the system that are high risk. Rather than presenting the risk assessment as illustrated in

Figure 20-2, filtering can be used to only show the high risk areas (shown in Figure 20-3), improving the overall signal-to-noise ratio and presenting a clear picture of the state of the system (good or bad).

Filtering High Risk

RISK CRITERIA	Customer registration	Catalog checkout	Order fulfillment	Order shipment	TOTAL RISK
Scalability		6			6
Availability					0
Performance				6	6
Security	6				6
Data integrity	9	6			15
TOTAL RISK	15	12	0	6	

Figure 20-3. Filtering the risk assessment to only high risk

FACULTY OF COMPUTER SCIENCE

Another issue with Figure 20-2 is that this assessment report only shows a snapshot in time; it does not show whether things are improving or getting worse. In other words, Figure 20-2 does not show the direction of risk. Rendering the direction of risk presents somewhat of an issue. If an up or down arrow were to be used to indicate direction, what would an up arrow mean? Are things getting better or worse? We've spent years asking people if an up arrow meant things were getting better or worse, and almost 50% of people asked said that the up arrow meant things were progressively getting worse, whereas almost 50% said an up arrow indicated things were getting better. The same is true for left and right arrows. For this reason, when using arrows to indicate direction, a key must be used. However, we've also fund this doesn't work either. Once the user scrolls beyond the key, confusion happens once again.

We usually use the universal direction symbol of a plus (+) and minus (-) sign next to the risk rating to indicate direction, as illustrated in Figure 20-4. Notice in Figure 20-4 that although performance for customer registration is medium (4), the direction is a minus sign (red), indicating that it is progressively getting worse and heading toward high risk. On the other hand, notice that scalability of catalog checkout is high (6) with a plus sign (green), showing that it is improving. Risk ratings without a plus or minus sign indicate that the risk is stable and neither getting better nor worse.

Risk Direction

RISK CRITERIA	Customer registration	Catalog checkout	Order fulfillment	Order shipment	TOTAL RISK
Scalability	2	6 +	1	2	11
Availability	3	4	2 −	1	10
Performance	4 −	2 +	3 −	6 +	15
Security	6 −	3	1	1	11
Data integrity	9 +	6 −	1 −	1	17
TOTAL RISK	24	21	8	11	

FACULTY OF COMPUTER SCIENCE

Occasionally, even the plus and minus signs can be confusing to some people. Another technique for indicating direction is to leverage an arrow along with the risk rating number it is trending toward. This technique, as illustrated in Figure 20-5, does not require a key because the direction is clear. Furthermore, the use of colors (red arrow for worse, green arrow for better) makes it even more clear where the risk is heading.

Risk Direction

RISK CRITERIA	Customer registration	Catalog checkout	Order fulfillment	Order shipment	TOTAL RISK
Scalability	2	6 ↑4	1	2	11
Availability	3	4	2 ↓3	1	10
Performance	4 ↓6	2 ↑1	3 ↓4	6 ↑4	15
Security	6 ↓9	3	1	1	11
Data integrity	9 ↑6	6 ↓9	1 ↓2	1	17
TOTAL RISK	24	21	8	11	

Figure 20-5. Showing direction of risk with arrows and numbers

FACULTY OF COMPUTER SCIENCE

Risk Storming

- No architect can single-handedly determine the overall risk of a system.
  - Might miss or overlook a risk area
  - Less knowledge in some part of the systems
- Risk storming is a collaborative exercise used to determine architectural risk within a specific dimension.

FACULTY OF COMPUTER SCIENCE

No architect can single-handedly determine the overall risk of a system. The reason for this is two-fold. First, a single architect might miss or overlook a risk area, and very few architects have full knowledge of every part of the system. This is where risk storming can help.
Risk storming is a collaborative exercise used to determine architectural risk within a specific dimension. Common dimensions (areas of risk) include unproven technology, performance, scalability, availability (including transitive dependencies), data loss, single points of failure, and security. While most risk storming efforts involve multiple architects, it is wise to include senior developers and tech leads as well. Not only will they provide an implementation perspective to the architectural risk, but involving developers helps them gain a better understanding of the architecture.

Risk Storming Effort

- Individual part: All participants, individually, assign risk to areas of the architecture using the risk matrix
- Collaborative part: All participants work together to gain consensus on risk areas, discuss risk and form solutions for mitigating risk.

The risk storming effort involves both an individual part and a collaborative part. In the individual part, all participants individually (without collaboration) assign risk to areas of the architecture using the risk matrix described in the previous section. This noncollaborative part of risk storming is essential so that participants don't influence or direct attention away from particular areas of the architecture. In the collaborative part of risk storming, all participants work together to gain consensus on risk areas, discuss risk, and form solutions for mitigating the risk.

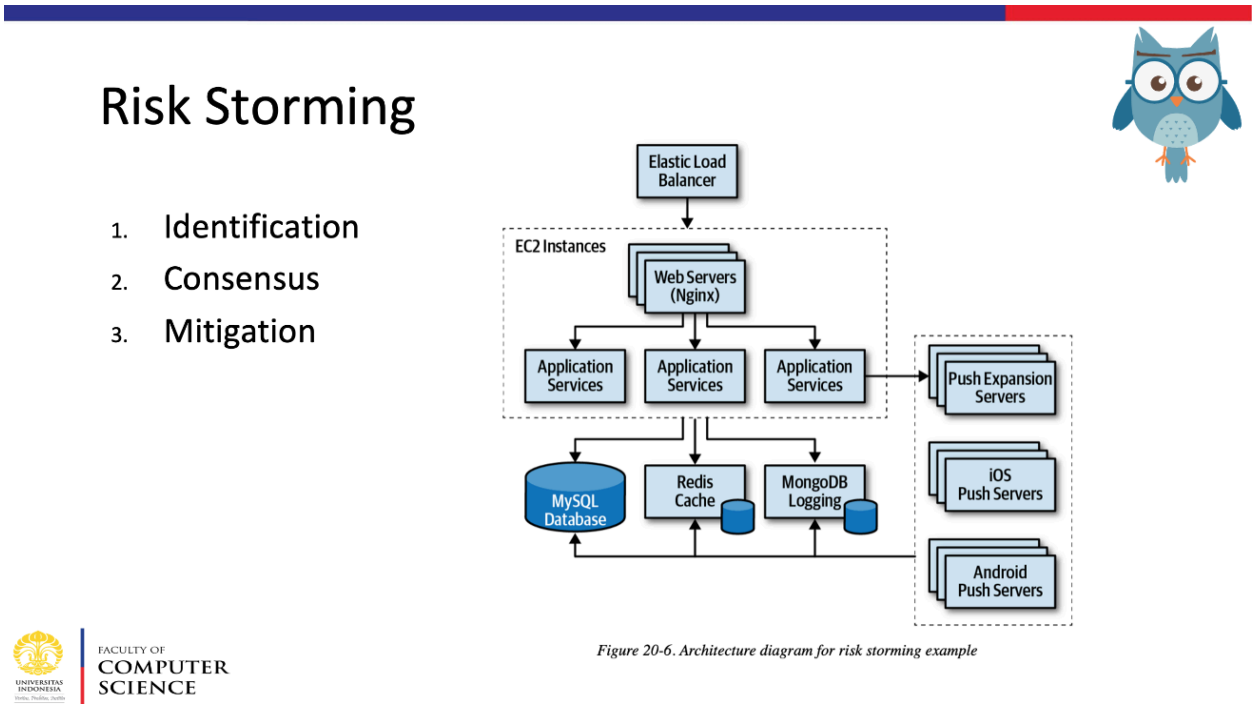


Figure 20-6. Architecture diagram for risk storming example

An architecture diagram is used for both parts of the risk storming effort. For holistic risk assessments, usually a comprehensive architecture diagram is used, whereas risk storming within specific areas of the application would use a contextual architecture diagram. It is the responsibility of the architect conducting the risk storming effort to make sure these diagrams are up to date and available to all participants.

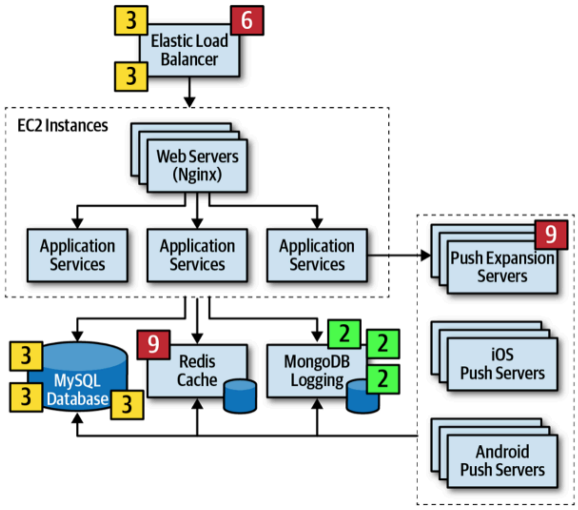
Figure 20-6 shows an example architecture we'll use to illustrate the risk storming process. In this architecture, an Elastic Load Balancer fronts each EC2 instance containing the web servers (Nginx) and application services. The application services make calls to a MySQL database, a Redis cache, and a MongoDB database for logging. They also make calls to the Push Expansion Servers. The expansion servers, in turn, all interface with the MySQL database, Redis cache, and MongoDB logging facility.

Risk storming is broken down into three primary activities:

1. Identification
2. Consensus
3. Mitigation

Identification is always an individual, noncollaborative activity, whereas consensus and mitigation are always collaborative and involve all participants working together in the same room (at least virtually).

Risk Storming: Identification



FACULTY OF COMPUTER SCIENCE

Figure 20-7. Initial identification of risk areas

The identification activity of risk storming involves each participant individually identifying areas of risk within the architecture. The following steps describe the identification part of the risk storming effort:

1. The architect conducting the risk storming sends out an invitation to all participants one to two days prior to the collaborative part of the effort. The invitation contains the architecture diagram (or the location of where to find it), the risk storming dimension (area of risk being analyzed for that particular risk storming effort), the date when the collaborative part of risk storming will take place, and the location.
2. Using the risk matrix described in the first section of this chapter, participants individually analyze the architecture and classify the risk as low (1-2), medium (3-4), or high (6-9).
3. Participants prepare small Post-it notes with corresponding colors (green, yellow, and red) and write down the corresponding risk number (found on the risk matrix).

Risk Storming: Consensus

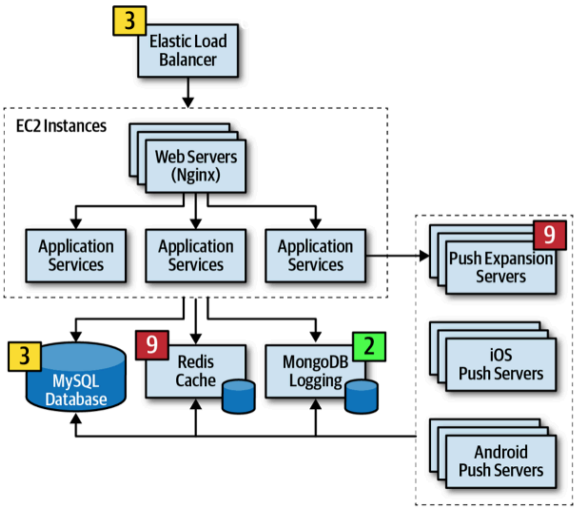


Figure 20-8. Consensus of risk areas

The consensus activity in the risk storming effort is highly collaborative with the goal of gaining consensus among all participants regarding the risk within the architecture. This activity is most effective when a large, printed version of the architecture diagram is available and posted on the wall. In lieu of a large printed version, an electronic version can be displayed on a large screen. Upon arrival at the risk storming session, participants begin placing their Post-it notes on the architecture diagram in the area where they individually found risk. If an electronic version is used, the architect conducting the risk storming session queries every participant and electronically places the risk on the diagram in the area of the architecture where the risk was identified (see Figure 20-7).

Once all of the Post-it notes are in place, the collaborative part of risk storming can begin. The goal of this activity of risk storming is to analyze the risk areas as a team and gain consensus in terms of the risk qualification. Notice several areas of risk were identified in the architecture, illustrated in Figure 20-7:

1. Two participants individually identified the Elastic Load Balancer as medium risk (3), whereas one participant identified it as high risk (6).
2. One participant individually identified the Push Expansion Servers as high risk (9)
3. Three participants individually identified the MySQL database as medium risk (3).
4. One participant individually identified the Redis cache as high risk (9).
5. Three participants identified MongoDB logging as low risk (2).
6. All other areas of the architecture were not deemed to carry any risk, hence there are no Post-it notes on any other areas of the architecture.

Items 3 and 5 in the prior list do not need further discussion in this activity since all participants agreed on the level and qualification of risk. However, notice there was a difference of opinion in

item 1 in the list, and items 2 and 4 only had a single participant identifying the risk. These items need to be discussed during this activity.

Item 1 in the list showed that two participants individually identified the Elastic Load Balancer as medium risk (3), whereas one participant identified it as high risk (6). In this case the other two participants ask the third participant why they identified the risk as high. Suppose the third participant says that they assigned the risk as high because if the Elastic Load Balancer goes down, the entire system cannot be accessed. While this is true and in fact does bring the overall impact rating to high, the other two participants convince the third participant that there is low risk of this happening. After much discussion, the third participant agrees, bringing that risk level down to a medium (3). However, the first and second participants might not have seen a particular aspect of risk in the Elastic Load Balancer that the third did, hence the need for collaboration within this activity of risk storming.

Case in point, consider item 2 in the prior list where one participant individually identified the Push Expansion Servers as high risk (9), whereas no other participant identified them as any risk at all. In this case, all other participants ask the participant who identified the risk why they rated it as high. That participant then says that they have had bad experiences with the Push Expansion Servers continually going down under high load, something this particular architecture has. This example shows the value of risk storming—without that participant's involvement, no one would have seen the high risk (until well into production of course!).

Item 4 in the list is an interesting case. One participant identified the Redis cache as high risk (9), whereas no other participant saw that cache as any risk in the architecture. The other participants ask what the rationale is for the high risk in that area, and the one participant responds with, "What is a Redis cache?" In this case, Redis was unknown to the participant, hence the high risk in that area.

Risk Storming: Mitigation

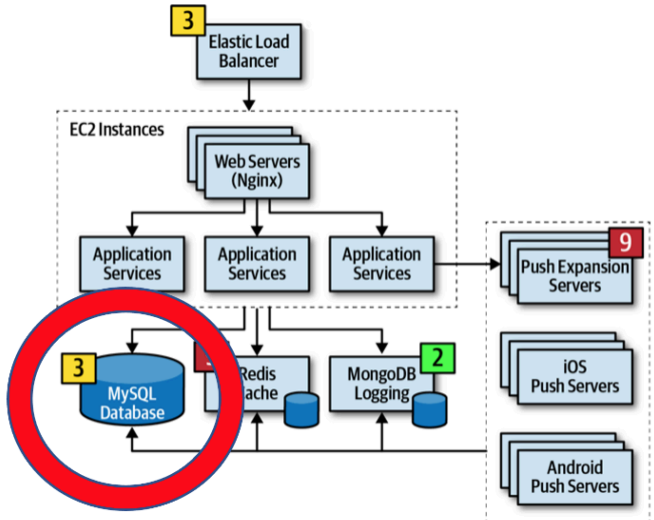
- Mitigating risk within an architecture usually involves changes or enhancements to certain areas of the architecture that otherwise might have been deemed perfect the way they were.
- This activity, which is also usually collaborative, seeks ways to reduce or eliminate the risk identified in the first activity.

FACULTY OF COMPUTER SCIENCE

Once all participants agree on the qualification of the risk areas of the architecture, the final and most important activity occurs—risk mitigation. Mitigating risk within an architecture usually involves changes or enhancements to certain areas of the architecture that otherwise might have been deemed perfect the way they were.

This activity, which is also usually collaborative, seeks ways to reduce or eliminate the risk identified in the first activity. There may be cases where the original architecture needs to be completely changed based on the identification of risk, whereas others might be a straightforward architecture refactoring, such as adding a queue for back pressure to reduce a throughput bottleneck issue.

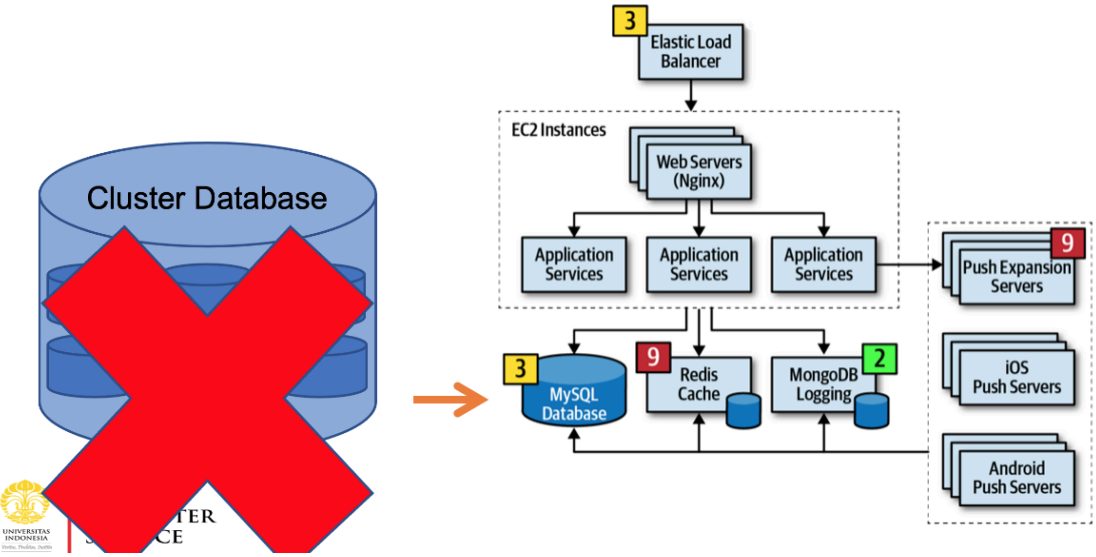
Risk Storming: Mitigation



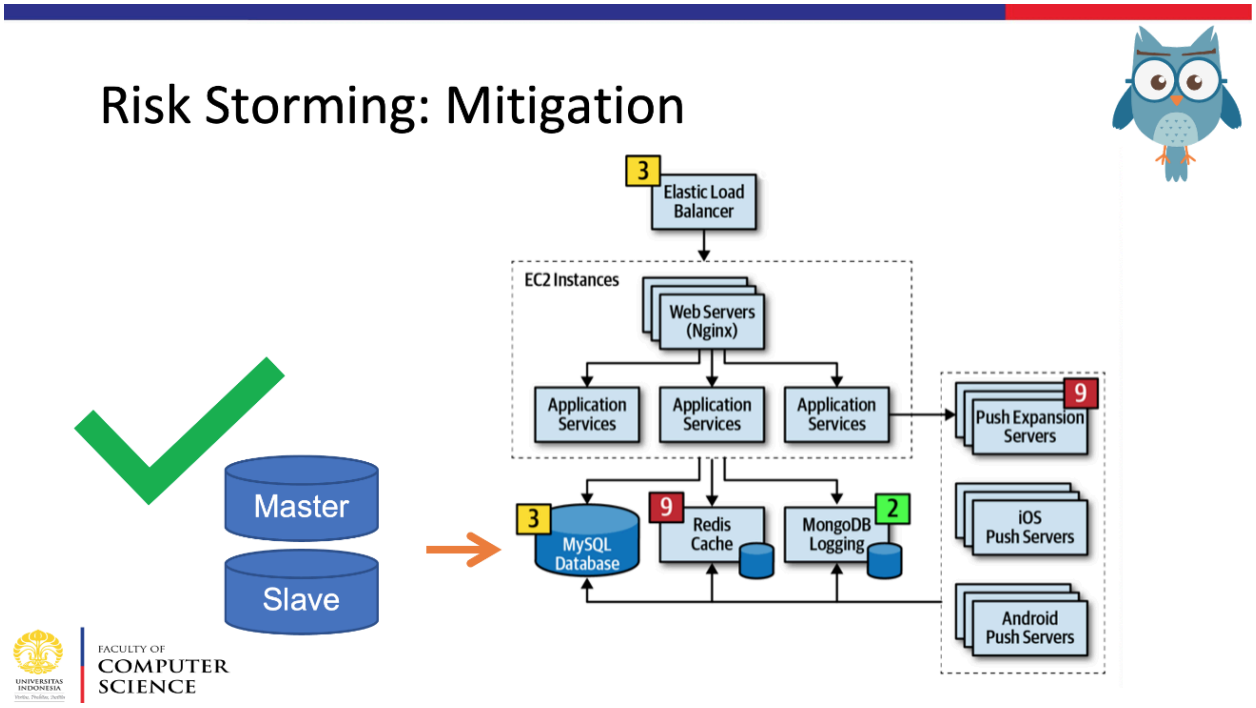
FACULTY OF COMPUTER SCIENCE

Regardless of the changes required in the architecture, this activity usually incurs additional cost. For that reason, key stakeholders typically decide whether the cost outweighs the risk. For example, suppose that through a risk storming session the central database was identified as being medium risk (4) with regard to overall system availability. In this case, the participants agreed that clustering the database, combined with breaking the single database into separate physical databases, would mitigate that risk. However, while risk would be significantly reduced, this solution would cost $20,000.

Risk Storming: Mitigation



The architect would then conduct a meeting with the key business stakeholder to discuss this trade-off. During this negotiation, the business owner decides that the price tag is too high and that the cost does not outweigh the risk. Rather than giving up, the architect then suggests a different approach—what about skipping the clustering and splitting the database into two parts? The cost in this case is reduced to $8,000 while still mitigating most of the risk. In this case, the stakeholder agrees to the solution.



The previous scenario shows the impact risk storming can have not only on the overall architecture, but also with regard to negotiations between architects and business stakeholders. Risk storming, combined with the risk assessments described at the start of this chapter, provide an excellent vehicle for identifying and tracking risk, improving the architecture, and handling negotiations between key stakeholders.

Example: A Nurse Diagnostic System

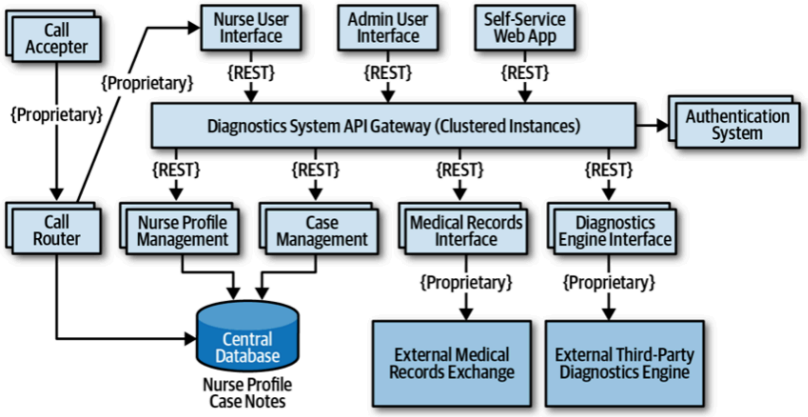
Risk Storming Example: A Nurse Diagnostics System

- The system will use a third-party diagnostics engine that serves up questions and guides the nurses or patients regarding their medical issues.
- Patients can either call in using the call center to speak to a nurse or choose to use a self-service website that accesses the diagnostic engine directly, bypassing the nurses.
- The system must support 250 concurrent nurses nationwide and up to hundreds of thousands of concurrent self-service patients nationwide.
- Nurses can access patients' medical records through a medical records exchange, but patients cannot access their own medical records.
- The system must be HIPAA compliant with regard to the medical records. This means that it is essential that no one but nurses have access to medical records.
- Outbreaks and high volume during cold and flu season need to be addressed in the system.
- Call routing to nurses is based on the nurse's profile (such as bilingual needs).
- The third-party diagnostic engine can handle about 500 requests a second

FACULTY OF COMPUTER SCIENCE

Risk Storming Example: A Nurse Diagnostics System

- Availability
- Elasticity/ scalability
- Security



FACULTY OF COMPUTER SCIENCE

Figure 20-9. High-level architecture for nurse diagnostics system example

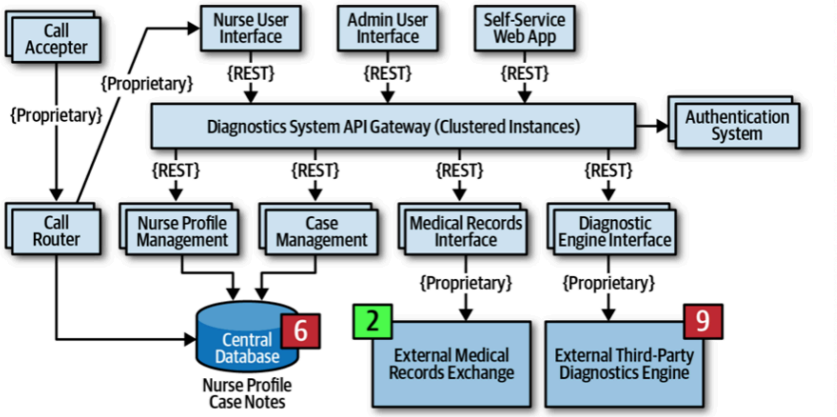
The architect of the system created the high-level architecture illustrated in Figure 20-9. In this architecture there are three separate web-based user interfaces: one for self-service, one for nurses receiving calls, and one for administrative staff to add and maintain the nursing profile and configuration settings. The call center portion of the system consists of a call accepter which receives calls and the call router which routes calls to the next available nurse based on their profile (notice how the call router accesses the central database to get nurse profile information).

Central to this architecture is a diagnostics system API gateway, which performs security checks and directs the request to the appropriate backend service.

There are four main services in this system: a case management service, a nurse profile management service, an interface to the medical records exchange, and the external third-party diagnostics engine. All communications are using REST with the exception of proprietary protocols to the external systems and call center services.

The architect has reviewed this architecture numerous times and believes it is ready for implementation. As a self-assessment, study the requirements and the architecture diagram in Figure 20-9 and try to determine the level of risk within this architecture in terms of availability, elasticity, and security. After determining the level of risk, then determine what changes would be needed in the architecture to mitigate that risk.

Risk Storming Example



FACULTY OF COMPUTER SCIENCE

Figure 20-10. Availability risk areas

Availability

During the first risk storming exercise, the architect chose to focus on availability first since system availability is critical for the success of this system. After the risk storming identification and collaboration activities, the participants came up with the following risk areas using the risk matrix (as illustrated in Figure 20-10):

- The use of a central database was identified as high risk (6) due to high impact (3) and medium likelihood (2).
- The diagnostics engine availability was identified as high risk (9) due to high impact (3) and unknown likelihood (3).
- The medical records exchange availability was identified as low risk (2) since it is not a required component for the system to run.

- Other parts of the system were not deemed as risk for availability due to multiple instances of each service and clustering of the API gateway.
-

During the risk storming effort, all participants agreed that while nurses can manually write down case notes if the database went down, the call router could not function if the database were not available. To mitigate the database risk, participants chose to break apart the single physical database into two separate databases: one clustered database containing the nurse profile information, and one single instance database for the case notes. Not only did this architecture change address the concerns about availability of the database, but it also helped secure the case notes from admin access. Another option to mitigate this risk would have been to cache the nurse profile information in the call router. However, because the implementation of the call router was unknown and may be a third-party product, the participants went with the database approach.

Mitigating the risk of availability of the external systems (diagnostics engine and medical records exchange) is much harder to manage due to the lack of control of these systems. One way to mitigate this sort of availability risk is to research if there is a published service-level agreement (SLA) or service-level objective (SLO) for each of these systems. An SLA is usually a contractual agreement and is legally binding, whereas an SLO is usually not. Based on research, the architect found that the SLA for the diagnostics engine is guaranteed to be 99.99% available (that's 52.60 minutes of downtime per year), and the medical records exchange is guaranteed at 99.9% availability (that's 8.77 hours of downtime per year). Based on the relative risk, this information was enough to remove the identified risk.

Architecture Modification: Availability

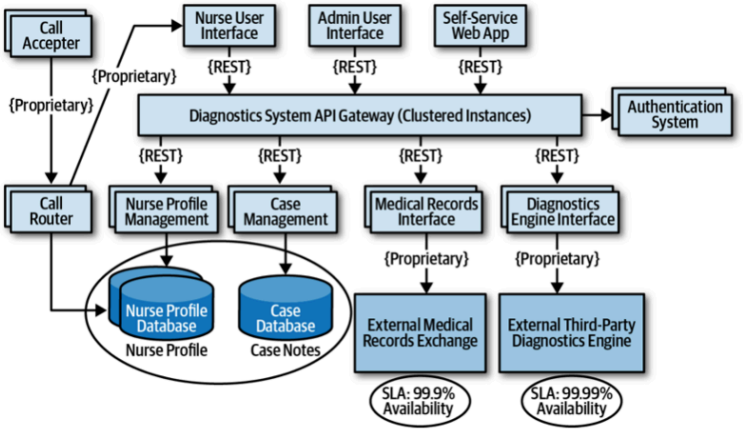


Figure 20-11. Architecture modifications to address availability risk

Notice that two databases are now used, and also the SLAs are published on the architecture diagram.

Architecture Modification: Scalability

- There are only 250 nurses
- Participants were concerned about outbreaks and flu season, when anticipated load on the system would significantly increase.
- With only 500 requests per second, the participants calculated that there was no way the diagnostics engine interface could keep up with the anticipated throughput, particularly with the current architecture utilizing REST as the interface protocol.

FACULTY OF COMPUTER SCIENCE

One way to mitigate this risk is to leverage asynchronous queues (messaging) between the API gateway and the diagnostics engine interface to provide a back-pressure point if calls to the diagnostics engine get backed up. While this is a good practice, it still doesn't mitigate the risk, because nurses (as well as self-service patients) would be waiting too long for responses from the diagnostics engine, and those requests would likely time out. Leveraging what is known as the Ambulance Pattern would give nurses a higher priority over self-service. Therefore two message channels would be needed. While this technique helps mitigate the risk, it still doesn't address the wait times. The participants decided that in addition to the queuing technique to provide back-pressure, caching the particular diagnostics questions related to an outbreak would remove outbreak and flu calls from ever having to reach the diagnostics engine interface.

The corresponding architecture changes are illustrated in Figure 20-12. Notice that in addition to two queue channels (one for the nurses and one for self-service patients), there is a new service called the Diagnostics Outbreak Cache Server that handles all requests related to a particular outbreak or flu-related question. With this architecture in place, the limiting factor was removed (calls to the diagnostics engine), allowing for tens of thousands of concurrent requests. Without a risk storming effort, this risk might not have been identified until an outbreak or flu season happened.

Architecture Modification: Elasticity

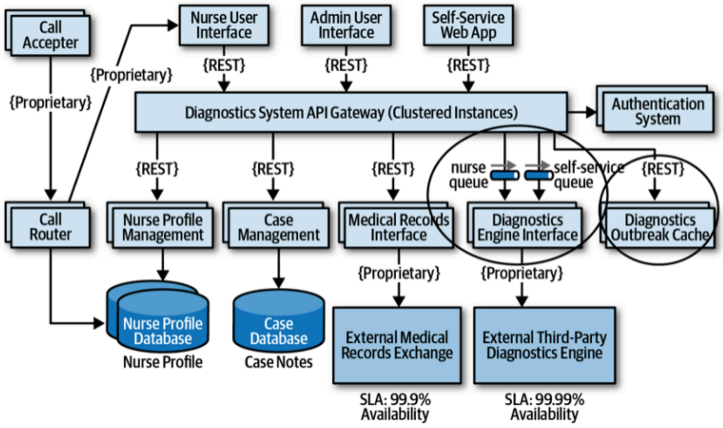


Figure 20-12. Architecture modifications to address elasticity risk

FACULTY OF COMPUTER SCIENCE

Security

Architecture Modification: Security

- During the risk storming, the participants all identified the Diagnostics System API gateway as a high security risk (6).
- The rationale for this high rating was the high impact of admin staff or self-service patients accessing medical records (3) combined with medium likelihood (2).
- Likelihood of risk occurring was not rated high because of the security checks for each API call, but still rated medium because all calls (self-service, admin, and nurses) are going through the same API gateway.

FACULTY OF COMPUTER SCIENCE

The participants all agreed that having separate API gateways for each type of user (admin, self-service/diagnostics, and nurses) would prevent calls from either the admin web user

interface or the self-service web user interface from ever reaching the medical records exchange interface.

Architecture Modification: Security

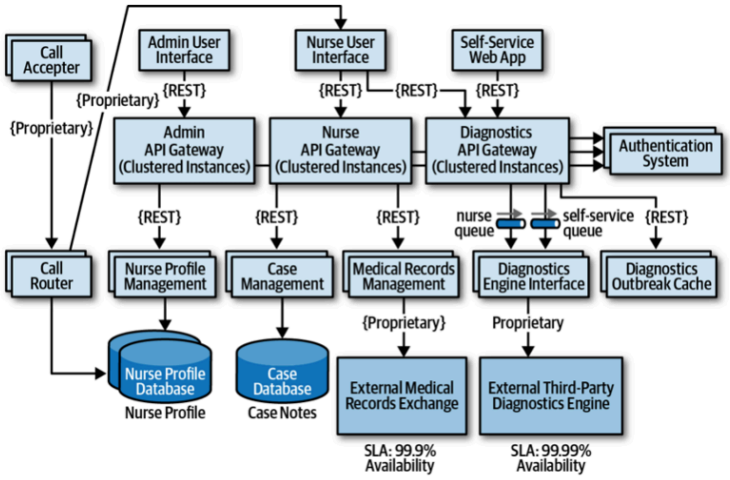


Figure 20-13. Final architecture modifications to address security risk

Architecture: Before - After

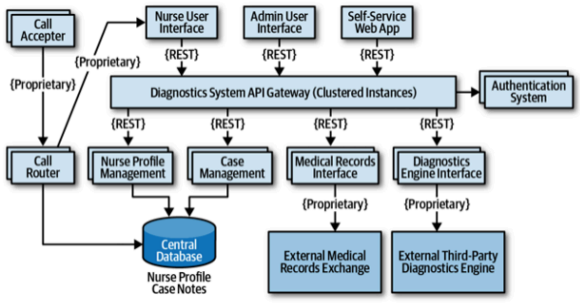


Figure 20-9. High-level architecture for nurse diagnostics system example

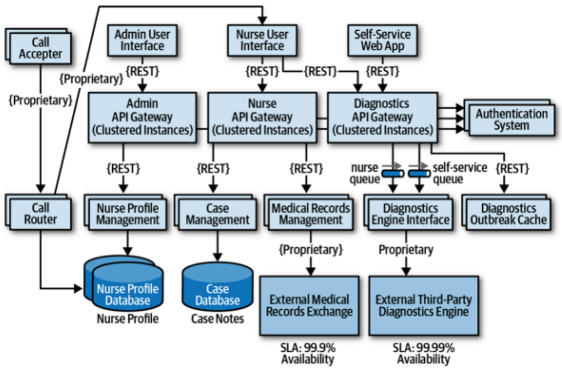
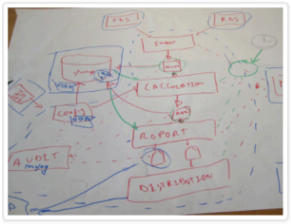


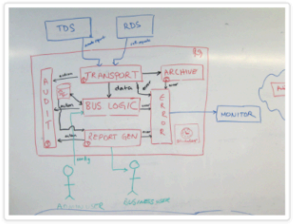
Figure 20-13. Final architecture modifications to address security risk

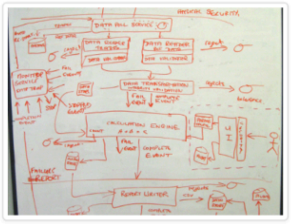
9. Visualizing Software Architecture

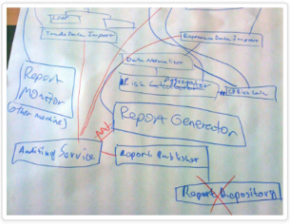
Visualizing Software Architecture

- Visualization is a way to communicate your architecture









FACULTY OF COMPUTER SCIENCE

There is several ways to communicate. A picture worth a thousands words. In this course we use the c4model.

Ask somebody in the building industry to visually communicate the architecture of a building and you'll be presented with site plans, floor plans, elevation views, cross-section views and detail drawings. In contrast, ask a software developer to communicate the software architecture of a software system using diagrams and you'll likely get a confused mess of boxes and lines ... inconsistent notation (colour coding, shapes, line styles, etc), ambiguous naming, unlabelled relationships, generic terminology, missing technology choices, mixed abstractions, etc.

As an industry, we do have the Unified Modeling Language (UML), ArchiMate and SysML, but asking whether these provide an effective way to communicate software architecture is often irrelevant because many teams have already thrown them out in favour of much simpler "boxes and lines" diagrams. Abandoning these modelling languages is one thing but, perhaps in the race for agility, many software development teams have lost the ability to communicate visually.

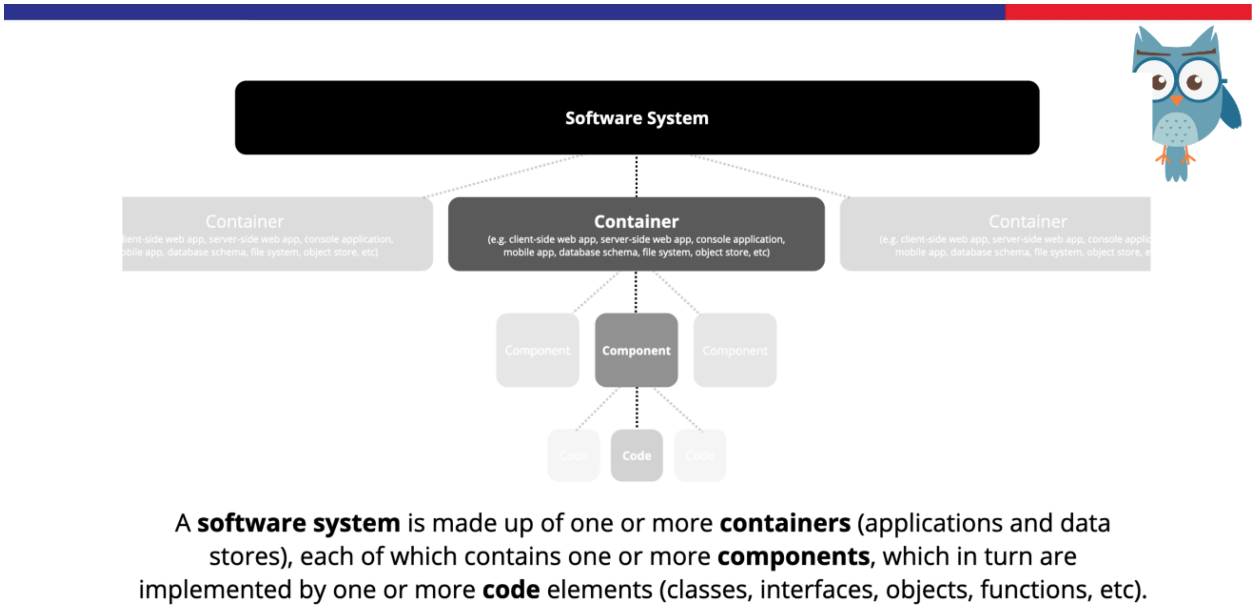
C4 Model

The C4 model is...

1. A set of hierarchical abstractions (software systems, containers, components, and code).
2. A set of hierarchical diagrams (system context, containers, components, and code).
3. Notation independent.
4. Tooling independent.

Uses and benefits

The C4 model is an easy to learn, developer friendly approach to software architecture diagramming. Good software architecture diagrams assist with communication inside/outside of software development/product teams, efficient onboarding of new staff, architecture reviews/evaluations, risk identification (e.g. risk-storming), threat modelling, etc.



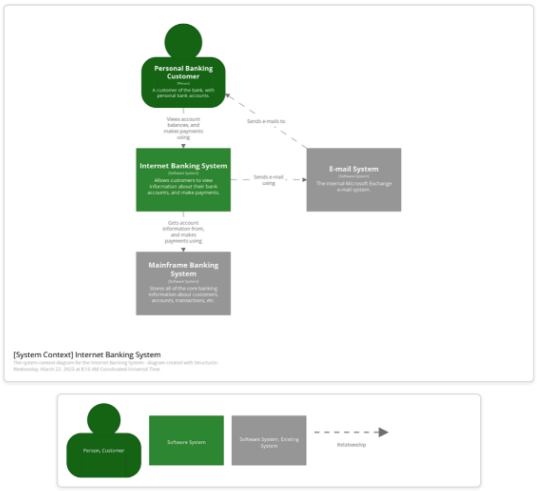
A **software system** is made up of one or more **containers** (applications and data stores), each of which contains one or more **components**, which in turn are implemented by one or more **code** elements (classes, interfaces, objects, functions, etc).

FACULTY OF COMPUTER SCIENCE

In order to create these maps of your code, we first need a common set of abstractions to create a ubiquitous language that we can use to describe the static structure of a software system. A software system is made up of one or more containers (applications and data stores), each of which contains one or more components, which in turn are implemented by one or more code elements (classes, interfaces, objects, functions, etc). And people may use the software systems that we build.

Software System / System Context Diagram

- A System Context diagram is a good starting point for diagramming and documenting a software system, allowing you to step back and see the big picture.
- Draw a diagram showing your system as a box in the centre, surrounded by its users and the other systems that it interacts with.



FACULTY OF COMPUTER SCIENCE

Scope: A single software system.

Primary elements: The software system in scope.
Supporting elements: People (e.g. users, actors, roles, or personas) and software systems (external dependencies) that are directly connected to the software system in scope. Typically these other software systems sit outside the scope or boundary of your own software system, and you don't have responsibility or ownership of them.

Intended audience: Everybody, both technical and non-technical people, inside and outside of the software development team.

Recommended for most teams: Yes.

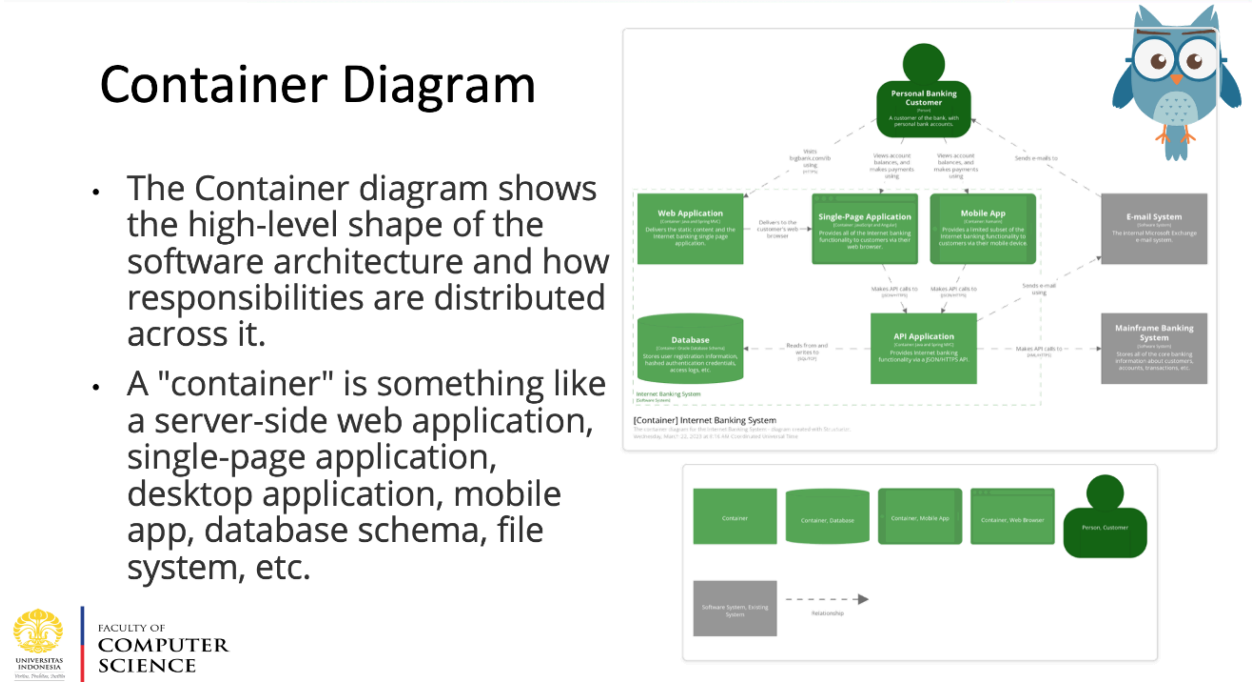
A software system is the highest level of abstraction and describes something that delivers value to its users, whether they are human or not. This includes the software system you are modelling, and the other software systems upon which your software system depends (or vice versa).

Unfortunately the term "software system" is the hardest of the C4 model abstractions to define, and this isn't helped by the fact that each organisation will also have their own terminology for describing the same thing, typically using terms such as "application", "product", "service", etc. One way to think about it is that a software system is something a single software development team is building, owns, has responsibility for, and can see the internal implementation details of. Perhaps the code for that software system resides in a single source code repository, and anybody on the team is entitled to modify it. In many cases, the boundary of a software system will correspond to the boundary of a single team. It may also be the case that everything inside the boundary of a software system is deployed at the same time.

A System Context diagram is a good starting point for diagramming and documenting a software system, allowing you to step back and see the big picture. Draw a diagram showing your system as a box in the centre, surrounded by its users and the other systems that it interacts with.

Detail isn't important here as this is your zoomed out view showing a big picture of the system landscape. The focus should be on people (actors, roles, personas, etc) and software systems rather than technologies, protocols and other low-level details. It's the sort of diagram that you could show to non-technical people.

Container Diagram



Scope: A single software system.

Primary elements: Containers within the software system in scope.
Supporting elements: People and software systems directly connected to the containers.

Intended audience: Technical people inside and outside of the software development team; including software architects, developers and operations/support staff.

Recommended for most teams: Yes.

Notes: This diagram says nothing about clustering, load balancers, replication, failover, etc because it will likely vary across different environments (e.g. production, staging, development, etc). This information is better captured via one or more deployment diagrams.

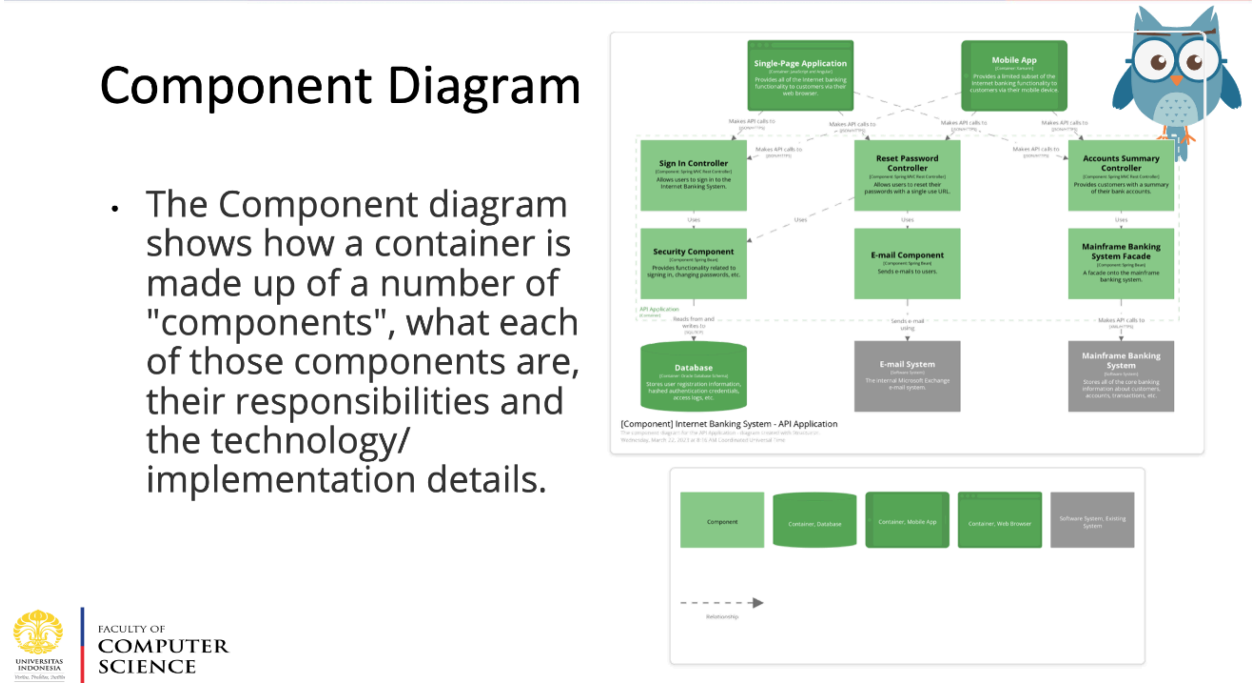
Not Docker! In the C4 model, a container represents an application or a data store. A container is something that needs to be running in order for the overall software system to work. In real terms, a container is something like:

- Server-side web application: A Java EE web application running on Apache Tomcat, an ASP.NET MVC application running on Microsoft IIS, a Ruby on Rails application running on WEBrick, a Node.js application, etc.
- Client-side web application: A JavaScript application running in a web browser using Angular, Backbone.JS, jQuery, etc.
- Client-side desktop application: A Windows desktop application written using WPF, an OS X desktop application written using Objective-C, a cross-platform desktop application written using JavaFX, etc.
- Mobile app: An Apple iOS app, an Android app, a Microsoft Windows Phone app, etc.
- Server-side console application: A standalone (e.g. "public static void main") application, a batch process, etc.
- Serverless function: A single serverless function (e.g. Amazon Lambda, Azure Function, etc).
- Database: A schema or database in a relational database management system, document store, graph database, etc such as MySQL, Microsoft SQL Server, Oracle Database, MongoDB, Riak, Cassandra, Neo4j, etc.
- Blob or content store: A blob store (e.g. Amazon S3, Microsoft Azure Blob Storage, etc) or content delivery network (e.g. Akamai, Amazon CloudFront, etc).
- File system: A full local file system or a portion of a larger networked file system (e.g. SAN, NAS, etc).
- Shell script: A single shell script written in Bash, etc.

Once you understand how your system fits in to the overall IT environment, a really useful next step is to zoom-in to the system boundary with a Container diagram. A "container" is something like a server-side web application, single-page application, desktop application, mobile app, database schema, file system, etc. Essentially, a container is a separately runnable/deployable unit (e.g. a separate process space) that executes code or stores data.

The Container diagram shows the high-level shape of the software architecture and how responsibilities are distributed across it. It also shows the major technology choices and how the containers communicate with one another. It's a simple, high-level technology focused diagram that is useful for software developers and support/operations staff alike.

Component Diagram



Scope: A single container.

Primary elements: Components within the container in scope.
Supporting elements: Containers (within the software system in scope) plus people and software systems directly connected to the components.

Intended audience: Software architects and developers.

Recommended for most teams: No, only create component diagrams if you feel they add value, and consider automating their creation for long-lived documentation.

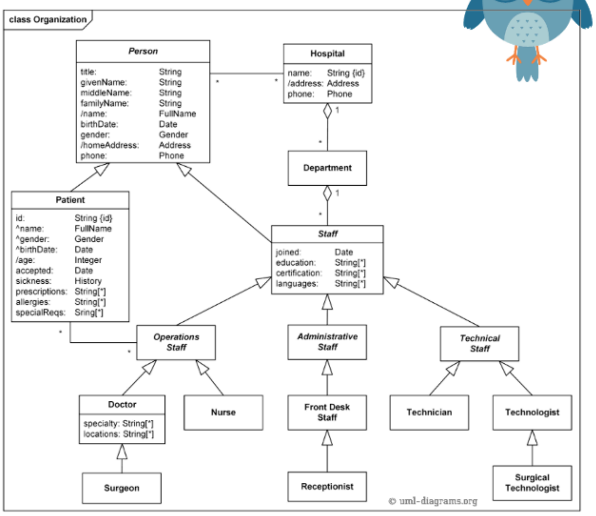
The word "component" is a hugely overloaded term in the software development industry, but in this context a component is a grouping of related functionality encapsulated behind a well-defined interface. If you're using a language like Java or C#, the simplest way to think of a component is that it's a collection of implementation classes behind an interface. Aspects such as how those components are packaged (e.g. one component vs many components per JAR file, DLL, shared library, etc) is a separate and orthogonal concern.

An important point to note here is that all components inside a container typically execute in the same process space. In the C4 model, components are not separately deployable units.

The Component diagram shows how a container is made up of a number of "components", what each of those components are, their responsibilities and the technology/implementation details.

Code Diagram

- This is an optional level of detail and is often available on-demand from tooling such as IDEs.
- Ideally this diagram would be automatically generated using tooling (e.g. an IDE or UML modelling tool), and you should consider showing only those attributes and methods that allow you to tell the story that you want to tell.



FACULTY OF COMPUTER SCIENCE

Scope: A single component.

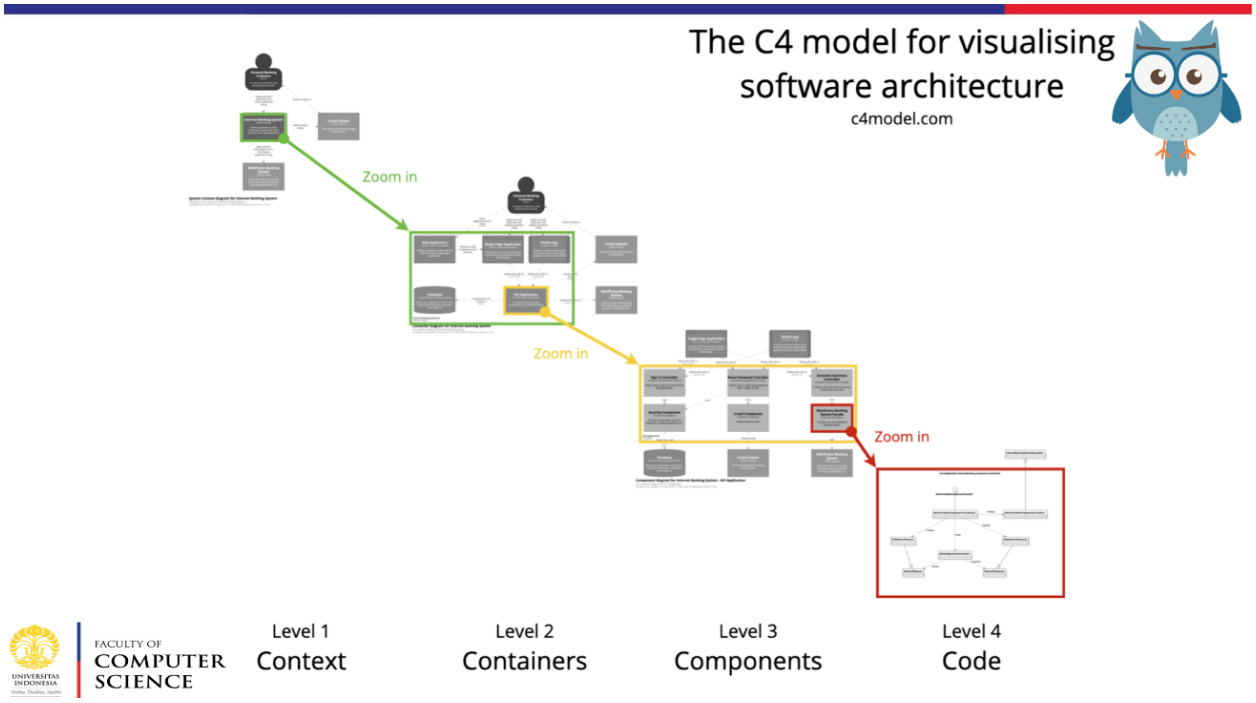
Primary elements: Code elements (e.g. classes, interfaces, objects, functions, database tables, etc) within the component in scope.

Intended audience: Software architects and developers.

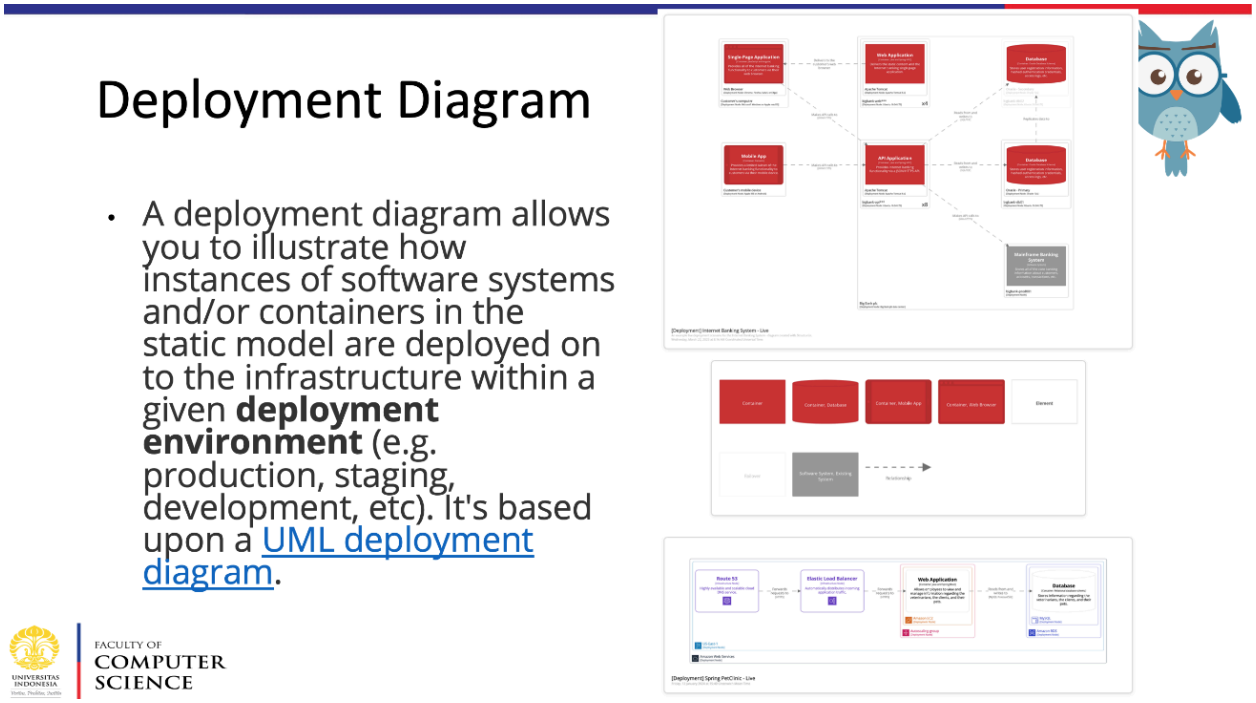
Recommended for most teams: No, particularly for long-lived documentation because most IDEs can generate this level of detail on demand.

Finally, you can zoom in to each component to show how it is implemented as code; using UML class diagrams, entity relationship diagrams or similar.

This is an optional level of detail and is often available on-demand from tooling such as IDEs. Ideally this diagram would be automatically generated using tooling (e.g. an IDE or UML modelling tool), and you should consider showing only those attributes and methods that allow you to tell the story that you want to tell. This level of detail is not recommended for anything but the most important or complex components.



Deployment Diagram



Scope: One or more software systems within a single deployment environment (e.g. production, staging, development, etc).

Primary elements: Deployment nodes, software system instances, and container instances.
Supporting elements: Infrastructure nodes used in the deployment of the software system.

Intended audience: Technical people inside and outside of the software development team; including software architects, developers, infrastructure architects, and operations/support staff.

Additionally we should also provide deployment diagram. A deployment diagram allows you to illustrate how instances of software systems and/or containers in the static model are deployed on to the infrastructure within a given deployment environment (e.g. production, staging, development, etc). It's based upon a UML deployment diagram.

A deployment node represents where an instance of a software system/container is running; perhaps physical infrastructure (e.g. a physical server or device), virtualised infrastructure (e.g. IaaS, PaaS, a virtual machine), containerised infrastructure (e.g. a Docker container), an execution environment (e.g. a database server, Java EE web/application server, Microsoft IIS), etc. Deployment nodes can be nested.

You may also want to include infrastructure nodes such as DNS services, load balancers, firewalls, etc.

Feel free to use icons provided by Amazon Web Services, Azure, etc to complement your deployment diagrams ... just make sure any icons you use are included in your diagram key/legend.

Tutorial B: Visualizing and Architectural Risk

This tutorial is started as a group discussion, but **the submission and evaluation will be individually**.

Group Discussion:

Groups Discussion

- Discuss the current architecture of your group project
- Draw the ***context diagram***, ***container diagram*** *and* ***deployment diagram.***
- Put it in your Readme.md in your group repository (Deliverable G.1)
- (Later, individually, draw the ***component*** and ***code diagram*** of your own works, put it on the readme.md or separate files). (Deliverable Individual)
- Next, do the Risk Analyzing, modify your Architecture. Re draw the Context and Container Diagram.
- Add the new future software architecture in your Readme.md (Deliverable G.2)
- Add some notes 2-3 paragraph explanation of the ***risk analysis*** and ***architecture modification justification*** in the Readme.md (Deliverable G.3)

FACULTY OF COMPUTER SCIENCE

Discuss your groups project current architecture. Discuss the context, container and also the deployment diagram. Draw it in your favorite drawing editor. Download an image and add it in to your group project repository.

Commit and push with message: "1. The current architecture of group …. , the context, container and deployment diagram".

Discuss your group project architectural risk. Be creative and innovative, imagine that your project is having a great success. Think and discuss what will be the risk of the current architecture. Apply the technique of *Risk Storming* as presented before.

Draw the expected new future architecture. Put it in your Readme.md

Commit and push with message: "2. The future architecture of group … "

Discuss it and the write a well written an concise explanation why the *risk storming* technique is applied. Write it your Readme.md project repository.

Commit and push with message: "3. Explanation of risk storming of group … "

Individual works:

Based on the container diagram of your group. Expand it with drawing the component diagram and code diagram related to your work the group project. Draw at least one container diagram, and several (up to 4) code diagram individually. Draw more diagram may give you more score. Put it in your group project Readme.md repository.

Commit and push with message: "4. Individual container diagram … (with its code diagram) of group … "

Submission:

Submission is conducted individually. In the submission form, put:

- Your project repository,
- Link commit of your group works, (1,2,3)
- Link commit of your individual works,

Grading Scheme

- Students should follow the instructions
- Make sure to do all the commits as requested.
- Do not squash your commit during the merge process. It may destroy your commits history and the teaching assistants cannot evaluate it.
- Push your work to the repository.
- Give access to the teaching assistants.
- Submit all the information of repository and link commits to scele as requested.

Scale

All components will be scored on discrete scale 0,1,2,3,4. Grade 4 is the highest grade, equivalent to A. Score 0 is for not submitting. No partial score.

Components

- 60% - Group Work (1,2,3)
  - 10 % - Commit 1
  - 10 % - Commit 2
  - 20 % - Commit 3
  - 20 % - Coherence of individual commit and merge-conflict management.
- 40% - Individual Work (4)
  - 20 % - one component diagram
  - 20 % - one or more code diagrams
- Bonus 10%: if student create another component diagram with its code diagram

Rubrics of Group work

	Score 4	Score 3	Score 2	Score 1
Commit 1:	All requested diagrams are presented well	All diagrams are presented, but there are some significant error	One or more diagram are missing	Unclear submission
Commit 2:	All requested diagrams are presented well and the changes are clearly seen	All diagrams are presented, but there are some significant error	One or more diagram are missing	Unclear submission
Commit 3:	Explanations are clear and concise, all the step of risk storming are described well.	Explanations seem are not complete, missing small steps or explanation is not cleare	Explanations is written badly. But some points are mentioned.	Unclear submission
Coherence of individual commit and merge-conflict management.	Commit and merge conflict is handled properly, nothing is broken or overwritten.	No score for 3 (it is either 4, 2 or 1, 0)	There is some issue in merge-conflct where some commits has been overwriiten improperly	The repository is chotic, merge history is terrible.

Rubrics of Individual work

	Score 4	Score 3	Score 2	Score 1
Component diagram	The diagram is presented clearly, and there signifant correlation with the container diagram	The component diagram is drawed well, but not clear correlation with the container diagram	The component diagram is not properly drawed.	Unclear submission
Code diagram	All related code diagrams are presented well	The code diagram is presented but not complete.	The code diagram is not properly drawed	Unclear submission

Bonus	Additional component diagram with its code diagram are presented well	Some thing is missing, such as the code diagram is not complete	The additional diagram is not properly drawed	Unclear submission